EVOLVABLE ROBOTS, UNIVERSAL DECISION DIAGRAMS, QUANTUM LOGIC, INTELLIGENT ANIMATED PUPPETS, AND OTHER MACHINE LEARNING PROJECTS FOR INQUISITIVE MINDS

Marek Perkowski, Anas Al-Rabadi and Alan Mishchenko

Contents

List of Figures

List of Tables

Preface

This section, one page, will characterize this book as a capstone projects manual with innovative structure and approach.

1. Lisp can be used to describe, simulate, synthesize MVL circuits, both in their logic, spectral and mixed (wavelet) domains. It is a fast way of prototyping, because not many tools for MVL are available. Most algorithms and corresponding hardware machines are realized here in software, hardware-software codesign is emphasized.

2. We emphasize complete designs, not pieces. Therefore FSMs, and larger blocks.

3. We emphasize student's creativity and problem-solving, because MVL, spectral diagrams, and other new logic systems lend themselves to these.

4. We do not put heavy emphasis on mechanical or analog electronic/sensors construction. As much as possible should be taken/borrowed from existing toys and other commercially available inexpensive devices. Our emphasis is on new techniques of software and digital hardware design. As you will see, one can now assemble a complete robot with several arms and head from inexpensive construction sets. Hence our emphasis is on machine learning, software and system integration.

Acknowledgments

The authors would like to thank many people who somehow influenced the writing of this book, mostly by teaching us, the people who encouraged us to write it, and the people who actually helped to produce the book.

Prof. Wieslaw Traczyk, Dr. Piotr Misiurewicz, Andrzej Rydzewski and Henryk Kruszynski from Warsaw Technical University have been our professors and collegues and also contributed some early design ideas and exercises. Dr. Kruszynski designed the first voice-controlled robot. Dr. Jan Baranowski helped to write the first Lisp tutorial. Dr. Doug Hall and Mr. James Brown from PSU were good critics of the early drafts.

Some of our ideas were inspired by Hugo De Garis and other folks in Evolvable Hardware community.

Kevin Stanton was the spiritus movens of the PSUBOT project. Shiliang Wang, and William Kelly Spiller and Cecilia Espinosa wrote image processing and pattern recognition software. Eli Cabelli build the MUVAL Lobster, and ... helped to build the Kali Ape. Mike Burns worked on path planning.

Finally, Mateusz Jan Perkowski draw most of the figures using XFIG, Summit and Visio tools, and helped to format using Latex. Ugur Kalay and Edith Vedanyagam (????) helped with proofreading, figures and tools. Justin Kam helped with several robotics projects.

We are very grateful to all these people, but we are the only ones responsible for all remaining errors.

Chapter 1

Introduction. What Will You Learn from this Book.

1.1 Using Computers to Design Computers

In recent years many new technologies and design methodologies have been introduced to the fields of computing, cybernetics and electronic design. They will all allow the design of much more **intelligent robots** and animated puppets - **animatrons**.

This book will teach you to design many innovative components of robotics systems: mechanics, logic circuits, software and complete subsystems. You will learn to understand on what principles some of these devices and systems operate, and also how you can practically design your own circuits, interfaces and programs. Another goal will be to teach you to write robotics software, adapting existing software, as well as to develop methods and original algorithms necessary to develop such software. In contrast to all books published until now, this book will introduce not only the standard binary logic, but also multiple-valued and quantum logics as fundaments of various robotic applications.

What is then the major goal of this book?

Although we will teach you a computer language Lisp, how to use it for solving problems based on searching space of possible solutions, or how to use it to simulate and design robotic systems, this is not a standard Lisp textbook, nor it is a standard robotics book. Lisp is just an excellent match with multi-valued logic, quantum logic, formal methods, learning and many other ideas that we want to introduce here, but without too much of a student's effort.

Our belief is that experimentation and creativity are more important for young students than formal and systematic study, and our goal here is to make you a successful engineer by having first fun to learn. Multi-valued tools are already used for two-level logic design, and for controlling robots, so that multi-valued logic ideas are becoming more and more practical in various applications.

Many companies are looking for engineers who know both about software and hardware and that can design new systems that include elements of both. Robotics engineers need to understand hardware and they should be also able to develop efficient software. All this in the world of very competitive and fast changing technology. Therefore, they must be profficient in performing research. So, in a sense, this will be a book about research. Again, robotics seems to be the source of excellent and not too difficult ideas for many types of research and development projects.

In graduate school research starts with class projects, next classes such as "Research" and "Reading and Conference", and finally, M.S. and Ph.D. theses. This book, however, is addressed to college juniors. Nowadays most U.S. Universities has classes called "Design" or "Capstone Projects", in which juniors or seniors design quite complicated hardware or software in close collaboration with industrial engineers and with help of their professors. At PSU, for instance, undergraduate students designed a wheelchair robot controlled by voice, a with voice recognition system, a digital camera, and many, many programs. Some of these students wrote papers for international conferences, and next delivered them to international audiences, some other students created companies based on these project ideas, or won regional design competitions. Proving thus, that the goals set by the University to "Capstone projects", although ambitious, can be reached by motivated and hardworking junior or even freshman-level students.

We believe, however, that even if you are reading this book as a freshman, you are ready to start learning class material as if you were a researcher, and not just a consumer of knowledge created by somebody else for you to use.

As mentioned, robot design cannot be nowadays performed without some mastery of using computer software, as an aid in your thinking and problem solving. We will assume here that you have already taken first course in digital design and you are familiar with using computers and Internet. You are also definitely familiar with software for games, drawing tools, or computer communication, but did you ever thought that computer can help you to understand better a problem to be solved? Did you think that computer can be more than just a large calculator

helping you with aritmetical operations? That it can actually help you to perform some practical design task faster and better than without it? Actually modern computers are now all designed with the help of other computers and we cannot even think about designing them "by hand", like it was common even only fifteen years ago.

There will be several ways emphasized in this book to make your learning process both deep and fun:

1. WWW searches.
2. essays.
3. design problems.
4. programming exercises.
5. research projects.

WWW searches will teach you how to find relevant material prepared by industrial companies, universities or research institutions. You can even connect to National Science Foundation, NASA, or military research labs. You can learn about newest patents in some countries. Searching information on Web will help you to write short essays, for instance on newest design tools or newest design technologies.

Next, you will find in this book very many problems for your individual solution. We believe in learning by doing and we have good experience with students who like to solve many design problems. Basically, this is the only way to learn the material. These problems are short, and most of them do not require to use computer.

Programming exercises will first teach you programming language Lisp. This is the so-called **functional language**, which means that programs resemble functions, and it has very interesting properties that allow beginners to write quite complex programs which would be not possible in other language. These programs are not very efficient but our goal will be to learn ideas, rather than to develop ready-to-sell software. We will be **fast prototyping** software-hardware systems for robotics.

The most important component of this class are the **group research projects**. The groups will be from 3 to 5 students, and the duration is three quarters. The project's documentation is usually the "whole book" which the students often show to the companies considering hiring them. These reports document all research, ideas, programs, tests, debugging hints, schematics, source codes, and often videos, transparencies, animations, and photographs of working devices and systems. Some projects are done with students from other departments, for instance from School of Bussiness or from Mechanical Engineering.

We will first describe the programming environment in which you will work, and we will follow with the explanation of the role of projects in this class,

1.2 Searching World Wide Web

You can search for almost any information on the Web Wide Web (WWW). We assume that you know how to use programs Netscape or Explorer. After loading Netscape, you click Net Search button and you are in Search Engines page. Select Alta Vista, as a page that has most information on science and technology. Now to perform the search, you have to type the keywords. Suppose your name is John Smith and you want to find all information on Johns Smiths on the WWW. You type "John Smith" and the system will respond with links to all WWW pages that have "John Smith" in them. If you will type John Smith then the system will find you all pages that include word John or word Smith in them. Finally, if you would type +John +Smith you will find all pages that have both John and Smith in them, but John can be McEnroe, and Smith can be Jack. Try to limit your search to avoid pages with too much irrelevant information. We will use WWW to find information useful to your essays and projects. Read carefully pages starting from Alta Vista WWW Page to learn about more sophisticated ways of using keywords for searching.

Look to the first author's WWW Page:

http://www.ee.pdx.edu/ mperkows/

for a starting information related to this book and projects. The CDROM that accompanies this book includes also many good links as well as many other information useful for your projects.

1.3 Environment for Research

Now that we know what goals we want to achieve, let us look back what is our experience with trying to achieve these goals in our classes. Why we were successful in few cases, why we were not in the others?

Both authors have taught various logic, design and software classes in Poland, Ukraine, Russia, USA, France, Holland, Germany and Japan, for university and industrial students, and we found that various groups of students

Search the Web for documents in

Search
Advanced
Usenet

Tip: To restrict a search to a host, try: **medicine host:stanford.edu**. More tips

Click Here!

28 matches were found.

1. No Title
[**URL:** www.ida.liu.se/~krzku/DSD-98-program.html]
Preliminary Program Schedule. Tuesday, August 25, 1998. Registration. Opening Session. Keynote Speech. 10:45-12:45 Combinational Logic Design....
Last modified 28-Apr-98 - page size 12K - in English [Translate]

2. IEEE Transactions on Computers, Volume 45
[**URL:** joinus.comeng.chungnam.ac.kr/~dolphin/db...ls/tc/tc45.html]
IEEE Transactions on Computers, Volume 45. Volume 45, Number 1, January 1996. Cellular Automata. S. Nandi, P. Pal Chaudhuri: Analysis of Periodic and...
Last modified 12-May-98 - page size 47K - in English [Translate]

3. Texture Analysis Using Logical Transform
[**URL:** exodo.upr.clu.edu/~pkumar/proposal.html]
Study of Using Logical Transforms for Texture Analysis. JUSTIFICATION. One of the problems in Image Processing is the classification of a textured image...
Last modified 29-Jul-97 - page size 14K - in English [Translate]

4. No Title
[**URL:** i12www.ira.uka.de/~reiner/reiner.bib]
STRING{fac = "Formal Aspects of Computing"} @STRING{amai = "Annals of Mathematics and Artificial Intelligence"} @STRING{jsl = "Journal of Symbolic Logic"}.
Last modified 20-Mar-98 - page size 492K - in English (HZ) [Translate]

5. Shimizu Lab.,sotuken2.
[**URL:** www.ja4.cs.gunma-u.ac.jp/labo/sotuken2.html]
.... S.60. ....
Last modified 16-Feb-98 - page size 155K - in Japanese (EUC)

6. MVL-TC Bulletin, Vol.18 No.3

Amazon.com suggests
Titles about +Perkowski +...
Books of the Day

ABCNEWS.com
U.S.: Land of Guns and Death
More GM Plants Close

Zones
Careers
Entertainment
Finance
Health
Travel

Services
Browse Categories
Find a Business
Find a Person
Free Email
Translation

International
Our Search Network
Search in Chinese
Search in Japanese
Search in Korean

Figure 1.1: Looking for WWW pages that include both words "Perkowski" and "Logic Design" in them, using Alta Vista search engine on Netscape. Top of the screen

[**URL:** wwwj1.comp.eng.himeji-tech.ac.jp/~mvl/19.1.html]
IEEE MVL-TC Bulletin. VOLUME 19 Number 1 March 1998. 1. Chair's Message. This is my first message as chair of the MVL-TC. I feel honored to serve in this..
Last modified 7-Apr-98 - page size 24K - in English [Translate]

7. No Title
[**URL:** www.sigda.acm.org/Conferences/Conference.../DAC.papers.txt]
KEY: S denotes short paper, 15 minute presentations * denotes best paper candidate Tuesday, June 7, 1994 SESSION 1 Time: 10:30 to 11:30 Room: 6A EMBEDDED..
Last modified 9-Apr-97 - page size 30K - in English [Translate]

8. DAC94: References
[**URL:** www.mpi-sb.mpg.de/services/library/proce...ents/dac94.html]
DAC94: References
Last modified 14-Apr-98 - page size 49K - in English [Translate]

9. MVL-TC Bulletin, Vol.18 No.1
[**URL:** wwwj1.comp.eng.himeji-tech.ac.jp/~mvl/18.1.html]
IEEE MVL-TC Bulletin. VOLUME 18 Number 1 March 1997. 1. Chair's Message. The 27th International Symposium on Multiple-Valued Logic will be held in Nova...
Last modified 31-Mar-98 - page size 21K - in English [Translate]

10. Table of Contents DAC'94
[**URL:** ballade.cs.ucla.edu:8080/~kohcc/sigdacdr...4/dac94/toc.htm]
TABLE OF CONTENTS DAC'94. Executive Committee General Chair Message Technical Program Committee 1993 ACM/IEEE Best Paper Award 1994 ACM/IEEE Best Paper...
Last modified 3-Jul-97 - page size 39K - in English [Translate]

Pages: **1** 2 3 [>>]

word count: Perkowski: 1789; logic design: about 10000

Figure 1.2: Looking for WWW pages that include both words "Perkowski" and "Logic Design" in them, using Alta Vista search engine on Netscape. Bottom of the screen

have had various kinds of troubles in them. Some students, especially from Europe and from Far East, have good mathematics background, so they grasp concepts quickly, but have a hard time to apply them to practical software or hardware designs. Also, they are not able to use computer quickly to write software related to these classes: it becomes a major undertaking for them, and using C or C++ is too big a hurdle. The result is that they are not able to complete C or C++ software design project in the prescribed time.

On the other hand American university students are usually quite good programmers, but have often no patience or background to read purely theoretical and mathematically complex papers and translate these ideas to software. They are often lacking advanced knowledge of data structures and algorithms. They have also quite weak mathematical preparation and no desire to learn theoretical fundamentals. Use of advanced data structures and programming techniques is also often a surprise even to those who developed very advanced and commercially successful tools. They expect to "jump" to design robots immediately.

To make all these different types of students to learn as much as possible in "Design" classes, we found that the best way is to ask students, that as a part of their projects and related to them class material, they will write some relatively short but interesting programs. The students should have a ready design environment for it, such as complete sets of functions manipulating Binary Decision Diagrams (BDDs) (for solving some combinational problems like Boolean equations or minimization of logic functions), in order to not repeat all the hard programming work from scratch.

We tried various University BDD packages and tools, but we became quite disappointed, because interfacing to their internals was a major problem and it takes few months, for instance, to learn in sufficient detail the data structures of MIS tools from U.C. Berkeley.

Then, we thought, let us use commercial prototyping systems. We used in the past Lisp, Prolog, Mathematica, Matlab, and Pascal for fast prototyping, and **our choice after all these experiences is clearly Lisp.** Other languages were either too difficult to learn and master in one quarter (Mathematica is really difficult to write programs in it), or were too narrow for robotics applications.

1.3.1 Software Prototyping, Functional and Object-Oriented Programming. Lisp, Java and WWW.

Lisp is the major language of Artificial Intelligence, well developed and supported (especially by Franz and Gold Hill companies), and also with many shareware and public domain compilers and development systems available on WWW. A student who understands the general mathematical concept of a function, is able to write (in one week!) quite interesting Boolean and Multiple-Valued logic manipulating programs assigned to them at the end of the first meeting of the class. This two-hour meeting introduces Lisp fundamentals together with Boolean representation in Lisp and the concepts of rule-based logic minimization. In contrast, a student who took few quarters of classes in C++ is often not able to write a working problem for this task in one week, because, for instance, of his troubles with memory management functions of C++ (see problems 2.5.8).

There is much you can do in Lisp with a very limited knowledge of it, which is good for impatient students who want to quickly achieve some useful and stimulating results. Many Universities teach Lisp and Scheme (a clean subset of Lisp) as the first language. Lisp is also used in AutoCAD and in many commercial products related to advanced CAD, learning, search, user interfaces, editors, graphic tools (Interleaf), and data bases. As far as we know, all commercial logic synthesis software has been written in only three languages: C, C++ and Lisp. Internal Tektronix tools and their internal hardware description language AMBER are only few examples of using Lisp in CAD. Many systems were first prototyped in Lisp before being rewritten to C++ (the only other similar example is Smalltalk, but it is more difficult to teach).

Moreover, it is our opinion that Lisp language, in contrast to many languages that appeared and practically died in industrial applications (Prolog, Ada, Pascal, WildLife to mention just few) will survive, especially because of the recent "Java revolution". Java seems to be an improved Lisp, but it still does not have all of Lisp's nice features that make it so great in all prototyping environments, especially those related to functions, hardware, graphics, and interactivity.

If Lisp is that great, why so few people use it?

Firstly, people hate to count parentheses, being not aware that this is no longer a problem in modern Lisp development environments. Next, Lisp is perceived to be slow, which is absolutely not true for compiled Lisp programs. Even interpreted Lisp is good enough for learning algorithmic and data structure concepts. When your prototyped program works, you compile it with Lisp compiler. If you want to further speed it up its execution, you compile your Lisp program into C, optimize in C and add some fast C functions that do not exist in Lisp, and next you use your favourite C compiler and link to your other C/C++ programs, There are several Lisp-to-C compilers available in public domain on WWW. Lisp thus becomes a part of your entire C/C++ based development.

Java has learned a lot from Lisp, probably via Guy Steele and other researchers at Sun who contributed first much to Common Lisp and Scheme. Java is a **Very Good Thing** for the Lisp community: by popularizing the features that supposedly make Lisp slow, it destroys one of the primary (though bogus) arguments against Lisp.

Lisp has its strong believers and enthusiasts, and Lisp truly diserves it. On the other hand, some of them bash Java which is not reasonable, because Java is close to Lisp in C clothing. Therefore, if Java will progress as well as it has in the last few years, it will "eat" Lisp in one way or another, and will be doing even better. It does not matter if it will be called Lisp or Java, it does not matter what will be the syntax, the ideas of Lisp will be popularized and widely used. Development of Java is the best news for Lisp enthusiasts in the last 30 years. It is finally thanks to Java that the great early 1960's ideas of McCarthy, find after 30 years the access to a really wide public. Finally now the computers are powerful enough, the software technology is mature enough, and people learned enough programming already, to make these powerful algorithmic and representational ideas widely acceptable.

The first part of the book presents the *"Visual Hardware Lisp"*, VHL for short, the Lisp variant that we developed for designing complex models of logic circuits. At various stages of our work and for different projects, we used U. Texas Lisp, shareware XLISP, Symbolics Lisp, Gold Hill Common Lisp, and Allegro Common Lisp; traces of these languages you can find in our Lisp. There may be still some inconsistencies and errors in the provided examples of VHL programs, so the reader is kindly asked to report all bugs or problems to the author.

Learning Lisp will have several advantages for you as a student in "Capstone Project" class.

1. You will be able to quickly prototype software which will embrace your own new ideas, without going through standard pains of software development. You will be able to reuse much of the existing VHL environment.

2. You will be able to simulate, verify, and generate tests, in a comrehensive software-hardware environment that you fully understand, and you can control, in contrast to commercial tools that you spend much time to learn, and you cannot modify.

3. You will learn how to design hardware computers that interpret subsets of VH Lisp, so-called Hardware Virtual Machines of languages. This will be examples of designing *special-purpose general computers*, a very useful methodology for another projects, for instance in DSP or image processing.

4. You will learn how to design special circuits from Lisp specifications, which means, using Lisp as a very-high-level hardware description language. for instance, how to design a circuit to calculate a factorial from a specification like this: $factorial(n) = n * factorial(n-1)$, $factorial(1) = 1$.

All these things will be taught through the whole book, so the first few chapters about Lisp are only the first introduction to VHL, and the next chapters will give you more details, whenever necessary for your experimentation and assigned project. You are welcome to solve the problems, even not assigned formally, and bring me (or better email) your solutions, VHL programs, data and results (*tar* format is OK).

1.4 How to Perform Research and Work on Class Projects

Learning simple design processes and programming, you will be ready to work on a larger research project. However, do not start with the project until "you will know everything". Start as early as possible, and tie all short exercises to the big project that you have to turn in to your professor in the end of the third quarter.

This project will involve a lot of research. Is it realistic to teach people how to do research?

Although we know that it is impossible to teach people how to do research (even assuming that we know ourseves ;-)), We think, from our own experiences with both Capstone Projects and Graduate classes, that the following remarks might be useful for beginners.

The first author wrote most of these remarks several years ago and posted on his door for his students. You see, his door are just opposite to the entrance of the building which used to be a Portland Water Bureau, so many people came and ask questions like this: "Hi, can you explain me how, you guys, are calculating my sewer rates?". So, he explained them that he is sorry but this is not a Water Bureau anymore, but in the meantime the visitors look at his "How To Do Research Page" and asked for a xerocopy. Actually one older farmer came back after few weeks and told that he liked it. Not mentioning many, many students, who wrote that these remarks were actually very useful.

So, let us start.

Although the given below remarks seem obvious, We found that several of the students in the past didn't follow them assuming that "they know better", They often wanted to find shortcuts and skip some of the points.

Do not do this, please.

It will ultimately take you more time to achieve quality results, because you will have to iterate several times through the procedure. If you have never done research, we advise you to follow these remarks literally. You can relax on them after writing your first professional report on a piece of original work completed.

We hear sometimes, "but I am not a researcher, I just want to learn" - but the belief of this book is that even if you only want to learn, you will have more fun, you will learn better, and you will remember what you learned forever, if you will do this as a researcher.

And here is my recommended procedure:

FIND-1 *After reading the book for the first time, you should select one of the projects suggested in it, or ask your professor for a project. May be, you will have a project idea on your own? If it is a formal Capstone project, you have to find a local company to sponsor your project. I believe that working on one sizeable project you will learn more than trying to solve all homework problems, and I would encourage you to form your research group and start the project in the first week of the class sequence, usually in Fall.*

The book will assign you a relatively narrow and well-defined subject, for instance "how to combine a talking bear toy with Eliza-like semi-intelligent English conversation program?" Now you even do not know what does it mean, but do not worry. This will be some quite complex but narrow topic. Because it is narrow, you will be able to go into depth. The amount of literature on such subject is very limited, so you will be able to read them.

In addition, there are many more literature positions somehow related, which you can find using computer program Netscape or Explorer. For instance, using ALTA VISTA or other search engine under under Netscape, write the following into the search window:

+"ELIZA" +"talkingtoys" or similar.

This way, you will find most if not all recent literature available on your narrow subject.

Not **around** *your subject, but* **on** *your subject. Use at first ALTA VISTA and other search engines, next University library, good bookstore such as Portland's Powell's Bookstore, everything you can.*

Use Citation Index, IEEE Index, and other. The more you will use search engines, the more you will learn how to search for information that you need. This will be the first, most trivial, use of computers to do your research. You will get the feeling of power that that much knowledge from top world researchers and the largest libraries in the world is directly available to be used in your work.

Use literature from the papers that you will find, you can start with the literature lists from this book, but remember, this is only a starting point and all books become quickly obsolete. Try to find rare reports, such as M.S. and Ph.D. theses from "Michigan repository". Do you know that every Ph.D. and most of the M.S. Theses ever written in USA can be obtained from the Michigan Repository of Theses? Ask in University Library how to get it! You can also find it on WWW. Do you know that you can find also every American patent, read its abstract on line, and next order the entire patent for just $3.00? Patents are a great source of information. Look to the accompanying CDROM for many useful WWW sites related to projects in logic design.

Looking for literature on **"exactly"** *the subject, you will find as a* **"by-product"** *much literature that is* "around" *your subject. This is good, keep it and read it, but do not dwelve too deeply into it. Stay focused!*

DO NOT BE DISTRACTED BY TOO MUCH OF INFORMATION.

STAY FOCUSED AND UNDERSTAND CLEARLY WHAT IS YOUR GOAL.

IF YOU GET CONFUSED, TALK TO YOUR PROFESSOR.

ORGANIZE *Make your own WWW Page and make links to all interesting papers and pages that you will find. Print the most important papers or, make xero-copies of the most important papers from library. Bind them in binders. Organize these binders, write tables of contents. Use your WWW Page to organize them.*

This is now the second use of computer in your research, again, a rather simple one.

Skip through the papers superficially. Know what is where. Read the "around" papers only to understand their ideas, not details. Read the "directly related" papers in all possible detail. Understand their assumptions, definitions, goals, theorems and proof methods, applications, algorithms. Read, if necessary, the literature referenced in them.

READ **Read the papers!!! Read the papers!!!! Read the papers!!!!!!!** *Underline with yellow, blue and other transparent markers lines of text that you find important. You have to understand the papers. I mean, ideas, not necessarily all small details. Learn how to find about ideas, and not about formalisms.*

Many students are sensitive to formulas, but seem to not understand the concepts that are behind them. Concepts are more important.

Learning how to see concepts in what you read is one of the most important things you are supposed to learn in your University education. Memorizing formulas is useless. You will have to USA them to practical problems.

Maybe, the paper you found has no any real idea, and the author covers this behind his erudition and skills in formalization. Such paper is usually useless, and you have to figure this out and do not waste your time on reading it. The skill of reading papers in this way is not easy and comes with time. But having common sense and staying focused on your project is always good.

When you read, be very critical. *Do not trust the author, try to find his weak points and inconsistencies. Be able to separate and distinguish the components of the paper: the organization, the knowledge, the examples, the way of evaluating ideas.*

The paper can be good in one respect and poor in the other.

MODIFY *Modify your topic, if you think that a narrower or slightly different topic it will improve the quality of your research, or make your work more practically useful. Consult the change with your professor, and/or your more experienced older colegues who work on a thesis. Know what are their specialties. Talk to them.*

SOFTWARE *Is there any software, related to your research, available for free or little charge, that you can use? You can use such programs to learn more about the topic. You can also incorporate them, legally, into your program (giving credit) if they are public domain programs.*

For instance, there is a program from from University of California in Berkeley, called Espresso, that minimizes the logic of the circuit, allowing thus a more compact and thus better solution (we will learn about such programs in this book). This program can have many other important applications in robotics such as controlling the movements of the robot. You can use this program for minimizing your design, but you may also take the source code of the program and modify it as you wish. Observe that this free program gives you more possibilities than using a very expensive commercial EDA program from a company such as ORCAD, Mentor or Synopsys. Using somebody's elses program for your design in the next skill that you will have to learn to do research, and modifying it for your own application is another useful skill that you need. Most EDA software is in languages C and C++, but we will not teach these languages here. There is many excellent textbooks of them.

To acquire source code or object code of these programs, use WWW to find addresses of developers, write to them asking for this software, but most often you can download it from the WWW, sometimes you will get the software in email, or, you will download it using FTP. Again, read your Netscape manual to learn about using necessary tools, which is being made very easy to use with recent WWW programs.

Remember that on the WWW Page,

http://www.ee.pdx.edu/ mperkows/

there are lots of EDA software for these projects available and you do not have to start your software search from scratch.
Using somebody's elses source code is a great way to learn programming. Is it in Lisp, C, C++, Java, Prolog? Avoid very rare and immature languages. Use your previous programming experience. In this book we will learn Lisp as a method of fast idea prototyping, and you will find plenty of Lisp source codes on the first author's WWW Page and on WWW in general.

RARE *Are the any rare reports and internal memoranda of the top groups in the field? Write to them, ask for them. The authors have learned a lot from CMU, Berkeley, Stanford, Columbia and MIT reports. More than from conference papers. Again WWW is the great gate to all this knowledge.*

PEOPLE *Who are the top groups in the world in this area? Usually you can find them on WWW. Is there a famous researcher in this area with whom you would like to talk? Like all those famous authors from the books that you read on computer and logic design? We observed that these top researchers are very friendly and eager to help, if you will show your knowledge and enthusiasm. But they have no time. So if you write to them, ask them only one or few questions, ask for advice, that can be formulated in not too many lines. Write them a letter, about what you are doing, what are your goals, ask questions. First consult with your professor or one of Ph.D. students in your department. It is also a good idea to talk to Ph.D. students from top departments.*

You can ask people to send you their software or hard to get reports, but you do not ask them to send you widely available papers that you can find by libraries. Do not bother them too much before first consulting your professor.

FOCUS *Now, the first part of research, "data acquisition" is over. At this point you should know what is the "state of the art in the world" in this area. Unless you do not know what is the current research, there is a high chance that your research will be useless. On the other hand, the feeling that you know the "state of the art" gives you a lot of self-confidence when you start your second phase of research - the real research. Again, at this stage you should consult somebody much more knowledgeable than you.*

THINK *Think "what is weak in the existing approaches?" Are they solving real problems? Is this problem interesting to you? Are they too theoretical to be accepted in industry? Does the industry really needs this? Are they too low-level, too down-to-earth to publish a paper in a top IEEE journal? Are the programs from those papers too*

slow to be practically acceptable? Still they can have some good ideas from which you can start. Are the problems formulated well from the practical point of view? Can they be re-formulated? generalized? Generalizing concepts, definitions, operations, theorems is a very easy way to start research. For instance from the 2-valued or binary logic that is used in standard logic and in most current computers to a logic with many values, called multi-valued logic. However, is such a generalization of any use? Will it help my robot to be more efficient, smarter, cheaper? Can several ideas from other authors be combined together? What is good and what was bad in each of them? How do some old papers from 1967 look in the light of recent research? For instance, availability of very fast approximate multi-value minimizers opens again a lot of research areas from the sixties. New technologies make some papers obsolete but resurrect interest in other older papers and ideas.

Think permanently about your problem. Don't be afraid to be obsessed with it. It is normal. ;-)

It is better to start your research from any idea, however small it may seem to you, and keep improving on it, than to waste time looping around and being too cowardly to start your own research.

CREATIVE *Be creative when you think.*

`THINK!!! THINK!!! THINK!!! THINK!!!`

WRITE *Try to write about your ideas. Keeping your "research diary" and next laboratory log are great and very practical ideas. Note all your ideas, both small and great, reasonable and crazy. They will become very useful some day, and the very process of writing your diary will teach you a lot. Write the mentioned above critics of the research of others. Be critical about your writing. Each your next section will better than previous.*

SPACE *Try to organize your knowledge about the topic and the "surrounding research areas" in some form of tables, arrays, graphs, data bases. Especially, use the modern software tools such as organizers, data bases, outliners, spreadsheets, and WWW tools. Organizing data in structures such as two-dimensional tables, trees or other structures is called* **morphological method of research** *and we will learn more about this method in the book. For instance the rows of the table may be circuit technologies, and columns may be design methods. On the intersection of column "Method 1" and row "Circuit type 3" you enter all papers and ideas of using Method 1 to design Circuit of type 3. An empty cell at the intersection may mean that you have not done a good search, or - that some new research topic is possible? Is it worthy of research? What may be the use of it? So - keep asking yourself these questions: "What are the empty spots in the tables?" "Why is this cell empty?" Organize tables like this in various formats, with various meanings of row and tables. This way, you are systematically analyzing the entire space of your problem.*

REPRESENT *What is a* **good representation** *to think about your topic? Remember, you always think in terms of some representation. The more representations you know, the more flexible you are, the more creative you are. Some interesting things can be discovered just by translating from one representation to another one, for instance you can do more with arabic than roman numbers, just by inventing positional number notation, a great contribution to inventing efficient calculations was done. In this book, you will learn many representations of logic functions, which are some functions used to design digital circuits. They are also called Boolean, to honor George Boole who first found the way to create an algebra of human logic reasoning. Some of these representations will be called Karnaugh maps, to honor Maurice Karnaugh who invented them, some other are called "arrays of cubes". Another graphic representation are the Binary Decision Diagrams (BDDs), which represent a Boolean function in a form of a graph. Visualizing logic function in one of these representations you can learn how to manipulate it and how to see some of its properties useful to optimize them.*

This research phase should be considered a very important one. Try to formulate definitions, concepts, theorems. Start from simple facts. Generalize them. Observe facts in your representations or by using your fast program prototypes. Don't be afraid to use a lot of paper at this stage of research. We have to investigate many variants. Throw away all pages with bad ideas. Make order with ideas that proved to be useful.

FUNDAMENT *At this point you should have a few ideas of your own. If yes, try to organize them, rethink them, re-check them in literature (may-be somebody has done it already? - you do not want to be second, you want to be first). Now write a report presenting your ideas. At first, do not care about English or details. Try to write all your ideas as quickly as possible. You may want to dictate them to a tape-recorder first. Organize them as the draft of your paper Now, word processing and text formatting software becomes very useful. If you can afford, purchase voice recognition software and just talk to computer about your great ideas ;-).*

Go to [DRAFT].

PLAN-1 *If you still have no ideas of your own, there must be something wrong with you ;-(.*

Don't panic, but try calmly to analyse yourself. You are a University student, so you proved that you are smart, hardworking and motivated to be there.

Your SAT is above, well, you know how much.....

But may be never, until now, your creativity has been really challenged by your high school professors? Well, you have now a chance.

Many students learn to pass exams, or even to learn procedures, but they never attempt to create new things. Again, **don't panic,** *you will be able to become creative, but you must analyse, why you didn't come up with new ideas. May-be you were not reading detailly enough? May-be you are not focused? May-be you are not critical enough? BE MORE CRITICAL. DEVELOP YOUR OWN THINKING. Don't trust the authors of the papers, try to disprove some of their basic assumptions, concepts, and methods; and see what will come out. May be nothing, but perhaps something partially useful. Review, what is your goal? What is a plan to achieve it? Go to [FIND-1]*

DRAFT *Show your draft paper to your professor or a friend. Make a class presentation. This helps very much. Discuss it with a group of your friends after class presentation. Expose yourself to some brainstorming sessions with those friends whom you consider creative. Encourage them to criticize your ideas. Do not be afraid of their criticizm. It is only good for you. It is better to be criticized now than later by the reviewers, if you ever try to submit you paper to a journal or conference. Also remember, you do not necessarily have to come out with a totlally new idea to write a good and useful paper. Sometimes the idea can be for instance only how to combine in a new way few ideas that already exist. But, the less original your idea, the more important it is how good artisan you are, how professionally you can use your skills and knowledge.*

VISUALIZE *Make graphs, figures, tables. Make the entire plan of your papers, how it will look like when the computer data will become available. What kind of data (tables, benchmarks) will be added to your paper? where ?*

BENCHMARKS *Find from WWW or IEEE organizations what are the official benchmark examples for your area? There are some for Boolean functions, state machines, high level synthesis, filter designs, microprocessors. Download them. Use them.*

PLAY-1 *Below, I assume that you use Lisp for programming. Play first with Lisp on book's exercises and WWW available simple examples, related to your project. Test Lisp functions on your data, do some simple graphics and interface for fun. Whatever you do, think about your final goal. Stay focused.*

PLAN-2 *Before writing your program, plan carefully. The most usual mistake done by students is to write a program first and to do the thinking about its structure and documentation next. I would suggest you to write the documentation and user manual first, before writing the program. Looks crazy? But it is the way how some top companies recommend their programmers to work.*

With a word processor, when the program will be ready, the modification of the respective files will be trivial. But writing it early will help you to structurize, modularize, and organize your program.

Most of young researchers are too impatient, so they often write programs too early. Soon, they know that it was a great mistake. They just waste a lot of their time. What they learn next in industry, is, that when you write a really large program, if you don't plan and document, you can be in a real trouble of being not able to manage the complexity and lack of organization of your program, when it will reach 3000-4000 lines of code. Even, if you will work very hard. You have to spend a lot of time for planning, before writing the code, **believe me this is not a wasted time.** *You have not appreciated these problems till now, writing small programs (less than 1000 lines), to which you are used. But for large programs that we write in this class (more than 2000 lines, usually) this is a must.* **Work not hard but smart.** *If you start bad, it will be too late.*

PLAY-2 *Play with your program. Analyse results of your program, is it correct? does it have full functionality? what is slow? Rewrite main loops, compare your results with the results of your competitors - authors of previous programs of this (or similar) type. You shall know in advance, who they are, what are they data, times, week points of their programs. If necessary, rewrite parts of your program.*

Do not worry too much about speed of processing, at first. Get a full functionality and use the program to better understand your problem.

DOCUMENT *Write complete technical report, using pieces of all previously developed texts. But first, plan its structure, as we discussed above.*

REPORT *Write the final version of the complete program documentation, the user manual, and the developer's manual. Provide appendices with examples, printouts, script files. How to call your program, what must be in the directory. All information that will be used to restore your program in future. Makefile, etc.*

MODIFY *Modify your technical report again, now taking into account the program documentation. Include all details in appendices. Include complete literature.*

PAPER *Write your final conference paper. Plan ahead, remember about deadline. Much of it can be written by paste and cut method from the report.*

SLIDES *Prepare good slides/transparences for conference (we write about it elsewhere). Make class presentation of seminar presentation (plan in advance with me).*

At each of these stages, for instance when thinking about your slides, you will get new ideas how to improve your work, or at least its presentation. Do this, or at least make detailed notes for future use.

CONFERENCE *Hopefully, your paper is accepted to the conference of your choice. Hopefully, you have money to attend this conference. Ask your Dean, most Universities have special money for students presenting papers at conferences. During conference try to have as much feedback about your paper as possible. If a paper is not accepted, read carefully remarks of the reviewers. Apply them.* **Go to** *one of the previous steps, depending on what in your paper was criticized, and repeat the above procedure.*

JOURNAL *After conference write immediately a journal paper. You have now the complete program with documentation, the research report, the conference report, and, hopefully, the journal paper. This will look good on your resume. But what is most important, you have grown as a person. You learned how to solve new kind of problems. You also got a lot of confidence in yourself. This will be helpful during interviews. Also important, you know now that you* **can** *do research. You will be able from now to use this knowledge. Send us a letter with your story, we will really appreciate it.*

DEMO *You will be soon very popular in your community if you will organize a demo of your robot in some technical fair or conference. Think how you can impress your audience by your robot doing something useful.*

As you see, the above procedure either leads to a successful research and papers, or is looping. Therefore, it is not an algorithm in the sense of theory of algorithms :-) (you will learn these in the book). But it is just fine, you can do your research forever, if you have fun. You can get an "incomplete" grade from your professor, I hope, we gave them to our students. In the worst case, you can always write a tutorial or an overview paper. Or implement somebody's else's theoretical ideas (giving him/her all credit!) in your program. This may be very useful, if your choice was right.

Of course everything above is an idealized version, but why not give a try? Definitely, writing journal papers will be not required by your professor. However, writing reports and conference papers is often required and graded in graduate classes. If you are a Ph.D. student, remember that having a journal paper is one of the graduation requirements in some departments, and rightly so!

Write the authors an email, communicate your ideas about organization of research to me. We will take them into account while working on the next edition of this book. You will help your future colegues in research-oriented classes.

And remember, you can always write us a letter with your doubts and ideas.

1.4.1 Why may some students be not successful in research?

One more important aspect is the student's personality. We already supervised many, many Capstone projects and graduate theses (yes, yes, we are that old), so let us share with you our experiences. Based on our past experiences, we are absolutely convinced that in most cases the reasons of all student's troubles with research projects are of psychological, not intellectual nature. We will try to present you here few typical cases of our former students who were, at first, not successful, but ultimately were quite successful, they even wrote conference or journal papers.

"STUDENT A".

He is obsessed with an idea of being a great scientist, doing perfect things, writing a great thesis. He reads hundreds of papers in all possible areas. He is full of ideas of how he will work "in the future". He writes small toy programs, is very eloquent, discusses everything with everybody, criticizes research of others. However, time goes on and on, but he does not start to do anything practical and focused. His real problem is that he is afraid of failure, of investing his time to work on any particular topic. In truth, he is not self-confident enough.

"STUDENT B".

Graduate student B is a sensitive person. His paper was not accepted to a major conference, although he spend considerable time working on it. So he changes the topic of his thesis and he complains to his advisor, and about his advisor, who gave him this terrible topic. His next paper is accepted, so he becomes happy, he becomes self-confident. But he wasted his time on changing the topic. His first topic was not worse than the second one, and he should keep continue working on it.

"STUDENT C".

Student C does everything right, but is very slow in his progress. The professor does not push him to be faster, because everybody has to work on his own pace. However, if you would observe his day, he wastes a lot of time on many side subjects, completely unrelated to his research. He talks a lot on the phone.

"STUDENT D". Capstone Project student D is a mixture of Student A and Student C. He finds himself many things to do, and is very laborious. But, if you analyse what kind of problems he solves, this is everything to postpone the work on the core of the thesis. All his pretty-printing programs, pet machines to solve games, etc. are very nice and give him recognition among his peers, but do not push forward his thesis. He is not in hurry, he has good time solving little pieces.

"STUDENT E".

He does everything OK, but has moments of depression and very low self-esteem. His supervisor has to encourage him permanently.

"STUDENT F".

He is very much afraid of failure. So within his research area he selects topics that have no special meaning but can be done. He is critical and laborious, so adds himself more and more more of these problems, but it does not help much the quality of his thesis. He is smart, and he should have just more trust in himself. He should select more ambitious topic. This would lead to better results, and achieved with a smaller effort.

"STUDENT G".

He is extremely critical. As you remember, I recommended to be critical, when you read papers. He is, however, too critical. Therefore, he is not able to find a research topic. When he finds one, he withdraws from it with the first difficulties that arrive. He is extremely talented. He knows a lot of things about many subjects, but he never goes deep enough into a subject. Maybe he is not just interested enough? Maybe he thinks too much of his success and is not free in his spirit to start a real adventure of research? He is a really hard case for his advisor. He has to be permanently pushed by his advisor to have self-confidence. Sometimes he is pushed with success. He is a really hard case, but not helpless.

"STUDENT H".

He is in a hurry. In such a hurry that he wants to complete the project as quickly as possible. He does not read papers, he impatiently writes his computer program, in a very disorganized way. The program is slow and/or gives results of poor quality, so he is asked by his advisor to do improvements. The programs becomes more and more messy. The only solution is to drop it and start again. This would save him much time. But he is in a hurry. So he tries to modify the program. Ultimately he wastes at least half a year.

"STUDENT I".

The real problem of this student is that he is not interested in engineering. He has to find this out for himself. But he is a smart person, and we are able to find an interesting research topic for him, for instance on the frontier of biomedical applications, Artificial Intelligence, or logic.

Which of those students resembles you? Think, maybe it will help. Definitely many of them resemble one of us.

Find a source of strong will in yourself, do not be afraid to take risks, this is the best time in your life to be free to take risks.

And, **"JUST DO IT!!!".** This slogan works.

1.4.2 And what previous students think about these research project ideas after completing them?

We will conclude this subsection with few remarks of some of the former students, often students who "invented something new" in the classes. We often ask them the traditional question *"how have you discovered this?"* (We keep their original English, and we faithfully repeat without making selections).

- ["FRESH ALUMNI 1"]

  " When I know what is the problem (how to test systolic processors) I just keep thinking about it all time, day and night. One night I woke up and the whole idea of research was ready in my head".

- ["FRESH ALUMNI 2"]

  " When you told me that it can be done, that even I as an undergraduate student I can do research, I was just thinking permanently how to solve the problem that you gave me".

- ["FRESH ALUMNI 3"]

  "I just wanted to write a program better than anybody's else. But to do it I had to learn what others have done. I found that many things can be improved".

- ["GRAD STUDENT 1"]

  "I think that the idea is: "invent something new, next try to improve it". That's it."

- ["CAPSTONE PROJECT STUDENT"]

  "I think that reading a lot is most important. When I have detailly red the entire literature on the subject, I know that I can also contribute something".

- ["A LITTLE BIT OLDER PHD STUDENT"]

  "I ought my PHD to your wife. One day I was invited to your house and she gave some terrible spicy food. I couldn't sleep the whole night and I invented the most important ideas of my PhD".

- ["STUDENT 3"]

  "Oh, I just did, what you asked me for".

- ["ANOTHER STUDENT"]

  "I was just playing a lot with Karnaugh maps. I invented something and next I formalized it".

- ["GRAD STUDENT 4"]

  "I wanted to do something useful. I felt that what you teach in class is too theoretical, and we had no tool like that in our company either. I wrote a program and I experimented with it".

- ["UNDERGRAD STUDENT 5"]

  "I just thought that I can use approach of prof. X to problem Y as well".

- ["STUDENT 6"]

  "This just seemed as the best method for this task".

- ["ALUMNI STUDENT 7"]

  "I think that reading much is most important".

- ["FRESH GRADUATE"]

  "I was so obsessed with the final topic on which we agreed that I was thinking about it permanently. Different systolic architectures were before my eyes day and night. I spend also a lot of time reading about this, but I realized that reading more is waisting of my time, so I concentrated only on thinking".

- ["ONE MORE STUDENT"]

  "Dr. Perkowski, you really didn't realize, how much time I think about your project. I think about it when I eat, when I work, when I sleep. I draw pictures on paper and in my imagination".

- ["YET ONE MORE STUDENT"]

  "Your guide "how to do research" really worked for me. I followed it and I learned much by trusting it. But now I can understand what you mean, before I thought that you exaggerate". But your enthusiasm helped me to start the project.

- ["FRESH GRAD"]

  "I just concerned on my topic, concentrate on my interest and let the rest flow freely."

- ["ONE MORE FRESH ALUMNI"]

  "I do not know, just doing everything that must be done as required".

- ["RECENT PHD"]

  "Most important is to read a lot. When you read 100 papers in your narrow area, you know more than anybody else, including your advisor. This gives you tremendous self-confidence during conferences, otherwise you would be scared to talk to all those famous people. You find that you have many ideas and that they appreciate your ideas".

- ["ONE WHO JUST FINISHED"]

  "I just like crazy ideas. For the fun of itself. I believe that good research comes from ideas that initially look crazy."

- ["ONE MORE"] "Thank you for being pesistent with me. Sorry it took so long."

Please write your own opinion about creativity, problem solving and research. Do not be afraid to disagree with us. There is a good chance that we will include your remarks to the future version and it will become useful for the next students.

`GOOD LUCK IN YOUR RESEARCH. HAVE FUN.`

1.5 What Else Will You Learn in this Book

We hope this book will teach in a non-standard way, easy to understand on one hand, but advanced.

Will you be able to achieve this?

If yes, how?

Instead of showing complete algorithms and their data structures, the book will only give you a "menu of dishes" but you will compose your own "dinner". The book will present you a space of methods, representations, algorithms, programs, circuits and ideas, from which you will create new ones and larger entities, such that they will be "inventions", not only "designs".

Like, when you learn how to combine Lego blocks, you can build everything that you want.

On the other hand, because the chapters will be devoted mostly to ideas, you will have to study problems and exercises for all details related to practical realizations. Thus, you will have to read this book systematically and solve problems, and not only to participate passively in class meetings and hear lectures. Finally, you have to work on one of large projects.

First we will present basic tools and next problems that we will be solving. We discuss combinatorial models to which thay can be reduced. Let us make a point, that it can be done in many ways.

Next we introduce abstract representations, next computer implementations and we investigate their various reductions and dependencies. Finally, we do the same for computer structures: algorithms, softare and hardware are similar ideas.

This book will attempt to present in an uniform and comprehensive way all knowledge necessary to designing new versions of logic algorithms, machines, and other innovative digital circuits and computer architectures.

We have already a very well defined **idea of this class**:

- Lisp and representation of combinational problems.
- Representation of Boolean, Multiple-Valued, Discrete, and Quantum logic functions, relations and machines.
- Search as a fundament of algorithms.
- Combinatorial problems (graph coloring, set covering, satisfiability, solving equations).
- Representations of Boolean and discrete functions; their uses to solve combinatorial problems (Binary Decision Diagrams, other Decision Diagrams).
- Finite state machines and their generalizations in robotics.
- New concepts of digital circuits and systems: design versus discovery.
- High-level algorithms in hardware.
- Adapting robotic and animatronic toys for mechanics, and home/industrial sensors.
- Experimenting with systems.

If you do not understand yet all these terms, do not panic, you will, with the end of reading this book. However, with students' feedback it will be our goal to convert this rough draft of a book to a more comprehensive and better didactically book shaped by the way how they think this material would be best transmitted to a wide audience. What I will write below is then our perception of this book, and may be, this perception is not yet reflected well enough in the partial chapters that you receive hereby. You will help me with your solutions to homeworks, your questions, and your figures submitted by email (xfig format only, plus postscript).

This is not a standard engineering textbook in robot design, nor it is a book on various logic-based robot design theories. Although we present several advanced issues in robotics this is not a monograph on these subjects as well. This book has, however, much to do with fast prototyping of complete robotics systems with use of FPGA technology.

The main goal of this books will be to teach you how to solve certain types of problems: mainly the combinatorial and sequential problems that require logic and can be reduced to logic problems. As we will see, problems of this type exist in the design of the logic of computers, but also in programming, Aritificial Intelligence, mathematics, Computer vision, Robotics, Operations Research, Graph Theory, and many areas of technology and practical life.

For instance, you are may be familiar with regular expressions as a user of Unix, but you did not realize that they can be used to design very well-minimized and easy verifiable circuits. The links between software and hardware are most easy to observe on the level of logic and state machines, as well as simple languages that serve for their formal specification, simulation, synthesis and documentation.

Heuristics, as a science or art of solving problems, cannot be taught in separation from some practical area from which examples could be taken, and utilized to illustrate certain *general purpose* problem-solving strategies,

such as search, rule-based behavior, problem decomposition, problem reduction, generalization, analogy or symmetry. As such illustrative domains, books on heuristics were historically using parts of mathematics (especially high-school geometry and algebra - for instance the famous book by Polya [708]), games and puzzles, rarely problems from business and computer programming. It seems to me that logic and especially machine design is an excellent area to develop, generalize and teach the science/art of heuristics. This fact was not recognized until now, perhaps because logic is not taught in high schools, nor it is much taught in universities outside few departments such as computer Science, Electrical Engineering and Mathematics. The formalisms of logic are difficult which often discourages its potential users. By writing a book on multi-valued and sequential logic that emphasizes many examples, we want to make this area closer and easier to comprehend. Remember always, that any function or machine can be build both in hardware or in software.

We believe that the material included here can be of interest also to anybody interested in solving logical problems: engineers of various areas, programmers, computer enthusiasts and even very advanced hobbyists. Although most of our problems are related to computer design, the reader will find many problems from other areas as well, such as image processing or solving puzzles. It is so, because it is easier to explain an idea using an easy example that is itself natural and taken from common life. In any case, the "toy examples" will be usually followed with real-life examples. The essence of our approach is, however, very general and can be applied everywhere.

In order to address the book to such a wide audience, we start from the very beginning, and virtually no initial knowledge of multi-valued logic is assumed by the reader. Step by step, however, we will guide him/her to the most advanced levels. So that, when completing reading this book, the reader will be able to formulate and solve the most complex problems, write programs for them in functional or logic programming language, and use the methods presented here to solve his/her problems posed by everyday life and techology. Moreover, he will be able to design complete robotic systems, hardware and software, that will instruct you totally innovative approaches to solving problems.

For the first time in history, the new techologies such as Field Programmable Logic Array (FPGAs) and based on them fast prototyping boards and emulators, allow to design robotic systems by individual interested people or small groups of people. The FPGA technology is at hand, let us use it in a really creative and innovative way.

We intend the projects for the class to be in the following areas:

- design new algorithms, theoretical concepts, software for VH Lisp environment. You can propose any project of your choice, and related to your experience. The projects that were already started, will be continued.
- design new circuits, systems, architectures using FPGAs, interfaces.
- design complete robotic systems based on new concepts (for instance, mimicking biology: Darwinian/Lamarckian evolution, immunologym, cellularity, gas dynamics, simulate annealing, ecological systems, etc.)

We have organized classes with this philosophy in minds for the 10 recent years. It worked! Students developed new software and hardware, published papers in top conferences and journals.

To be successful at the end, the reader has to develop several skills while reading this early draft of a book. Those skills include: designing circuits, writing prototype programs, posing and solving problems. In order to learn those skills there is no other way but **doing it**. There is no way you can learn those things by just reading the book, or any other book on similar topic that you can find or any other papers or notes that you received or will receive. You have to **actively participate**. In order to help the reader, I prepared here, at the end of each section, many problems to solve. Some of them are easy, but some other require a real thought - they are an integral part of these notes, and it is highly recommended that the reader will solve them by hand (or rather, head), and if necessary, also using a computer.

There are many Lisps in public domain. An inexpensive PC-compatible machine with public domain Lisp, or any other similar system is powerful enough to solve all the given here problems - a total expense of less than $ 700. Think about it. For under $ 700 you can have an entire system that is equivalent top research computer from 1980. Now you see that it is possible to take top "very complex" research projects from MIT or Stanford, such as MASCYMA, and run them on your lap-tops. Many such programs are avaiable free as "public-domain software".

Solving the problems given here prepares the user to read even the most advanced journal papers on logic synthesis and related areas. Such papers use difficult to follow formalisms to explain the ideas that are basically easy to comprehend. Therefore, good intuitive understanding of various logic problems, methods, circuits and algorithms presented in the main text and in the Problems_To_Solve sections, plus knowledge of general-purpose problem-posing and problem-solving strategies should help the reader not only to understand the research of others, but also to creatively approach their own problems and do some original contributions to logic research and applications.

The book devotes much attention to the representation of the problems. It is well-known that this aspect is extremely important while solving any problems, let us imagine for instance multiplication of two large numbers represented in Latin numeric systems, such as MCCIX * MCCDDDVI? Now compare how easy it is to multiply them while

represented in our standard way of representing numbers. Knowledge of good and popularly used representations together with tools for them, should simplify and speed-up the entire process of posing a problem, solving it, finding a method to solve this class of problems and next writing a program to solve problems of this class. By using of the existing functional and logic programming software, as well as using programming techniques shown, the user will have a tremendous time advantage.

By selecting materials from the very large body of knowledge to this book, we were quite biased. We selected them from the point of view of presenting the unified powers of problems formulating, reduction to logic problems, and using logic methods, such as Boolean equations or logic programming to solve them.

In this book, we want to emphasize the following:

- strategies of solving problems by reducing them to the well-known problems,
- reduction to logic problems, especially to logic equations,
- tree search methods for problem solving.
- hardware modeling and fast prototyping of new algorithms.

1.6 Logic, Logic Design, Programming, Problem-Solving, Formal Design and Verification - The Synergism

There is something intriguing and mysterious about logic. It is everywhere in our lives and our thinking, so much that we are even not able to see it. We use it in our every day reasonings and we do not know that we are using it. Logic is at the fundament of mathematics, physics and philosophy, but average person does not have to use this in the life. Logic describes what is possible, defines the space of solutions, decides what does not belong to it. Therefore logic can be used to solve many problems. Logic is however difficult to learn because of it highly abstract concepts and the necessity of understanding its different languages and meta-languages. We select thus only a small part of logic, but one that can be very easily illustrated with constructive and intuitive examples. What we teach here is "an engineering subset of logic", those topics that have or will have soon, practical applications. More advanced material on multi-valued logic which has no direct application to modern and forthcoming technologies has been not included, the same as more futuristic technologies of realizing multiple-valued logic in circuits.

The book has several goals and attempt to achieve them concurrently, believing that learning all of them together is easier and more pleasant than to do this separately. This is because they will be reinforcing one another. These goals are:

1. To teach functional programming (Lisp) and logic (Prolog in Lisp) programming in robotics environment.
2. To teach logic circuit for robotics.
3. To relate logic synthesis to robot operation.
4. To encourage use of commercial EDA tools from ORCAD, SUMMIT, XILINX and MENTOR companies in design, test, verification, simulation and documentation of your designs.
5. To teach software-hardware co-design.
6. To teach fast prototyping of FPGA architectures to solve problems.
7. To use all the above areas to teach the art of problem-solving by conciensious usage of heuristics.

We want to show the beauty and power of logic problem formulation, to give you flexible tools to solve many problems of different areas. On the other hand, many of those tools we next use to one large task: development of problem-oriented computers. We think that this **synergism** is very characteristic for the logic methods.

As you will find out in this class, logic as a theory is used to create simple logic designs and logic programming. Logic programming in turn is used to solve logic problems and design logic circuits. Similarly logic will be used to develop the logic computer, which, when ready, will be able to solve logic problems and execute logic programming programs in a more efficient way. We illustrate on this diagram and in the book only some of the possible relations. We hope that the reader will be able to find and creatively develop more of them.

This kind of mutual-reinforcement, mutual dependencies, usage of a formal system to "bootstrap itself" exists also in mathematics and physics. It exists to some extent in design automation of VLSI circuits, perhaps because it is so closely realed to logic. This "strange loop" mechanism was observed by Hofstadter, and he devoted the whole interesting book to it [685]. It is, perhaps, something more fundamental to it, that both [685] and our book will

only touch upon barely, and from different perspectives. We think, then, that this synergism, mutual reinforcement, dialectical dependency, will be reappearing in the future history of computing, and perhaps many more unexpected developments are awaiting us in the area of logic, being at the fundament of all knowledge. We want to make some small steps in the direction of better understanding of those relations. The Turing machine can simulate another logic machine, and universal Turing machine can simulate itself.

As mentioned above, logic describes all the possible worlds, creates the set of all solutions, but does not say how to find efficiently any single solution or how to find the best solution possible. For instance, in the case of geometry, all known and all geometry theorems yet to be derived are hidden in the set of axioms. You just use rules of logic reasoning (or an automatic theorem-proving program) and you can derive all those theorems. Can you really? You can, but it does not mean that you will, since the space of all derivations is infinite in some problems, and finite but astronomically large in other problems. You need, therefore, some other, additional mechanism to guide your search. This mechanism allows you to take only prospective pathes in the labirynth of all possible derivations. Your "guiding mechanism" can use some measures such as simplicity, interestingness, usefulness, cost, beauty, symmetry, analogy, to select some pathes and avoid the others. This is called *heuristics.* It can guide, it can be useful, but it can be not proven that it is useful always. Solving any kind of problems needs some kind of heuristics. Therefore heuristics is a part of any of the above methods from our space. Heuristics is used to solve logic problems, and logic gives ways to formalize and enhance heuristics. So, concluding, you need good problem formulation, good representation and good heuristic to be successful.

Why it is good to discuss heuristics with relation to logic? In logic the problems are cristally clear, not cluttered with the "noise" of real-life such as business problems, they are ideal models of the "hard" problems. We can therefore easily distinguish the logical and the heuristical part: we can show which part of the solution method is logical and which is heuristical. The only other area where this can be done is an axiomatic part of mathematics such as geometry, and this was shown in the best possible way by Poly'a and his students. Although their approach is excellent to learn and explain the heuristics and strategies of problem-solving themselves, the practical usefulness of the presented by them methods is limited because who really needs to solve geometry problems if he is alreay after SAT exams? However, still Polya's books are used in companies such as Microsoft to teach people of clear thinking, problem formulating and solving strategies.

While the developed by them methods can be *converted* to other areas, the methods developed here can be used *directly* to several areas of practical importance. Moreover, our methods can be used in more ways: as "hand methods", as methods to program the todays computers, as methods of designing special hardware for logic problems. Because one can learn a lot from Polya's books, which use mathematics as a vehicle, we try now to use his approach in a new domain of logic, design and programming.

The market for new robotic systems is unlimited and only our imagination is our limit. Sixteen-year-olds can be good programmers, and digital design should be not more difficult than software design with good learning and programming environments.

Just recently, the fuzzy logic had become useful in commercial products from Japan: it controls washing machines and focuses video cameras. There are tens if not hundreds of other logical systems whose importance is known to the logicians for years. Like fuzzy logic, for years they have been a domain of the theorists and rarely of the programmers, but not hardware developers. Implementing them in hardware will give tremendous speed advantages, which for the first time will make them truly useful practically.

We will learn about such systems in this book.

At the end, please find books that are influential, thought-provoking, and/or related to topics mentioned in this introduction.

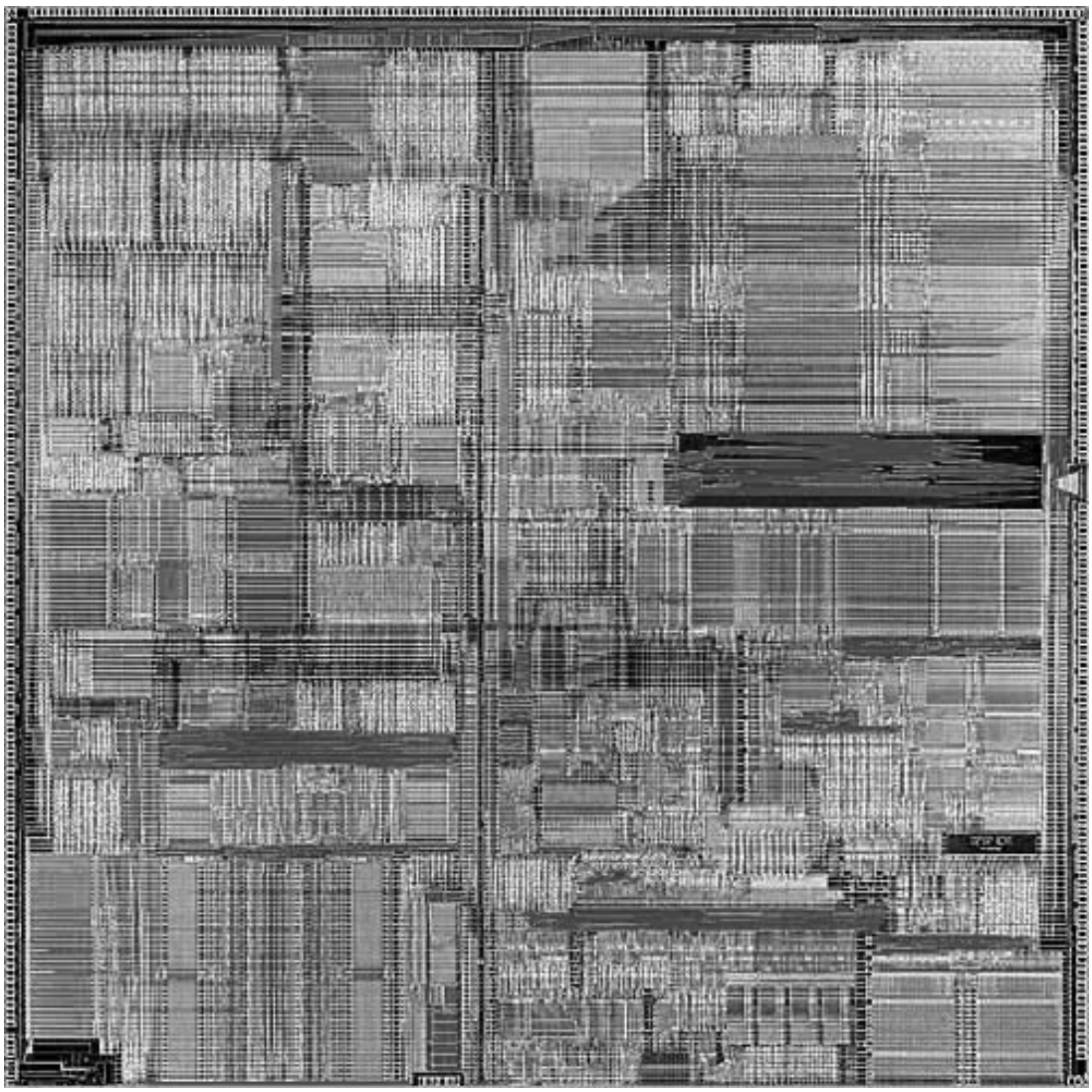


Figure 1.3: The inside view of a Micro Chip from Intel: you will learn how to design digital chips of this type (but not that large!). You will also learn how to write programs to design chips.

Part I

Lisp and Functional Programming

Chapter 2

Introduction to Functional Programming and Lisp.

2.1 Visual Hardware Lisp

This chapter is just a first try and it will be much modified. I appreciate your feedback. My goals were the following:

a. Students must have a good understanding of fundamentals to write complex design automation programs on their own, I do not want them to spend time on unimportant programming details but rather to concentrate on algorithmic and data structure ideas.

b. In our case, the fundamentals are: programming in Lisp, and implementation in Lisp of data structures and algorithms that are typical to digital design, especially search, combinatorial problems, format translations, and rule-based programs.

c. The best method of learning is by considering somebody's else examples and later doing similar examples on your own. Therefore, there will be a lot of problems to solve. Please, try to solve as many as possible. Some of the examples will be quite simple, but you will find also some that will be more complex and some that are quite difficult. The complexity of the problem will be evaluated with stars, the more stars, the more difficult the problem.

d. Good programming methodology is of a big importance. The students are referred to the literature on programming methodologies, and especially Lisp and AI programming methodologies for reference.

e. The extensive list of literature is given as the initial source of information for students who want to extend their knowledge. Basically every problem assigned in this chapter has been solved in at least one of the cited papers.

1. Capital and Small Letters. Capital letters are different than small letters. We will write both of them. Consult your Lisp which one is good for writing functions, other atoms does not matter.
2. Italics.
3. Comments.
MUCH Material will be added.

2.2 Your First Expressions in Lisp

A program in Lisp is a sequence of expressions. Each expression consists of left parenthesis, name of function, arguments of this function and right parenthesis. When the expression is typed to the computer, its value is calculated and returned (printed on screen). Suppose for instance, that you want to add two numbers. You type:

```
(+ 3 4)
```

Computer returns:

```
7
```

This will be denoted as:

```
(+ 3 4) ==> 7
```

What have we done?
We called function + with two arguments 3 and 4. The value of the expression (+3 4), is calculated and returned. It is equal 7. We will use the above notation in all examples. Symbol ==> will denote evaluation of the expression by the Lisp system. Symbol ==> is preceded by an expression to evaluate and followed by the value of this expression. Lisp can be then treated as an evaluator of expressions. These expressions are sometimes called functional expressions.

The expressions can have arbitrary number of arguments:

```
(* 2 3 5) ==> 30
```

Expressions can be arguments of other expressions :

```
(* (+ 4 2) 5) ==> 30
```

```
(+ (- 4 3) (* 3 2) (+ 3 4 5)) ==> 19
```

When the expression is typed into the system, its first element is treated as the name of a function. The arguments of this function are calculated from left to right (we say that arguments are evaluated). Finally the value of the function is evaluated and returned. For instance, when the expression (* (+ 4 2) 5) is typed in, first the value of the internal expression (+ 4 2) is calculated and result of this calculation, 6, is an argument in the expression (* 6 5).

Some functions do not evaluate some arguments or even all of them. Some other functions can have arbitrary number of arguments, while the others have the always specified number of arguments. We will discuss these problems in more detail, but first we must introduce some basic functions of Lisp.

Quote

Function QUOTE has one argument, which is an arbitrary expression.

```
(QUOTE apple)
```

means that the argument apple is not evaluated but its value is returned. We can also write `'apple`. This abbreviated form will be used in most of the following examples.

Function `QUOTE` is always used when we want the argument to be returned in the same form in which it is given to it. In other words function `QUOTE` does nothing. Why we need a function that does nothing? Most functions in Lisp evaluate its arguments, but sometimes we don't want them to be evaluated and we use `QUOTE` for this reason. Abbreviated form of QUOTE is '.

```
(QUOTE apple) ==> apple
'apple ==> apple
345 ==> 345
'345 ==> 345
(* 2 '3) ==> 6
(QUOTE (* 2 3)) ==> (* 2 3)
```

SETQ and SET. Assigning values

We know already how to calculate simple expressions. But what shall we do if we want to store the results for future use?. How make the computer to remember the expressions? This can be done with use of variables. You can imagine variables as certain memory location in which numbers, names and expressions are stored. Each such location has its name, we will refer to this name as to the variable's name or simply to the variable.

Function `SETQ` serves to assign values to variables.

```
(SETQ fruit  'apple) ==> apple
```

We assigned the value apple to the variable fruit.

```
(SETQ cost_of_fruit 30) ==> 30
```

We assigned the value 30 to the variable cost_of_fruit.

Function `SETQ` has two arguments. First of them is the name of the variable and the second is the value of the variable that is assigned to the first argument. The value of the first argument is not evaluated, but the second argument is. Therefore we have used `QUOTE` before `apple`. Numbers are evaluated to themselves, then 30 was not quoted. What would happen if we would forget to quote `apple`?

```
(SETQ fruit apple) ==> NOT BOUNDED VARIABLE apple
```

System responds that it does not understand what is `apple`. It treats apple as a variable and wants to evaluate its value. But no value was assigned previously to `apple`. Variable `apple` has then no value, or in other words is not bounded. The system points out this fact. Let's then try to assign a value to `apple`:

```
(SETQ apple 'MacIntosh) ==> MacIntosh
```

We call again the expression:

```
(SETQ fruit apple) ==> MacIntosh
```

Now everything is OK!

A value of apple, i.e. `MacIntosh` is assigned to variable `fruit`. We have then now two variables, `fruit` and `apple` that have the same value - `MacIntosh`. What can be the other values of variables, except of names and numbers? In the future we will learn about many possible types of values, let us now introduce just one of them - a list.

A simple list is a sequence of the left parenthesis, an arbitrary number of names and the right parenthesis:

```
(oranges lemons apples)
```

Of course, we cannot forget to quote the list, when assigning it as a value to a variable.

```
(SETQ fruits '(oranges lemons apples))
            ==> (oranges lemons apples)
```

Now the list `(oranges lemons apples)` is the value of the variable `fruits`. To check the value of any variable it is sufficient to type this variable:

```
fruits ->  (oranges lemons apples)
apple -> MacIntosh
fruit -> MacIntosh
cost_of_fruit -> UNBOUNDED VARIABLE cost_of_fruit
cost_of_fruit -> 30
```

Can we use a name assigned as a value or a name from a list as the variable? Sure, we can:

```
(SETQ oranges fruits) ->  (oranges lemons apples).
```

The value `(oranges lemons apples)` is assigned to `oranges` and also returned as the value of the expression. Calling the function with its arguments (*"evaluating the expression"* - in other words) has then two effects. One of them is returning a value, the other one - changing something permanently in the memory. This is true for most of the Lisp functions. We will refer to a value and a behavior (side effect) of calling a function.

Let us now consider a more complex example:

$$(\texttt{SETQ friends } (\texttt{SETQ neighbors } '(\texttt{Jane Mary Elizabeth}))) \tag{2.1}$$

By evaluating the above expression the name `SETQ` is interpreted as the name of the function. Its first argument, whose - according to the behavior of function `SETQ` - is not evaluated, is the name `friends`. The value of the second argument is the value of the expression

$$(\texttt{SETQ neighbors } '(\texttt{Jane Mary Elizabeth})) \tag{2.2}$$

By evaluating this expression the name `SETQ` is interpreted as the name of the function. Its first argument, whose value is not evaluated, is the name neighbors. The value of the second argument is the value of an expression

$$'(\texttt{Jane Mary Elizabeth}) \tag{2.3}$$

This value is `(Jane Mary Elizabeth)`. Now evaluation of an expression 2.2 follows, which results in assigning the value `(Jane Mary Elizabeth)` to variable `neighbors`. The same list is also returned as the value of the functional expression 2.2. Now the value of the functional expression 2.1 can be already evaluated. The entire evaluation

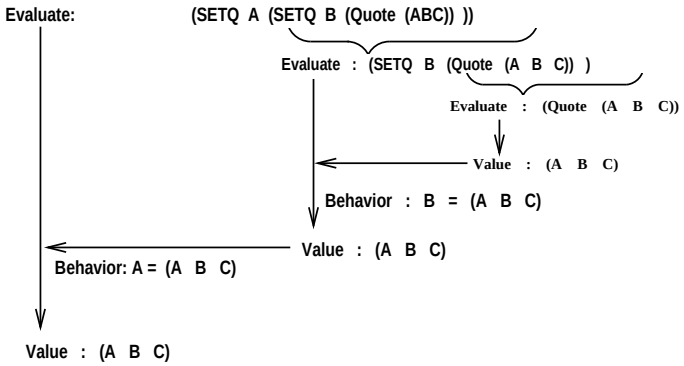


Figure 2.1: The process of evaluating functional expressions and assigning values to variables

results in assigning value `(Jane Mary Elizabeth)` to variable `friends` and the same is also returned as the value of expression 2.1.

The described above process of evaluating functional expressions and assigning values to variables can be presented graphically as in Fig. 2.1. To make it slightly more complex, we replaced variable `friends` with variable `A`, variable `neighbors` with variable `B`, and list `(Jane Mary Elizabeth)` with list `(A B C)`. This way, the same atoms `A` and `B` are used twice.

The reader is asked to verify few more examples:

```
(SETQ expression_1 '(+ 3 4 6 (* 2 7))) ==> '(+ 3 4 6 (* 2 7))
expression_1 ==> '(+ 3 4 6 (* 2 7)))
(SETQ result (+ 3 4 6 (* 2 7))) ==> 27
result ==> 27
```

We will now present a little bit strange, but quite didactic example:

```
(SETQ SETQ (QUOTE QUOTE)) ==> QUOTE
```

Let us imagine that you are a Lisp system. To evaluate the expression you first interprete the first element of the list as the name of the function. The function is `SETQ`. Then the second argument is not evaluated and it is the name of some variable, `SETQ`. The second argument must be evaluated. By evaluating the expression `(QUOTE QUOTE)` the first item is interpreted as a name of the function and the second item as its argument. As the result of evaluation, the value `QUOTE` is assigned to variable `SETQ`. `QUOTE` is also returned as the value of the expression.

This example illustrates that, depending on the context (place in the expression), the same name can be interpreted as the function's name, variable's name or the function's or variable's value.

There is one more function for assigning values to variables, that does not quote its first argument. This function, called `SET`, evaluates both its arguments:

```
(SET 'tool 'hammer) ==> hammer
```

As the result of evaluation, the value of the second argument is assigned to the value of the first argument and returned as the value of the expression.

As we see, expression `(SET 'a b)` is equivalent to `(SET (QUOTE a) b)` and also to `(SETQ a b)`. This explains also the name of the function `SETQ` as the composition of names `SET` and `QUOTE`. We rarely use `SET` as shown above, `SETQ` is better in such respect. `SET` is used in the cases when the first argument is a result of some evaluation.

```
(SET tool 'on_the_shelf) ==> on_the_shelf
```

As the result of evaluation of the above expression name `on_the_shelf` is assigned as the value of the variable being the result of evaluation of variable `tool`. Since the value of `tool` was `hammer`, then `on_the_shelf` is assigned as the value of `hammer`:

```
tool ==> hammer
hammer ==> on_the_shelf
```

As we see, functions `SETQ` and `SET` are similar. The only difference is in `SET` evaluating its first argument and `SETQ` not evaluating it. We can also say, that `SETQ` takes the name of variable as its first argument, while `SET` takes the value of the variable.

What are the other applications of SET Let us assume that depending on some conditions we want to assign the value (Jane Mary Elizabeth) either to variable friends or to variable enemies. Let function Who_my_neighbors_are returns as its value the first or the second argument, respectively to satisfaction of some conditions tested on variable neighbor (evaluation of gossips in this case). Evaluation of an expression:

```
(SET (Who_my_neighbors_are 'friends 'enemies neighbors) neighbors)
       ==> (Jane Mary Elizabeth)
```

will assign the list (Jane Mary Elizabeth) either to variable friends or to variable enemies.

How to write an interesting function Who_my_neighbors_are, that will help us in our troubled life? We must wait with an answer by first learning more about variables, lists and names.

Concluding, let us remember that:

```
(SETQ ATOM EXP)
PSEUDOFUNCTION, FSUBR.

(SET ATOM EXP)
PSEUDOFUNCTION, SUBR.
```

2.3 Symbolic Expressions

Both programs and data are written in Lisp in the form of the so called *symbolic expressions*. There are basically three types of symbolic expressions: *atoms, lists* and *dotted pairs*. The most basic symbolic expression is an atom. Atom is a sequence of *letters* (in some systems capital letters) and *digits*. It cannot include spaces, commas, dots and parentheses. There are two types of atoms: symbolic atoms and numbers. Let us discuss the symbolic atoms first.

2.3.1 Symbolic Atom

A *symbolic atom* is a sequence of letters (capitals) and digits, starting from a letter (capital). A symbolic atom can include not more than 30 characters (it depends of course on the Lisp implementation, but we can assume that atoms are long). It is forbidden in Lisp to use atoms that start from digits and are not numbers.

Examples of atoms:

```
A
Atom
atom
Mary
Elizabeth
J23
KGB
fruits
```

The following sequences are not atoms :

```
32df
5a
3,4
c(s.
```

An atom is treated in Lisp as the smallest distinguishable, nonseparable entity - component of symbolic expressions. The letters and digits from an atom are not treated separately, but as an entire item - regardless whether the atom is a single letter or a 30-character sequence. Hence, the atoms A, B, and AB have nothing more in common than the fact that they are all literal atoms. In this text we will assume that small and capital letters are equivalent. This property differs also from Lisp to Lisp.

Each expression is created from atoms, spaces, commas, dots and parentheses.

The dotted pair

The *"dotted pair"* is an expression which is a sequence of: the left parenthesis, the atom or another S-expression, the dot, the next atom or S-expression and the right parenthesis. Note, that the S-expression in a dotted pair can also be a dotted pair, and each S-expression in this dotted pair can be again a dotted pair, and so on. The S-expression on the deepest level must be however a dotted pair of atoms, since the expression must be finite. We say that such definition is *recursive*. We will learn much more about recursion in the coming chapters.

Examples of dotted pairs:

```
(a . b)
((Mary . has) . a_little_lamb)
((cost_of_beer . 30) . (cost_of_bread . 2))
((1 . 2) . ((3 . 4) . (5 . 6)))
```

An atomic dotted pair is a dotted pair of two atoms - i.e.

```
(firstAtom . secondAtom)
```

In such a case it is advised to surround the dot with spaces. There can be no spaces if the dot stands between parentheses. Thus `((small).(baby))` will be OK.

The list

We are already familiar with simple lists. What is the relation of lists and dotted pairs? If we denote arbitrary S-expressions by a_1, a_2, a_3, ..., a_k,
then the list (a_1 a_2 a_3 ... a_k) is equivalent to the expression (a_1 . (a_2 . (a_3 . (.... (a_k) . *nil*) ....))))
where `nil` is the special atom, which denotes the end of the list. The atom NIL itself means an empty list and is equivalent with the expression (). The elements of the list are separated with spaces or with commas. The *arbitrary number of spaces* or *exactly one comma* can stand between two subsequent elements. If the two elements are already separated by comma, then no more spaces can stand between them (comma is not allowed in some Lisps). The notions (a b c) and (a,b,c) are then equivalent. However, the first one is more common. The elements surrounded by parentheses, i.e. the elements other than atoms, must not be separated with spaces from their neighboring elements (but they can).

The elements of the list can also be the lists, lists of lists, etc. The examples of lists and their dotted pair counterparts are the following:

```
(A B C) is equivalent to (A . (B . (C . NIL)))
((A B) C) is equivalent to ((A . (B . NIL)) . (C . NIL))
(A B (C D)) is equivalent to (A . (B . ((C . (D . NIL)) . NIL)))
(A) is equivalent to (A . NIL)
((A)) is equivalent to ((A . NIL) .  NIL)
(A (B . C)) is equivalent to (A . ((B . C) . NIL))
```

The above examples show evidently the simplicity and efficiency of the list notation, as compared to the dotted pair notation. Understanding of the dotted pair notation is however useful for data structures more complex than simple embedded lists.

The two notations are not equivalent - the dotted notation is more general and more S-expressions can be used in it. It is relatively less used in examples, but it is very useful to represent Binary and other Decision Diagrams, when the data are large and we want to save on the number of pointers.

All lists except the empty list () can be written as the dotted pairs. On the other hand it can be observed that the expressing `(A . B)` has no counterpart in the list notation. It can be proved that the necessary and sufficient condition for the S-expression in dotted pair notation to have its counterpart in the list notation is the requirement, that for each dotted pair from this expression the second element is either NIL or an expression which can be transformed to the list notation. We leave it to the reader as an exercise. If a transition from the one to the another notation it is possible, then the respective expressions can be obtained according to the following algorithms.

2.3.2 Algorithms for Changing Notations

Algorithm of transition from the list notation to the dotted pair notation

1. Replace the empty list with NIL.

2. If the list is not empty, then replace it with the dotted pair with the first element of the list as the first element, and the list obtained by deleting the first element from the list as the second element of the pair.

3. Apply rules 1 and 2 to the list and all its sublists as soon as possible.

Examples of application of the algorithm.

```
(A B C) ->rule2-> (A . (B C)) ->rule2-> (A . (B . (C)))
->rule2-> (A . (B . (C . () ))) ->rule1-> (A . (B . (C . NIL )))

((A B) C (D E)) ->rule2-> ((A B) . (C  (D E))) ->rule2-> ((A B) . (C . ((D E))))
->rule2-> ((A . (B . ())) . (C . ((D E)))) ->rule1->
((A . (B . NIL)) . (C . ((D E) . ()))) ->rule1->
((A . (B . NIL)) . (C . ((D E) . NIL))) ->rule2->
((A . (B . NIL)) . (C . ((D . (E)) . NIL))) ->rule2->
((A . (B . NIL)) . (C . ((D . (E . ())) . NIL))) ->rule1->
((A . (B . NIL)) . (C . ((D . (E . NIL)) . NIL)))
```

The order of applying rules of the same number is arbitrary.

Algorithm of transition from dotted notation to list notation

A1. Replace NIL with the empty list.

A2. If the list is the second element of the dotted pair, then replace this pair with the list, composed of the first element of the pair and all the elements of the list, which stands as the second element of the pair.

A3. Repeat rules A1 and A2 whenever possible.

```
(A . (B . (C . NIL))) ->ruleA1-> (A . (B . (C . () ))) ->ruleA2->
(A . (B . (C))) ->ruleA2-> (A . (B C)) ->ruleA2-> (A B C)

((A . (B . NIL)) . (C . ((D . (E . NIL)) . NIL)))  ->ruleA1->
((A . (B . ())) . (C . ((D . (E . () )) . ())))  ->ruleA2->
((A . (B)) . (C . ((D . (E)) . ()))  ->ruleA2->
((A B) . (C . ((D  E)  . ()))) ->ruleA2->
((A B) . (C . ((D  E)))) ->ruleA2->
((A B) . (C  (D  E))) ->ruleA2->
((A B) C  (D  E))
```

In the above example, several rules were used at once for simplification.

2.3.3 S-Expressions

Now we specify the precise definition of the S-expression, using Backus-Naur Form (BNF) notation. In this notation "::=" means "equal from definition", and "|" means alternative.

For instance, the notation

```
<digit> ::= 1|2|3|4|5|6|7|8|9|0
```

defines the digit which can be 1 or 2 or 3 ... or 0. The definition of the expression is the following:

```
<S-expression? ::= <atom> | <dotted pair> | <list> ;
<atom> ::= <literal atom> | <number> ; *
<sequence of characters> ::= <character> | <character> <sequence of characters> ;
<character> ::= <letter> | <digit> ;
<letter> ::=a|A|b|B.... Y|z|Z ;
<digit> ::= 1|2|3|4|5|6|7|8|9|0 ;
<dotted pair> ::= (<S-expression> . <S-expression>) ;
<list> ::= ( <sequence of S-expressions> ) | () ;
<sequence of S-expressions> ::=
                  <S-expression> | <S-expression> <sequence of S-expressions> ;
```

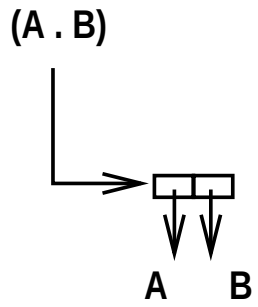


Figure 2.2: Graphics for the dotted pair `(A . B)`

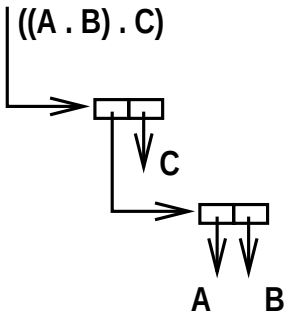


Figure 2.3: The expression `((A . B) . C)` interpreted graphically

For simplicity, the above definition does not take into account the previously described rules of using spaces and commas. It results from this definition that the atom, the dotted pair and the list are the S-expressions. The elements of the lists and the dotted pairs can be arbitrary S-expressions. The complex expressions can be created, which are the mixtures of dotted pairs and lists and have a practically unlimited numbers of atoms. [1]

2.3.4 Computer Realization of S-Expressions

We will present now how the S-expressions are placed in the memory and will also present their graphical representation.

The method of representation of an atom in the computer memory will be initially not specified. We will only signalize that some information about the atom is stored, as the alphanumeric code of its characters, the value of the atom, and the other. We assume also initially, that one memory cell is subordinated to each atom when writing the S-expression. The number being the address of this cell will be replaced with the name of the respective atom. If we write then, that the value `ALPHA` is in the cell of address 125, then it will mean that the cell of address 125 includes the address of the cell corresponding to the atom `ALPHA`. We will also say that the cell of address 125 includes a *pointer* to the cell corresponding atom `ALPHA`.

We will also assume that the memory which implements the S-expressions has 32 bit wide cells. We have also mentioned that one memory cell is subordinated to each atom. To implement the dotted pair, the memory cell respective to this pair is divided into two parts, each of them has 16 bits. The first part includes the address of the cell corresponding to the first element of the pair (the pointer to the word corresponding to the first element of the pair). The second part includes the address (the pointer) of the cell corresponding to the second element of the pair. This can be graphically interpreted with the box of two parts. For instance, the dotted pair `(A . B)` can be graphically represented as in Fig. 2.2.

We remind that `A` and `B` in the figure are the pointers to the cells corresponding to atoms `A` and `B`. Of course, arbitrary complex S-expressions can be interpreted in a similar way. If one of the pair's elements is not an atom, then the respective part of the box in the graphic representation is left empty and joined by an arrow with the graphic representation of this element.

For instance the expression

```
((A . B) . C)
```

can be interpreted graphically as in Figure 2.3.

[1] Definition of numbers will be given in chapter 3.

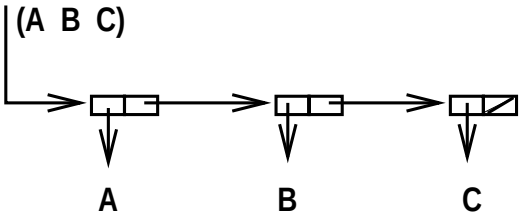


Figure 2.4: List (A B C) presented graphically.

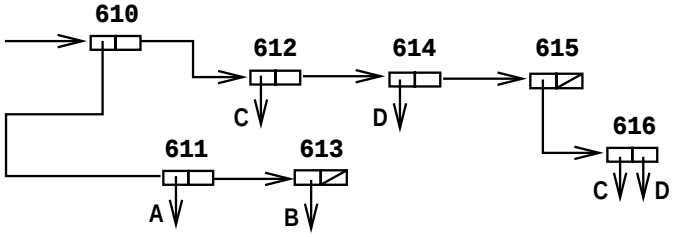


Figure 2.5: Graphic interpretation of `((A B) C D (C . D))`

Below we present the method illustrating how the expression is implemented in computer memory. The left column includes the address of the cell, the middle column - the contents of the first part of the cell, the right column - the contents of the other part of the cell. All addresses are given as the octal numbers. * [2]

```
000320 000321 C
000321    A   B
```

We assume that the memory cell of address 320 (octal) corresponds to the S-expression. The implementation of the list in box-and-arrows notation is done as follows:

In the first part of the first cell of the group of cells corresponding to the list, the pointer to the first element of this list is placed, while the second part of the cell includes the pointer to the list, which occurs from the initial list by deleting its first element. This list is implemented further in analogous way, until the empty list is obtained. Instead of a pointer to the empty list, the pointer to NIL atom is created (in the graphic representation this is done by crossing the respective part of this box). For instance, the list

```
( A B C)
```

can be graphically presented as in Figure 2.4, which in the computer can be implemented as follows:

```
620  A  621
621  B  622
622  C  NIL
```

It was assumed in this example, that the cell of the address 620 corresponds to the implemented list. This notation will become quite useful when we will discuss hardware implementation of Lisp.

We will give now some examples of computer implementation and respective graphic representation of the more complex S-expressions.

1. `((A B) C D (C.D))`
Graphic interpretation in Fig. 2.5
Computer implementation

```
  0 0 0 6 1 0    0 0 0 6 1 1    0 0 0 6 1 2
  0 0 0 6 1 1          A        0 0 0 6 1 3
  0 0 0 6 1 2          C        0 0 0 6 1 4
  0 0 0 6 1 3          B            NIL
  0 0 0 6 1 4          D        0 0 0 6 1 5
  0 0 0 6 1 5    0 0 0 6 1 6        NIL
  0 0 0 6 1 6          C             D
```

[2]* It is useful that Lisp user knows this notation.

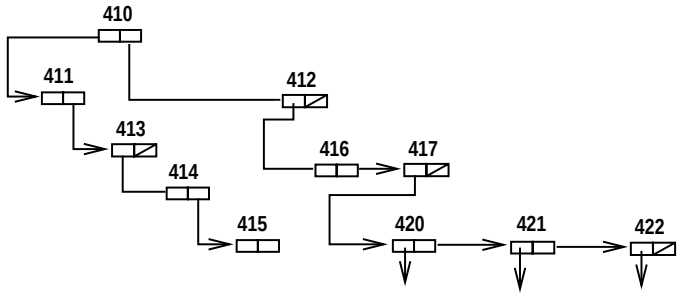


Figure 2.6: Graphic interpretation of expression `((A (B C) (A (X Y Z )))`

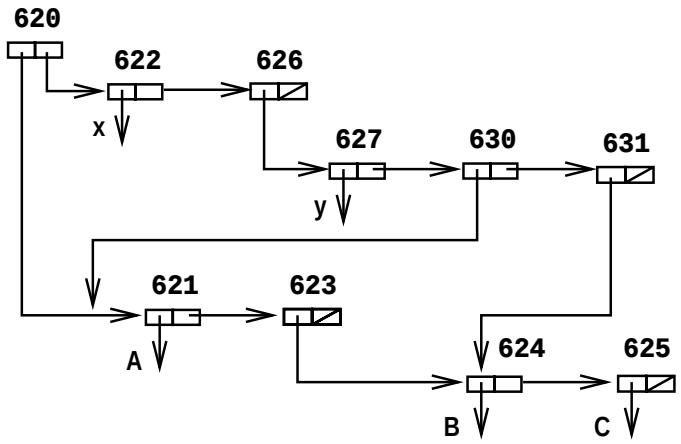


Figure 2.7: Graphical interpretation of expression `((A (B C)) X (Y (A (B C)) (B C)))`

2. `((A (B C) (A (X Y Z )))`
Graphic representation given in Fig. 2.6.
Computer implementation:

```
0 0 0 4 1 0    0 0 0 4 1 1    0 0 0 4 1 2
0 0 0 4 1 1          A        0 0 0 4 1 3
0 0 0 4 1 3    0 0 0 4 1 4        NIL
0 0 0 4 1 4          B        0 0 0 4 1 5
0 0 0 4 1 5          C            NIL
0 0 0 4 1 2    0 0 0 4 1 6        NIL
0 0 0 4 1 6          A        0 0 0 4 1 7
0 0 0 4 1 7    0 0 0 4 2 0        NIL
0 0 0 4 2 0          X        0 0 0 4 2 1
0 0 0 4 2 1          Y        0 0 0 4 2 2
0 0 0 4 2 2          Z            NIL
```

If there exists several indentical subexpressions in the given S-expression, then they can be implemented as a single subexpression.

3. `((A (B C)) X (Y (A (B C)) (B C)))`
Graphical representation

```
0 0 0 6 2 0    0 0 0 6 2 1    0 0 0 6 2 2
0 0 0 6 2 1          A        0 0 0 6 2 3
0 0 0 6 2 3    0 0 0 6 2 4        NIL
0 0 0 6 2 4          B        0 0 0 6 2 5
0 0 0 6 2 5          C            NIL
0 0 0 6 2 2          X        0 0 0 6 2 6
0 0 0 6 2 6    0 0 0 6 2 7        NIL
0 0 0 6 2 7          Y        0 0 0 6 3 0
0 0 0 6 3 0    0 0 0 6 2 1    0 0 0 6 3 1
```

```
0 0 0 6 3 1    0 0 0 6 2 4        NIL
```

The implementation of the arbitrary S-expression needs not be written into the successive memory cells, but can be in arbitrary way placed inside the area of memory specified by the operating system of the computer and the LISP interpreter to our specific job. This permits for flexible storage allocation and for creating and extending arbitrary S-expressions in arbitrary way.

2.3.5 Problems

1. Specify if the below expressions are the atoms:
   (a) ALFA
   (b) (ALFA)
   (c) A1. FA
   (d) A13
   (e) 113
   (f) 1A1
   (g) (A . B)
2. Specify if the below expressions are the S-expressions:
   (a) ALFA
   (b) (ALFA)
   (c) ( )
   (d) NIL
   (e) (A. 12)
   (f) (X Y Z)
   (g) (X Y . Z), interpreter understand this as (X . (Y . Z))
   (h) ((X . Y) (A . B) (C D E))
   (i) ((ABC)
3. Write the following S-expressions in list notation:
   (a) (ALFA . NIL)
   (b) (X . (Y . (Z . NIL)))
   (c) (X . (A B)) . (X Y))
   (d) (((A . NIL) . NIL) . (((D . (E . NIL)) . NIL) . NIL))
4. Write the following S-expressions in dotted pair notation:
   (a) (A B C D)
   (b) ((A B) (X Y (A B)))
   (c) ((X Y) X Y (A . B))
   (d) (((A) B) C)
5. Give the graphic interpretation for the following S-expressions.
   (a) (A (B C D) (X . Y))
   (b) (X Y Z (A . B))
   (c) (ALFA . (X . (Y Z)))
   (d) (A (X (Y . Z)) (A . (X (Y . Z))) (Y . Z))
6. Write in any notation the S-expressions with the graphical representations presented in Fig. 2.8.
7. Propose the method of writing the S-expression for the following array:

```
| A  13   7 |
| X   B   6 |
| Y  12   3 |
| Z   8   2 |
```

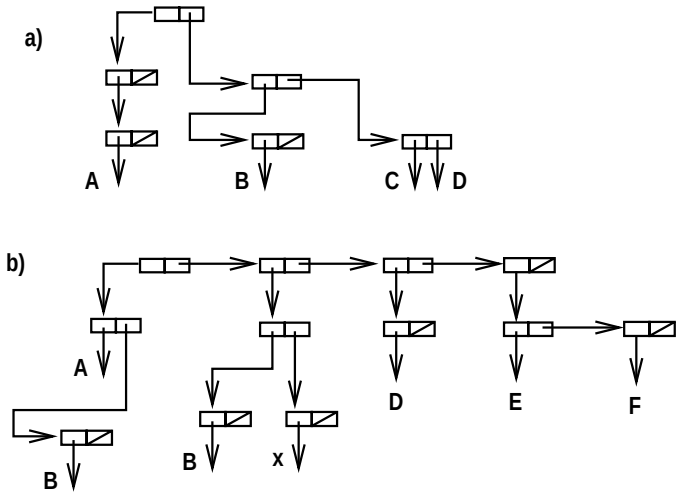


Figure 2.8: Graphic interpretation of S-expressions to Problem 6.

2.3.6 Solutions to Problems

1 a) yes b) no c) no d) no e) yes f) no g) no.
2 a) yes b) yes c) yes d) yes e) yes f) yes g) no h) yes i) no
3 a) `(ALFA)` b) `(X Y Z)` c) `((X A B) X Y)` d) `(((A)) ((D E)))`
4 a) `(A . (B . (C . (D . NIL))))`
b) `((A . (B . NIL)) . ((X . (Y . ((A . (B . NIL)) . NIL))) . NIL))`
c) `((X . (Y . NIL)) . (X . (Y . ((A . B) . NIL))))`
d) `(((A . NIL) . (B . NIL)) . (C . NIL))`
5 See Figure 2.9a,b,c,d.
6 a) `(((A)) . ((B) . (C . D)))`
b) `((A B) ((B) X) (D) (E F))`
7 `((A 13 7) (X B 6) (Y 12 3) (Z 8 2))`

2.4 Constants, Variables and Functions in Lisp

2.4.1 Numbers

In the last section we have introduced the concept of a number as the atom, which can stand in an S-expression on equal terms with the literal atoms. Now we will precise and extend this concept.

There are three kinds of numbers in Lisp:

1. Integers. These are the atoms composed of the sign (+ or -) and the sequence of digits (minimum one), among which no any other characters can stand. Character "+" can be omitted. The admissible number of digits, which are included in the given number is in each implementation limited by the actual hardware.

   Examples: -13, 312023.

2. Floating-point decimal. These are the atoms consisting of the sign (+ or -) and the sequence of digits, among which single decimal point can stand. This point (a dot) cannot occur as either the first or the last element in the sequence of digits. Sign + can be omitted. At the end of the number can stand the description of the exponent, consisting of letter `E` followed by the natural number, being the exponent of number 10, as is usually applied.

   Examples: 20.0, 2.E1, 200.00E-1, 0.2E+1 . There can be no space on the left and the right side of the decimal dot. For instance, 2 .E1 causes an error. All the above examples present the same number. The erroneous (in sense of Lisp) numbers are .2E+2 and 20. .

3. Octal numbers. These are atoms composed of the sign (+ or -), the sequence of digits 0 - 7 and letter `Q`. Sign "+" can be omitted. The letter `Q` can be followed by the exponent (zero, or natural number without sign). These numbers are interpreted as numbers in octal system, while occurrence of exponent causes multiplication of a given number by the respective power of 8.

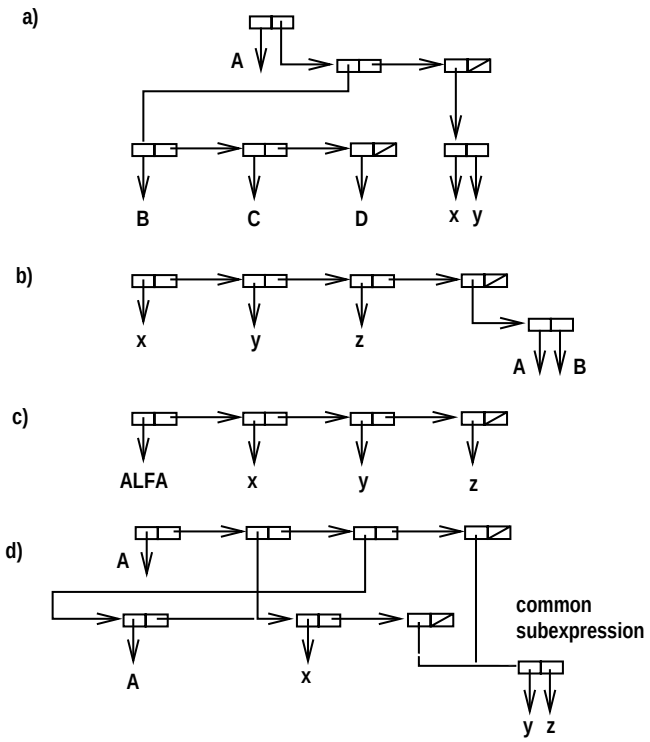


Figure 2.9: Solutions to Problem 5

Examples: 77Q, 21Q4 -3Q11, +1Q7. The octal numbers are written analogously at the computer's output. In the case of programs directly processing the bits, they serve to present in a readable form the state of bits of the memory cell. (They can be thus used to represent sets and Boolean functions, as will be shown in subsequent sections.) For instance, the cell of the contents

000 000 001 111 010 011

is printed as

001723Q

There exist some limitations specified for the respective language implementation, as to the number of digits in the number and the absolute value of the number, both in the case of the floating-point numbers as well as the octal numbers.

Some uncertainty may be caused by using the dot both for floating point numbers and in the dotted pairs. Practically such uncertaintes occur very rarely. In order to eliminate them it is sufficient to assume that if the numbers stand as the elements of the dotted pair then the second element is supplied with its sign. Then

(6.+5)

is treated as the dotted pair, while

(6.5)

is treated as a list composed of a single element, number 6.5.

Octal numbers and integers are not distinguished for arithmetic operations; both are considered to be fixed-point. They differ only in their printed representation. All of the arithmetic functions may accept either fixed-point or floating-point inputs or a mixture of the two. The result is floating-point if any of the inputs are floating-point and will be fixed-point otherwise.

2.4.2 Constants

Now we will present one of the most important concepts of language Lisp - the *symbolic atom* (abbreviated, atom). To each atom corresponds some value (the value can be, however, not always specified). With respect to the value we can distinguish *the atoms having the constant and always specified, (the constants),* and *those whose values can be specified arbitrarily and changed during program execution (the variables).*

The constant of language Lisp is an atom whose value is constant and always specified.

All numbers are constants in Lisp. Their values result from their description. The constants of Lisp are also the literal atoms: T, F, and NIL. Their values are *T*, NIL and NIL, respectively. [3]

2.4.3 Variables

A variable of Lisp is an atom to which arbitrary value can be assigned. Before the first specification of the values, the value of the variable is undefined.

The variables of Lisp can be arbitrary symbolic atoms different than T, F, and NIL. Each symbolic atom, except for the atoms mentioned above, can be treated as a variable.

We must explicitely distinguish the notions of the *name of the atom* and the *value of the atom.* In languages such as Fortran and Basic these concepts are easy to distinguish, because the names of variables are symbolic atoms and their values are numbers. In Lisp arbitrary S-expressions (and the atom in particular case) can be the variables's value.

The reader's ability to distinguish these two concepts well is the necessary condition for writing any program in Lisp.

The variables are in Lisp usually the arguments of some functions. In such case we use *names* of variables in the calls, and the functions being executed process the *values* of these variables. The only exception are the so-called *special forms* (presented later on) whose execution is slightly different.

2.4.4 Functions

We will discuss now the concept of the function in Lisp.

The *functional expression* is the S-expression written in the form of a list, whose first element is the function's name and the next elements are the successive arguments of the function.

The name of the function can be an arbitrary literal atom, different than NIL, F, and T. It is not especially important if this atom exists somewhere else in the program as a name or as a value of a variable. The meaning of symbolic atom (being it the name of a function, the name of a variable or the value of a variable) depends solely on the context of its occurence.

The argument of the function can be an arbitrary variable, constant or functional expression. This can in particular be a functional expression of the identical name of the function.

Execution of Lisp function consists in execution of some operations (specified by the name of the function) on values of its arguments). The execution of the function is characterized by two elements, behavior of the function and the value of the function.

The behavior of the function is the collection of the transformations executed by it on the values of variables. *The value of the function* is the final value assigned to this functional expression. Arbitrary S-expression can be the value of the function.

The execution of the function includes two stages:

1. Calculating (evaluating) the values of arguments of the function. This stage can be very complex because the function's arguments can be the functional expressions, whose arguments are also functional arguments, and so on.

   The number of arguments for some functions can be arbitrary, but for most of them it is strictly specified. If the number of the arguments in the functional expression is not proper or if the values of some of the arguments cannot be calculated (some variables have no value specified), then the execution of the function is interrupted and the error is signalized.

2. Execution of operations, attached to this function and calculation of the function's value. At this stage the operations attached to these functions are executed, using the *values of the arguments* (and the *names of the arguments* in the case of some special functions) and created is the S-expression being the value of this function.

The method of execution of the function will be discussed on some abstract level. Let us assume that the functional expression has the form.

```
(A B ( C (D) E))
```

While execution of this expression atom `A` is interpreted as a name of the function whose arguments are: the variable `B` and the functional expression `(C (D E))`. Let us assume that function `A` is specified as having two arguments. The values of its arguments are calculated in order to execute it. The value of the first argument is the value of variable `B` (we assume that it is specified). In order to calculate the value of the second argument the value of the functional expression `(C (D) E)` must be first calculated.

[3] In some Lisps, the values of T and F can be redefined

Atom C in this expression is interpreted as the name of a function, whose arguments are: functional expression (D) and variable E. To calculate the value of the first argument, the value of the functional expression (D) is evaluated. Atom D is interpreted as a name of the non-argument function. After execution and calculation of the value, the value of the first argument of function C is specified. The value of the second argument is the value of variable E (we assume that it is specified). Now we can start to execute function C, while the values of both its arguments have been specified (we assume that C is a two-argument function). After execution of this function and evaluation of its value - this value gets to be the value of the second argument of function A. Now the function A can be finally executed and its value evaluated - this value being also the value of the entire functional expression.

Of course, in the example presented above, the atoms A, C, D must be the names of some functions, specified at the moment of entering the execution of the above functional expression.

It arises from the above procedure that only those functions are executed, which already do not include functional expressions as arguments or whose arguments have been already previously evaluated. This principle does not apply to some special functions called the *pseudo-functions* and to the *special forms.*

The *special forms* are the functions which possess at least one of the properties specified below:

1. they process the non-specified (arbitrary) number of arguments.
2. they process the non-evaluated arguments (or - names of arguments - using other notions) or do not evaluate their names before starting the execution of the function.

Pseudofunctions are some special functions which behavior is more general than in ordinary functions. This results in the occurence of the so-called *side effects* of function's evaluation. This effect can be: assigning values to some variables, creating or transforming some data structures which can be used both by Lisp interpreter and the executed program. The side effect can consist also in the execution of some input/output directives or other control statements. The values and their transformations caused by the side effects of function execution can have nothing in common with the evaluated value of this function and are preserved in the system after completion of its execution, in contrast to the values of the auxiliary variables which exist only inside the function.

In the next section we will present the most important functions of Lisp. In many cases these are the pseudofunctions and the special forms as well as the functions having properties of both of them.

We will use the following notation to describe the functions.

```
DESCRIPTOR          MEANING
==========          =======
ATOM        The argument must be an atom, either a symbolic atom or a numeric atom.
BOOLEAN     The argument may be any S-expression, but it will be interpreted as a truth value:
              NIL is equivalent to false; anything else is equivalent to true.
CHARACTER   The argument must be a symbolic atom whose name is a single character.
EXP         The argument must be some Lisp expression which can be evaluated by EVAL.
FILENAME    The argument must be a symbolic atom containing at most seven letters
              and digits, starting from a letter.
FLNUMBER    The argument must be a floating-point numeric atom.
FIXNUMBER   The argument must be a fixed-point numeric atom (either integer or octal).
FNEXP       The argument must be a functional expression. Either it must be the name of
              a function preceded by QUOTE, or FUNCTION, or it must be a Lambda-expression
              preceded by QUOTE, FQUOTE or FUNCTION.
FUNCTION    The argument must be a function name, Lambda-expression, or label expression.
FW          The argument must be a single full-word, that is a word in full-word space.
FWL         The argument must be a list of full-words.
LAT         The argument must be a list of literal atoms.
LETTER      The argument must be a literal atom whose name is a single letter (aA..zZ).
LIST        The argument must be a list or NIL.
LITATOM     The argument must be a single symbolic atom.
NATS        The argument must be some non-atomic S-expression.
NUMBER      The argument must be a numeric atom of any type.
S           The argument must be an arbitrary S-expression.
```

2.4.5 Review Questions

1. What kinds of numbers exist in Lisp. Give examples.
2. What is a Lisp constant? Give examples.
3. What is a Lisp variable? Give examples.
4. What is a functional expression? Give examples.
5. What is the difference between the pseudofunctions and special forms and ordinary Lisp functions? Give examples explaining any of them.

2.5 Basic Functions of Lisp

We can distinguish a certain set of Lisp functions called the *basic functions* * [4] These are the functions which execute the most basic operations, as assigning values to variables or simple transformations of S-expressions.

By appropriate composition of basic functions, we can define many complex Lisp functions of various behaviors. Some of these functions will be discussed in the following sections of this chapter, and next in the subsequent chapters.

Discussion of the basic functions will be started from those functions, which can be used for assigning values to the variables of Lisp. Such approach will enable us to discuss and interprete the behavior of the remaining functions.

2.5.1 Few More Important Functions of Lisp

The three most important functions to transform S-expression are `CAR`, `CDR` and `CONS`.

CAR.

```
(CAR NATS)
NORMAL SUBR
```

`CAR` is a one-argument function. The value of its argument can be arbitrary S-expression but not an atom. `CAR` returns the first element of the list or the dotted pair being the value of the argument. The `CAR` of an atom gives usually an error, but you must consult it in your system.

Let us assume that our previous assignments of values are still in our computer, for instance, the value of variable `neighbors` is `(Jane Mary Elizabeth)`.

Examples:

```
(CAR '(Jane Mary Elizabeth)) ==> Jane
(CAR neighbors) ==> Jane
(CAR expression_1) ==> +
(CAR 4) ==> ERROR
(CAR '((small house) in a prairie)) ==> (small house)
(CAR '(dotted . pair)) ==> dotted
(CAR '((dotted . pair) . (another . dotted_pair))) ==> (dotted . pair)
(CAR 'A) -> nonspecified
(CAR nil) ->
(CAR 'NIL) ->
```

Let us assume, that we forgot to quote the list `(Jane Mary Elizabeth)` in the first call:

```
(CAR (Jane Mary Elizabeth)) ==> FUNCTION Jane NOT DEFINED
```

In such case system treats expression `(Jane Mary Elizabeth)` as the expression to evaluate. `Jane` is treated as a function name. Because function of such name is not defined, an error occurs.

Let `((A B) (C D) E)` be the value of variable `A`. We want to evaluate expression `(CAR A)`. System treats the first atom - `CAR` from this list as the name of the function. Atom `A` is the name of the function's argument. Because the value of this argument is the S-expression `((A B) (C D) E)` then `CAR` will select the first element of this list, i.e. S-expression `(A B)`.

```
(SETQ A '((A B) (C D) E)) ==> ((A B) (C D) E)
(CAR A) ==> (A B)
```

It is very useful to have a mental imagination how the basic functions operate on Lisp data-structures. S-expression `((A B) (C D) E)`, being the value of variable `A` can be graphically represented as in Fig. 2.10a. The value of the expression `(CAR A)` is the S-expression `(A B)`, which can be presented as in Fig. 2.10b.

Let us observe, that the pointer to the value of function `(CAR A)` is created by shifting the pointer which originally points to the value of argument `A` of this function - from the box to which it points, to the box pointed by the arrow outcoming from the first from left part of the box of argument. The first part of the memory cell will be then called *the* `CAR` *part.*

And how all this is realized in a computer memory?

For the same as before value of variable `A` we will assume that the cell of the address 250 is subordinated to the value of this variable.

[4]* To make the life of our reader easier, we will give a little different set of basic functions than Lisp purists, for instance [?] and [?].

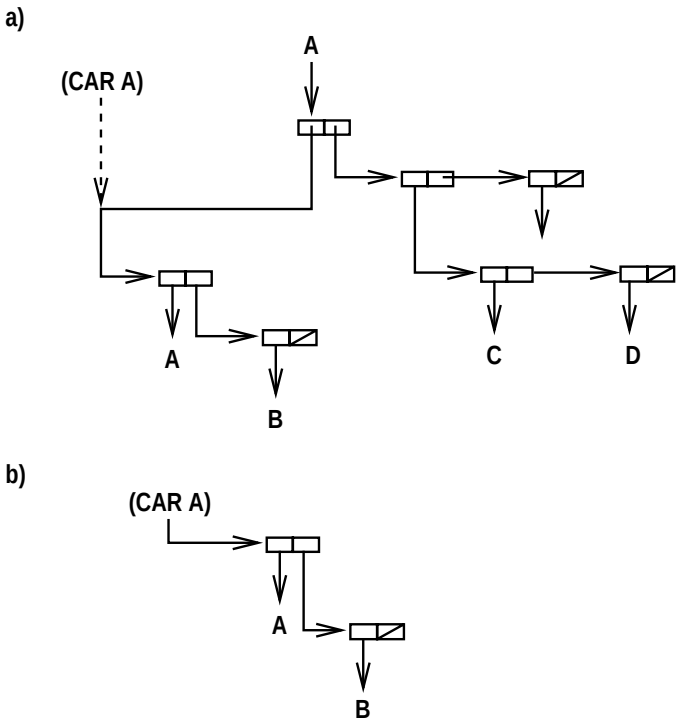


Figure 2.10: S-expression `((A B) (C D) E)`, being the value of variable `A` presented graphically

```
250      251     252
251       A      253
253       B      nil
252      254     255
254       C      256
256       D      nil
257       E      nil
```

In this case, the cell with the address from the first part of the cell corresponding to the value of variable `A` is subordinated to the value of expression `(CAR A)`. This is the cell with address 251.

Few more examples:

```
(CAR 'A) ==> undefined
(CAR '(A.NIL)) ==> A
(CAR '(A)) ==> A
(CAR '() ) = (CAR (QUOTE NIL)) ==> undefined
(CAR '(()) ) ==> NIL
```

2.5.2 CDR

```
(CDR NATS)
NORMAL, SUBR
```

`CDR` is a one-argument function. The value of its argument can be an arbitrary S-expression, except for the atom. The value of `CDR` is the second element from the dotted pair. If argument is a list, the value of `CDR` is the remaining of the list after removing from it the first element. The `CDR` of an atom is unspecified.

```
(CDR '(dotted . pair)) ==> pair
(CDR '(more . (dotted . pair))) ==> (dotted . pair)
(CDR '(more dotted pair)) ==> (dotted pair)
(CDR 'atom) ==> unspecified
```

Let S-expression `((A B)(C D) E)` be the value of variable `A`. By evaluating the value of expression `(CDR A)` the atom `CDR` is interpreted as the function's name and atom `A` as the variable's name being the argument of the function.

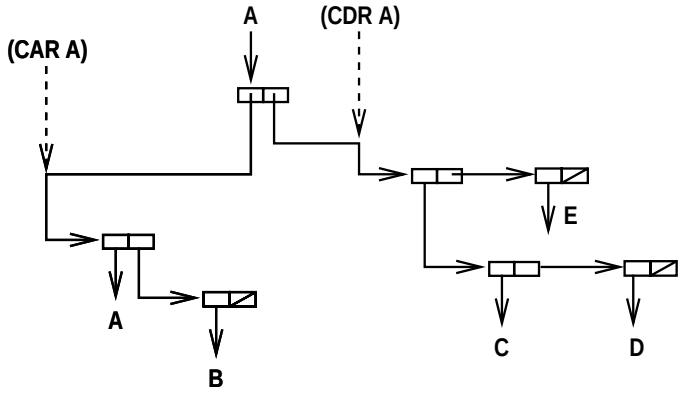


Figure 2.11: Graphical interpretation of execution of CDR

The value of this argument ((A B)(C D) E) is evaluated. Function CDR deletes the first element from this list and the S-expression ((C D) E) is returned as the value:

```
(CDR A) ==> ((C D) E)
```

The graphical interpretation of execution of CDR is presented in Fig. 2.11.

As we see in the figure, the pointer to the function's value (CDR A) is created from the pointer to the value of argument A of this function by shifting it to the cell pointed by the pointer outcoming from the second part of the cell that corresponds to the value of argument A. For completeness, the pointer to CAR of the same argument is also shown.

The computer interpretation of the CDR function consists in returning as its value the pointer to the cell from the second part of the cell being the value of the argument. Therefore, the second part of the computer cell is called its CDR *part.* For the example discussed above the pointer to the cell with address 252 is returned as the value.

Some more examples:

```
(CDR neighbors) ==> (Mary Elizabeth)
(CDR expression_1) ==> () ; the same as NIL or nil, system depending.
(CDR '(something . nil)) ==> nil
(CDR '(something)) ==> nil
```

As we remember, the arithmetic expressions which are arguments to another arithmetic expression were evaluated. The same is true for arbitrary Lisp expressions. Let us consider for instance the functional expression

```
(CAR (CAR '((A B)(C D))))
```

By evaluating this expression the first from left CAR evaluates its argument. Now the value of expression (CAR '((A B)(C D))) is evaluated. It equals (A B) and is returned as the value of internal (second from left) CAR. The expression (CAR '(A B)) is now calculated for the external CAR. Atom A is returned as the value of the entire expression, thus:

```
(CAR (CAR '((A B)(C D)) )) ==> A
```

Compositions of functions CAR and CDR can written in simplier form, for instance the above expression is equivalent to:

```
(CAAR '((A B)( C D))) ==> A
```

The sequence of two letters A in the function's name means the composition of two functions CAR, where the value of the second function CAR is taken as the argument of the first CAR.

Similarly, we can create arbitrary compositions of CAR and CDR:

```
(CDDR A) is equivalent to  (CDR (CDR A)),
(CADR A) is equivalent to (CAR (CDR A)),
(CADAR A) is equivalent to (CAR (CDR (CAR A))).
```

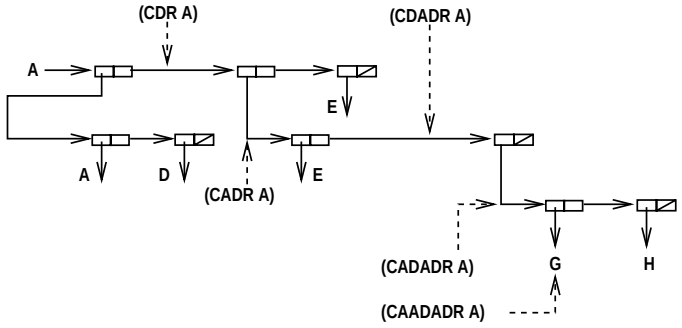


Figure 2.12: `(CAADADR A)` $\Rightarrow$ `G` presented graphically

and so on...

Example.
Let `A ==> ((A D) (E (G H)) E)` and the value of the expression `(CADADR A)` is to be evaluated. Because `D` is the last letter in the function's name before `R` then `CDR` is evaluated as the first function: `(CDR A)` with the value of

$$((E(GH))E) \tag{2.4}$$

The next letter (looking from left to right) is `A`. The value of function `CAR` is then evaluated with the argument being the value of expression 2.4.
The expression

$$(E(GH)) \tag{2.5}$$

is returned as the result. Next (letter `D`) the function `CDR` is called for argument of expression 2.5. As the result

$$(GH) \tag{2.6}$$

is returned. At last, the value of `CAR` (letter `A`) from expression 2.6 is returned as the value of the entire expression:

```
(CADADR '((A D)(E (G H)) E)) ==> (G H)
```

The above procedure can be summarized as follows:

```
(CADADR A) ==> (CADADR '((A D) (E (G H)) E))) ==>
(CADAR '( (E (G H)) E))) = (CADR '(E (G H))) ==>
(CAR '((G H))) ==> (G H)
```

Then

```
(CADADR A ) ==> (G H)
```

Similarly we can show that

```
(CAADADR A) ==> G
```

This can be presented graphically as in Fig. 2.12.
Assuming the same value for `A` as before, we have:

```
(CAAR A) ==> A
(CADAR A) ==> D
(CDDAR A) ==> nil
(CDDR A) ==> (E)
(CADDR A) ==> E
(CAADR A) ==> E
```

Analyzing the graphical interpretations of the above functions, we can see that the pointers to the respective cells can be obtained by shifting the output pointer as many times as there are letters between leters `C` and `R` in the function's name. The name shall be analyzed from right to left. Letter `A` causes shift as for `CAR` and letter `D` the shift respective to `CDR`. The computer realization of these operations is analogical and is left to the reader. No more than 8 letters `A` and `D` can stand between `C` and `R` (in general, this value depends on Lisp implementation).

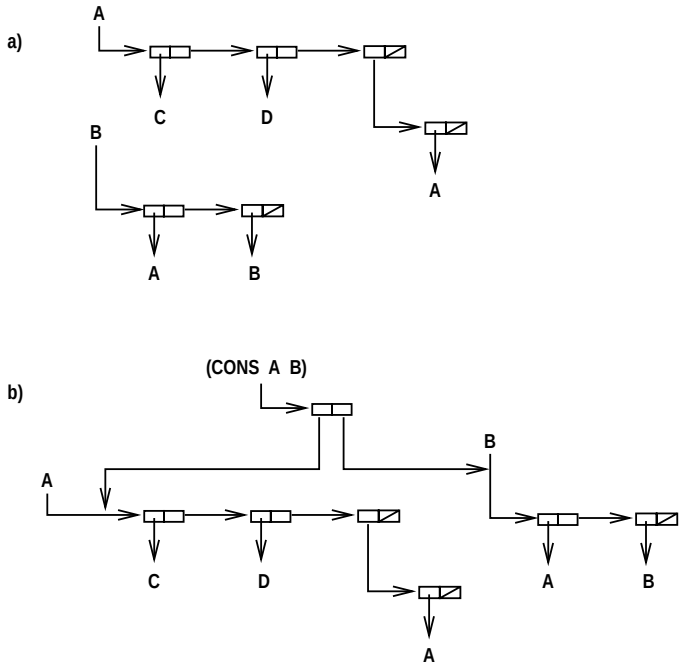


Figure 2.13: Graphical interpretation of execution of `CONS`: a) arguments, b) a result together with arguments

2.5.3 CONS

```
(CONS SI S2)
```

Functions `CAR` and `CDR` were used to select parts of expressions. How can we construct new expressions? We need a new function for this. This respective function is called `CONS` (for *Constructor*). `CONS` is a two-argument function that takes arbitrary S-expressions as arguments. The value of the expression `(CONS a b)` is the dotted pair whose first element is the value of `a` and the second element is the value of `b`. When possible, computer prints the result in the *list notation*, performing respective transformations (from the dot to list notations) whenever possible.
Example:

```
(CONS '(A B) '(C D))  ==> ((A B ) C D)
```

By evaluating this expression atom `CONS` is interpreted as the function's name , with `'(A B)` and `'(C D)` as arguments. The values of the arguments are `(A B)` and `(C D)`, respectively. According to the definition of function `CONS`, the value of this function is `((A B) . (C D))`, which can be written in the form `((A B) C D)`.

Other examples:

```
(CONS 'A 'B) ==> (A . B)
(CONS 'A '(B)) ==> (A B)
(CONS 'A '(B C D)) ==> (A B C D)
(CONS '((1 . 2) . 3)  ' ((1 . 2) . 3)) ==> ( ((1 . 2) . 3) . ((1 . 2) . 3))
(CONS 'A nil) ==> (A)
(CONS nil 2) ==> (nil . 2)
(CONS A B) ==> ((C D (A)) A B) ; with values of A and B set previously
```

Using the last example we will present now the graphical interpretation of `CONS` execution (Fig. 2.13).

As the result of `CONS` execution the new cell is created. The `CAR` of this cell is a pointer to the value of the first argument of the function. The pointer to the second argument stands in the `CDR` part. The value of the functional expression is the pointer to the newly created cell (Fig. 2.13b).

To explain the computer realization of `CONS` we will use the expression `(CONS A B)` where `(A B)` is the value of variable `A` and `(B)` is the value of variable `B`. The computer realizations of the variables is the following:

```
250     A     251
251     B     nil
252     B     nil
```

where the cell of address 250 corresponds to the value of variable A and the cell of address 252 corresponds to the value of variable B. As the result of function's execution the cell of address 253 is added, with the first part including address 250 and the second part including address 252. The cell of address 253 is returned as the value of the functional expression (CONS A B).
Let us also note that the following rules are true:

```
(CAR (CONS X Y)) ==> X
(CDR (CONS X Y)) ==> Y
```

where arbitrary S-expressions can be substituted for X and Y. Also

```
(CONS (CAR X)(CDR X)) ==> X
```

where X is an S-expression but not an atom.

Let us discuss now more examples how expressions can be composed out of another expressions. Let A ==> (A B C D) and B ==> (X (Y Z)). We want to assign the value ((C D) (Y Z) X) to variable C. Of course, this can be done by typing:

```
(SETQ C '((C D) (Y Z) X))
```

Assume however, that we are not allowed to use quote. We must then to create parts of our S-expression by selecting them in the S-expressions being values of A and B. Let us note, that the S-expression to be created is a list with three elements (C D) (Y Z) and X. The first of them can be obtained by calling (CDDR A). The second is obtained with (CADR B), and the third with (CAR B). The list can be then constructed by using repeated applications of CONS:

```
(CONS (CDDR A)  (CONS (CADR B) (CONS (CAR B) nil)))
```

Let us now check on the computer:

```
(CONS (CDDR A)  (CONS (CADR B) (CONS (CAR B) nil)))
                          ==> ((C D) (Y Z) X))
```

Now, the required value of C can be obtained as the value of:

```
(SETQ C  (CONS (CDDR A)  (CONS (CADR B) (CONS (CAR B) nil))))
                           ==> ((C D) (Y Z) X))
```

2.5.4 Form SELECT

```
(SELECT EXP
   ( EXP1.1 EXP1.2 ... EXP1.N)
   ( EXP2.1 EXP2.2 ... EXP2.N) ....
   ( EXPM.1 EXPM.2 ... EXPM.N)
     EXPM+1 )
NORMAL, FSUBR
```

SELECT allows the selection of a particular list of expressions to be evaluated depending on the value of the first argument of SELECT. It proceeds as follows. The first argument, EXP, is evaluated. Its value is then compared to the value of EXP1.1, EXP2,1,..., until the value of EXP equals the value of EXPI,1, in which case each EXPI,2, EXPI,3, ... EXPI,NI-1 are evaluated and the value of EXPI, NI is returned as a value of SELECT. SELECT is very similar to COND in this respect in that once a test is successfully completed SELECT will evaluate an arbitrary large number of associated expressions, returning the value of the last expression so evaluated. If the value of EXP is not EQAUL to the value of any EXPI.1, I = 1,.., M, then the value of EXPM+1 is returned as the value of SELECT.

2.5.5 Predicate EQ

We will pay our attention now to a special group of functions, which will be called *predicates*. The predicate is a function, that has two values, true and false. When the value is true, we say that the predicate is *fulfilled* or *satisfied*. When the value is false, we say that the predicate is *not fulfilled* or that it is *not satisfied*. In Lisp the value false is represented by F or nil. Value true is represented by atom t or by any value different from F or nil. t is printed as *T* and f as nil (or NIL, depending on the realization).

The basic predicates of Lisp are EQ and ATOM.

EQ.

```
(EQ S1 S2)
PREDICATE, SUBR.
```

EQ is a two-argument predicate. Its arguments can be arbitrary S-expressions. The value of EQ equals *T* if the same memory cells correspond to the values of both arguments.
Examples:

```
(EQ 'A 'A) ==> *T*
(EQ neighbors neighbors) ==> *T*
(EQ neighbors neighbors) ==> undefined variable neighbors
(EQ 'A 'B) ==> nil
(EQ fruit  apple ) ==> *T*
(EQ fruit  'MacIntosh) ==> *T*
(EQ '(A B) '(A B)) ==> nil
```

Execution of EQ consists in checking if the addresses of values of both arguments are identical. If they are identical, the *T* is returned and nil otherwise. In the last example the value was nil since while reading the expression from input different addresses were assigned to the first and the second expression (A B). However the location of atoms is unique, then `(EQ 'A 'A) ==> *T*`. Similarly, `(EQ A A) ==> *T*`, since the same pointer to expression with name A is called twice. You must be very careful when comparing with EQ two S-expressions which are not atoms. Also, take car when you are comparing numbers with EQ. The results depend on implementation. In general, the numbers have not necessarily unique locations in the memory. The same number used twice in the program can have two different locations in the memory, then `(EQ 2435231 2435231) ==> NIL`. In some Lisps small numbers have unique locations: `(EQ 3 3) ==> *T*`.

Concluding, EQ should be used in some applications for comparing symbolic atoms or S-expressions being not atoms. Application of EQ to numbers requires great care.

2.5.6 Predicate ATOM

ATOM.

```
(ATOM S)
PREDICATE, SUBR.
```

ATOM is a one-argument predicate. The value of its argument is an arbitrary S-expression. The value of predicate ATOM is *T* when the value of the argument is an atom (both a symbolic atom or a number) and is equal nil otherwise:

```
(ATOM 'ATOM) ==> *T*
(ATOM '(ATOM)) ==> nil
(ATOM nil) ==> *T*
(ATOM 'nil) ==> *T*
(ATOM '()) ==> *T*
(ATOM '(())) ==> nil
(ATOM 3) ==> *T*
(ATOM '(3 4 5)) ==> nil
(ATOM fruit) ==> *T* ;;    [since fruit ==> MacIntosh]
```

2.5.7 Problems

1. Write the values for the following functional expressions and next type the expressions to the computer to check your results.

   a) `(QUOTE A)`

   b) `(QUOTE (A . B))`

   c) `(SETQ A '(A B C))`

   d) `(SETQ A (SETQ B 'X))`

   e) `(CONS '(A B) '(D))`

   f) `(CAR '((A B) C D))`

   g) `(CDR '((A B) C D))`

h) `(ATOM 'X)`

i) `(ATOM (CAR '( A B )))`

j) `(EQ '(A B) 'A)`

2. Find the values assigned to variables and next verify your solutions on the computer.

a) `(SETQ A '(X Y))`

b) `(SETQ B (CONS 'A '(B)))`

c) `(SETQ D (CONS (CAAR (QUOTE ((X Y) A))) (CDR (QUOTE (A B C)) )))`

d) `(SETQ A (CONS (CDR (SETQ B (QUOTE (A B C)))) (SETQ D (CAR (SETQ C (QUOTE (X Y Z))) ))))`

3. Assuming that `(((A B) C) (X (Y Z)))` is the value of variable `A`, evaluate the following expressions

a) **(CAR A)**

b) **(CAAR A)**

c) **(CDAR A)**

d) **(CADAR A)**

e) **(CAAAR A)**

f) **(CDR A)**

g) **(CADR A)**

h) **(CAADR A)**

i) **(CADADR A)**

j) **(CAADADR A)**

k) **(CADADADR A)**

l) **(CONS (CDAR A) (CDR A))**

m) **(CONS (CAAR A) (CADR A))**

n) **(ATOM (CADR A))**

o) **(ATOM (CADADR A))**

p) **(ATOM (CAADR A))**

4. Assume that `(A (B C))` is the value of `A` and `((X Y) (Z Y))` is the value of `B`. Assign the following values to variable `C`, without using function quote in any form:

a) `(X (X Y) (B C))`

b) `((B C) (X Y))`

c) `(X Y A)`

d) `((Y X) (C A))`

e) `((B C) (A X Z) Y)`

2.5.8 Solutions to Problems

1

```
a)  A        b)  (A . B)     c)  (A B C)     d) X        e)  ((A B ) D)
f)  (A B)    g)   (C D)      h)  *T*         i)  *T*     j) *T*
```

2

```
a)  variable: A, value: (X Y)
b)  variable: B, value: (A B)
c)  variable: C, value: (A B)
d)  variable: D, value: (X B C)
e)  variable: B, value: X
    variable: A, value: (X  .  Y)
f)  variable: C, value: (X Y Z)
    variable: D, value: X
    variable: B, value: (A B C)
    variable: A, value: ((B C) . X)
```

3

```
a) ((A B) C)     b) (A B)           c) (C)              d) C        e) A
f) ((X (Y Z))    g) (X (Y Z))       h)  X               i)  (Y Z)   j) Y
k) Z             l) ((C)(X (Y Z)))  m) ((A B) X (Y Z))
n) F             o) F               p) *T*
```

4

```
a) (SETQ C (CONS (CAAR B) (CONS (CAR B)(CDR A)))
b) (SETQ C (CONS (CADR A) (CONS (CAR B) NIL)))
c) (SETQ C (CONS (CAAR B) (CONS (CADAR B) (CONS (CAR A) NIL))))
d) (SETQ C (CONS (CONS (CADAR B) (CONS (CAAR B) NIL))
                      (CONS (CONS (CADADR A) (CONS (CAR A)  NIL))  NIL)))
e) (SETQ C (CONS (CADR A) (CONS (CONS (CAR A) (CONS (CAAR B)
                      (CONS (CAADR B) NIL))) (CDAR B) )))
```

2.6 Conditional Expressions and Logic Predicates

2.6.1 Conditional Expressions

The conditional expression has the form:

```
(COND (p1 ex11  ex12  ...  ex1n)
      (p2 ex21  ex22  ...  ex2m)
                     .....
      (pr exr1  exr2   ... exrv)   )
```

where `COND` is a name of the function, `p1`, `p2`, ..., `pr` are *predicates* and `ex11`, `ex12`, ..., `exrv` are arbitrary functional expressions. The forms like (`p1 ex11 ex12 ... ex1n`) are called *clauses*. The number of clauses is unlimited. Also, the number of expressions in every clause is arbitrary. The evaluation of function `COND` consists in evaluation of the clauses, starting from the first. If predicate is non-NIL then all expressions from this clause are evaluated in their order. The value of the last expression in the clause is returned as a value of function `COND`. If the predicate in the first clause evaluates to NIL, then the next clause is evaluated. Again, if predicate in the next clause evaluates to non-NIL then the respective expressions from this clause are evaluated, else evaluation proceeds to the next clause. If values of all predicates are NIL then the value of `COND` is either NIL or undefined (depending on implementation). To avoid this we often place an atom t as a predicate in the last clause. This ensures evaluating the last sequence of expressions if none of the previous was. It is possible that there is only one expression in the clause. The clauses with only predicates and no expressions are also sometimes constructed. The method of evaluating the conditional expression can be illustrated with the following example:

```
(COND  ((ATOM A) (SETQ A (CONS A NIL)))
       (t A))
```

Let us first assume that atom `X` is the value of variable `A`. At first the value of the predicate `(ATOM A)` is evaluated. This value equals *T*. The next functional expression in the first clause: `(SETQ A (CONS A NIL))` is then evaluated. The variable `A` is then assigned a new value `(CONS 'X NIL) ==> (X)`, which is also returned as the value of the conditional expression. Notice please, that the second clause of `COND` was ignored.
Checking on computer:

```
A ==> X

(COND((ATOM A)(SETQ A (CONS A NIL))) (t A)) ==> (X)

A ==> (X)
```

If the value of `A` were initially `(X)`, then NIL would be the value of a predicate `(ATOM A)`. Then the system evaluates the next clause. Because predicate of this clause, t, is non-NIL, then the expressions in this clause are evaluated. It happens to be only one expression in this clause, namely `A`, then it is evaluated. Since it is also the last expression in this clause, its value is returned as the value of `COND`. Value of `A` equals `(X)`, then:

```
A ==> (X)

(COND((ATOM A)(SETQ A (CONS A NIL)))(t A)) ==> (X)

A ==> (X)
```

Let us now consider another example.

```
(COND ((ATOM A) (COND ((EQ A 'X) A)
                      (t (SETQ B (CONS A NIL)))  ))
      (t (CAR A)))
```

Three cases are possible:

1. Atom `X` is the value of variable `A`.
2. An atom, but different from `X` is the value of `A`.
3. S-expression other than an atom is a value of variable `A`.

In the first two cases the predicate `(ATOM A)` evaluates to *T* and the internal conditional is evaluated as the expression in the first clause. Now it is checked if it is an `X`. If yes, then value of `A` is returned as the value of both internal conditional, and next the external conditional. This value is `X`, therefore `X` is returned. In the second case an atom being the value of `A` is consed with NIL and then returned as the value. In the second case, the same value is also assigned to variable `B`:

```
A ==> X

B ==> UNDEFINED VARIABLE B
(COND ((ATOM A) (COND ((EQ A 'X) A)
                      (t (SETQ B (CONS A NIL)))  ))
      (t (CAR A))) ==> X

(SETQ A 'Y) ==> Y

B  ==> UNDEFINED VARIABLE B

(COND((ATOM A)(COND((EQ A 'X) A)
                  (t (SETQ B (CONS A NIL)))  ))
     (t (CAR A)))                              ==>  (Y)

B ==> (Y)
```

In the third case the value of the expression `(CAR A)` is returned as the value of the conditional:

```
(SETQ A '(V VV VVV)) ==> (V VV VVV)

(COND ((ATOM A) (COND ((EQ A 'X) A)
                      (t (SETQ B (CONS A NIL)))  ))
      (t (CAR A))) ==> V

A ==> (V VV VVV)

B ==> (Y)
```

As we have seen, the value of variable `B` was not affected since the expression in the first clause was not evaluated. This expression could be even not correct and this would be not noticed until the atom not equal `X` were given to the conditional. This can cause, that some errors can get unnoticed for long in a program. Suppose for instance that the clause were:

```
(COND ((ATOM A) (COND ((EQ A 'X) A)
                      (t (SETQ B (CONS A NIL)))  ))
      (t (CAR A)))
```

In such case the error (UNDEFINED VARIABLE B) will be produced only in cases when value of B was not yet assigned, which can depend on the order of reading the data to our program or , in our case, the examples given to the Lisp system! If the last example were read as the first - the error will occur. If it will be read now, when the value of B has already been assigned, there will be no error indication. We will give now some more examples of COND.

```
(SETQ A X) ==> X

(COND ((ATOM A) A)
      (t  (CAR A))) ==> X

(SET 'A '(A B C D)) ==> (A B C D)

(COND ((ATOM A) A)
      (t  (CAR A))) ==> (A B C D)


(COND ((EQ A t) (SETQ B X))
      ((EQ A 'C) (SETQ B Y))
      (t (SETQ B (CONS X Y))))
```

If t is a value of A then the value of variable X is assigned to X. If C is the value of variable Y then the value of variable Y is assigned to B. If an atom different from t or C is the value of A, then variable B obtains the value equal to the value of the expression (CONS X Y). In all the above cases the new value of variable B is returned as the value of the conditional.

```
(COND ((EQ A 'man) (assign_name  NAME))
      ((SETQ VAL  (FUNCTION2 A)))
      ((SETQ VAL2 (FUNCTION3 A))))
```

In the last example if the value of variable A is man then the function assign_name is called with variable NAME as an argument and the value of this function is returned as the value of the conditional. Otherwise the second clause is evaluated. If the value of (FUNCTION2 A) equals NIL then it is assigned to variable VAL and the program evaluates the third clause. If the value of (FUNCTION2 A) equals non-NIL, it is assigned to variable VAL and also returned as the value of this clause (as the last expression in the clause !) and finally returned as the value of COND. Similarly if the third clause is evaluated, if the value of (FUNCTION3 A) equals NIL then NIL is assigned to VAL2 and returned as the value of the conditional.

```
(COND (A B)
      (t S))
```

is equivalent to

```
(COND (A B)
      (S))
```

2.6.2 Logic Predicates

NOT.

```
(NOT S)
```

NOT is a one-argument predicate. An arbitrary S-expression can be the value of the argument. If the argument evaluates to NIL then t is returned as the value of NOT. Otherwise, if the value of the argument is non-NIL, the value NIL is returned by NOT:

```
(NOT 5) ==> NIL
(NOT NIL) ==> *T*
(NOT '(d f)) ==> NIL
(NOT 'NIL) ==> *T*
(NOT f) ==> *T*
(NOT t) ==> NIL
```

AND.

```
(AND <ex1>....<exn>)
NORMAL, SUBR
```

AND is a predicate with an arbitrary number of arguments (special form). To evaluate this predicate its arguments are evaluated from left to right. Whenever an argument evaluates to NIL, NIL is returned as the value of AND:

```
(AND 3 4 5) ==> 5
f3 ==> UNDEFINED VARIABLE f3
(AND 5 NIL (EQ f3 f4)) ==> NIL
```

Let us observe that when the second argument was evaluated to NIL, the next argument was not evaluated and the value NIL was returned immediately. The undefined value in the last predicate was then not observed. Again, this can cause that certain errors are unnoticed for a long period of time.

```
(AND (EQ 'A 'A) (ATOM 'S) (ATOM '(E)) (A . B)) ==> NIL
```

Again, the value of expression (A . B) is not evaluated and an error is not signalized.

```
(AND (EQ (CAR (CONS A B)) A)
        (EQ (CDR (CONS A B)) B)
        (EQ NIL (NOT t))) ==> *T*
```

The value of AND equals *T* since each of its arguments is a logical tautology - expression that is true for all values of inputs.

```
    (AND t  (EQ A (CONS (CAR A) (CDR A))) ==> NIL
```

since the second argument does not evaluate to t, because the cell created by CONS has different address than the cell with the value of A.

```
  (AND 'A (NOT 4)) ==> NIL
```

because (NOT 4) ==> NIL.

OR.

```
(OR <ex1> ... <exn>)
NORMAL, SUBR.
```

OR is a predicate of an arbitrary number of arguments (*special form*). Similarly as in the case of AND, the values of its arguments are evaluated from left to right. When one of the arguments is evaluated to a non-NIL value then the value of this argument is returned as the value of OR. The next arguments are not evaluated. If all arguments evaluate to NIL then NIL is returned as a value of the predicate.

```
(OR 5 6 8) ==> 5
(OR nil F (EQ 'D 'F) (ATOM '(R)) '(A B) '( D F G)) ==> (A B)
(OR (NOT 4) NIL) ==> NIL
(OR NIL f) ==> unbound variable f
(OR 'P 'W) ==> P
```

It is advised to use AND, OR and NOT in conditionals, because this can essentially simplify the descriptions. It is also often very useful to use them alone, outside conditionals:

```
(OR (AND (TRIANGLE A B C) (RECTANGLE A B C D)) (PARALLEL A B C D))
```

2.6.3 Problems

1. Evaluate the expressions

   a) `(COND ((ATOM 'A) 'ATOM) (t NIL))`

   b) `(COND ((ATOM '(A B)) 'ATOM) (t NIL))`

2. Evaluate the expression

```
(COND((ATOM A) (COND((EQ A 'A) A)
                    (t (CONS A 'A))))
     (t (CONS (CAR A) '(A))))
```

   for the following values of variable `A`:

   a) `A`

   b) `(A)`

   c) **B**

   d) `(X Y Z)`

   e) `((A B) C)`

3. What value will be assigned to variable B as the result of evaluating the following expression:

```
(COND ((ATOM A) (COND ((EQ A 'B) (SETQ B 0))
                      ((EQ A 'C) (SETQ B 1))
                      ((EQ A 'D) (SETQ B (CONS A NIL)))
                      (t NIL)))
      (t (SETQ B (COND ((EQ (CAR A) 'A) 2)
                       (t (CAR A) ))))
```

   for the following values of variable `A`:

   a) `X`

   b) `B`

   c) `C`

   d) `(B)`

   e) `(X Y Z)`

   f) `D`

   g) `((A B) D)`

   h) `(A)`

4. Describe evaluation of the expression

```
(COND ((SETQ A (F1)))
      ((SETQ A (F2)))
      ((SETQ A (F3)) (SETQ A (F4)))
      ((SETQ A (F5)))
      ((SETQ B (F6)) T ))
```

2.6.4 Solutions to Problems

1

a) `ATOM`

b) `NIL`

2

a) `A`

b) `(A A)`

c) `(B . A)`

d) `(X A)`

e) `((A B) A)`

3

a) will not obtain new value

b) `0`

c) `1`

d) `B`

e) `X`

f) `(D)`

g) `(A B)`

h) `2`

2.7 Arithmetic Functions and Predicates

In this section we will present Lisp arithmetic functions and predicates. Those functions accept only numbers as their arguments, and return numbers as their values. If the values of all arguments of the given function are the fixed point numbers then the fixed point number is returned as a value as well. If at least one argument is a floating point number then the value of the function is the floating point number as well.

2.7.1 Arithmetic Functions

PLUS
`PLUS` is a function of an arbitrary number of arguments. The arithmetic sum of the arguments is returned as a value:

```
(PLUS 4 5) ==> 9
(PLUS 5 8 13) ==> 26
(PLUS 5 0.1 4) ==> 9.1
```

DIFFERENCE.
`DIFFERENCE` is a two-argument function. Its value is the arithmetic difference of the values of the first and the second argument:

```
(DIFFERENCE 2 1) ==> 1
(DIFFERENCE 1.2 2) ==> - 0.8
```

MINUS.

`MINUS` is a one-argument function. Its value is the value of the argument with the sign reversed:

```
(MINUS 1) ==> -1
(MINUS -0.5) ==> 0.5
```

ADD1.
`ADD1` is a one-argument function. It returns the value of the argument increased by one:

```
(ADD1 1) ==> 2
(ADD1 -0.5) ==> 0.5
```

SUB1.
SUB1 is a one-argument function. It returns the value of the argument decreased by one:

```
(SUB1 1) ==> 0
(SUB1 25Q) ==> 20
(SUB1 -0.5) ==> -1.5
```

MAX.
MAX is a function of arbitrary number of arguments. It returns the maximum of the values of arguments:

```
(MAX 0 1 2.5 3) ==> 3.0
(MAX 0 2 1 3) ==> 3.
```

MIN.
MIN is a function of arbitrary number of arguments. It returns the minimum of the values of arguments:

```
(MIN 0 1 2 3) ==> 0
(MIN 0 1 2.5 3) ==> 0.0
```

TIMES.
TIMES is the function of arbitrary number of arguments. It returns the arithmetic product of the values of all arguments:

```
(TIMES 2 3) ==> 6
(TIMES 0.5 -2 4) ==> -4.0
```

RECIP.
RECIP is a one-argument function. It returns the reciprocal of the value of the argument. Value 0 is returned as the reciprocal of integers:

```
(RECIP 2) ==> 0
(RECIP 1) ==> 0
(RECIP 2.0) ==> 0.5
```

QUOTIENT.
QUOTIENT is a two-argument function. It returns the quotient of the value of the first by the value of the second argument. If the values of both arguments are fixed-point numbers then their quotient is rounded to the nearest integer less than this quotient. An attempt to divide by zero and the overflow are signalized as errors:

```
(QUOTIENT 2.1 3) ==> 0.7
(QUOTIENT 2 3) ==> 0
(QUOTIENT 3 2) ==> 1
```

REMAINDER.

```
(REMAINDER NUMBER1 NUMBER2)
NORMAL, SUBR.
```

REMAINDER is a two-argument function. Its value is the remainder from division of two integers, i.e. number1 / number2. The remainder is calculated by the formula

```
REMAINDER = DIVIDEND - (QUOTIENT * DIVISOR)
```

where QUOTIENT is the value of function QUOTIENT.

```
(REMAINDER 3 2) ==> 1
(REMAINDER 2 3) ==> 2
(REMAINDER 2.1 3) ==> 0.0
(REMAINDER -3 2) ==> -1
(REMAINDER 3 -2) ==> 1
```

DIVIDE.

```
(DIVIDE NUMBER1 NUMBER2)
NORMAL, SUBR.
```

DIVIDE is a two-argument function. If its arguments were denoted by A and B then the value of DIVIDE equals

```
(CONS (QUOTIENT  A B) (CONS REMAINDER A B) NIL))
```

The value of DIVIDE is then the list of values of functions QUOTIENT and REMAINDER applied to the same arguments:

```
(DIVIDE 3 2) ==> (1 1)
(DIVIDE 2 3) ==> (0 2)
(DIVIDE 2.1 3) ==> (0.7 0.0)
```

LOGAND.

```
(LOGAND NUMBER NUMBER ... NUMBER)
NORMAL, FSUBR.
```

LOGAND evaluates all its arguments and returns their bit-by-bit logical product (logical AND). Each number is treated as a 60-bit quantity (implementation dependent). The value of LOGAND is always of octal type.

```
(LOGAND 2 3 4) ==> 0Q
```

LOGOR.

```
(LOGOR NUMBER NUMBER ... NUMBER )
NORMAL, FSUBR.
```

LOGOR evaluates all its arguments and returns their bit-by-bit logical sum (logical OR). Each number is treated as a 60-bit quantity (implementation dependent). The value of LOGAND is always of octal type.

```
(LOGOR 2 3 4) ==> 7Q
```

LOGXOR.

```
(LOGXOR NUMBER NUMBER ... NUMBER )
NORMAL, FSUBR.
```

LOGOR evaluates all its arguments and returns their bit-by-bit logical difference (exclusive-OR) with associating to the left. Each number is treated as a 60-bit quantity (implementation dependent). The value of LOGAND is always of octal type.

```
(SETQ E 3) ==> 3
(SETQ D 4) ==> 4
(LOGXOR E D) ==> 7Q
```

LEFTSHIFT.

```
(LEFTSHIFT NUMBER FIXNUMBER)
NORMAL, SUBR.
```

LEFTSHIFT performs a shifting operation on its 60-bit first argument. fixnumber is a shift count of the number of bits number is to be shifted. If the shift count is positive, the shift is left, end-around circular. If it is negative, then the shift is right, end-off with sign extension. The value of LEFTSHIFT is always of octal type.

```
(SETQ E 3) ==> 3
(SETQ D 2) ==> 2
(LETSHIFT E D) ==> 14Q
(LETSHIFT 3 2) ==> 14Q
(LETSHIFT 2 -2) ==> 0Q
(LETSHIFT 1 4) ==> 2Q3
```

FIX.

```
(FIX FLNNUMBER)
NORMAL, SUBR.
```

The value of `FIX` is the largest INTEGER contained in the floating point number. The value of `FIX` is always of integer type.

```
(FIX 2.3) ==> 2
```

FLOAT.

```
(FLOAT FIXNUMBER)
NORMAL, SUBR.
```

`FLOAT` returns a floating-point number whose value is the same as that of the fixed-point argument. `FIXNUMBER` must be less than 2^{48} in magnitude for this function to give the proper result.

```
(FLOAT 2) ==> 2.00 000
```

OCTAL.

```
(OCTAL NUMBER)
PSEUDOFUNCTION, SUBR.
```

`OCTAL` converts its argument into a number of octal type. The print image of the resulting number is the octal representation of the original value. No conversion from floating-point to fixed-point is made. `OCTAL` actually modifies its argument directly. It does not make a copy of its argument.

```
(OCTAL 1.0) ==> 17204Q45
(OCTAL 0.0) ==> 0Q
(OCTAL 2) ==> 2Q
(OCTAL 22) ==> 26Q
```

RANDOM.

```
(RANDOM NUMBER)
NORMAL, SUBR.
```

`RANDOM` returns a new random number in the range 0-1 each time it is called with a zero argument. If called with an argument between 0 and 1, it will return that argument as its value and on subsequent calls with a 0 argument it will return new random numbers belonging to a new sequence begun by the call with a non-zero argument.

Below you can find my favourite books on Lisp and functional programming. They include interesting programming techniques that you can use in your programs. Some of these books are actually quite old, but still very good. Look for them in second hand bookstores or in libraries.

2.7.2 Arithmetic Predicates

Similarly as the arithmetic functions, the arithmetic predicates accept numbers as the values of the arguments. The predicate NUMBERP is an exception.

NUMBERP.

```
(NUMBERP S)
PREDICATE, SUBR.
```

NUMBERP is a one-argument function. It expects the value of its argument to be an atom. If it is a number (of arbitrary type) then *T* is the value of the predicate, else NIL is returned:

```
(NUMBERP 2) ==> *T*
(NUMBERP 'A) ==> NIL
(NUMBERP 2.3) ==>*T*
(NUMBERP 2 . 3) ==> ERROR: INCORRECT NUMBER  OF ARGUMENTS.
(NUMBERP '(W ER)) ==> ?
```

ZEROP.

```
(ZERORP NUMBER)
PREDICATE, SUBR.
```

ZEROP is a one-argument predicate. This predicate is satisfied if its argument evaluates to 0. It returns also t if the absolute value of its argument is less than $3 * 10^{-6}$.

```
(ZEROP 0) ==> *T*
(ZEROP 3e-20) ==> NIL
(ZEROP 1) ==> NIL
(ZEROP 3e-100) ==> *T*
```

MINUSP.

```
(MINUSP NUMBER)
PREDICATE, SUBR.
```

MINUSP is a one-argument predicate. It is satisfied when its argument evaluates to a negative number. Remember, that -0 is a negative number, while 0 is a positive number.

```
(MINUSP 5) ==> NIL
(MINUSP -4.52) ==> *T*
(MINUSP 0) ==> NIL ; (depends on implementation)
(MINUSP -0) ==> -0 ; (depends on implementation)
```

ONEP.

```
(ONEP NUMBER)
PREDICATE, SUBR.
```

ONEP is a one-argument predicate. It is satisfied if its argument evaluates to 1.

```
(ONEP 1) ==> *T*
(ONEP 1.0) ==> *T*
(ONEP 1.1) ==> NIL
```

FIXP.

```
(FIXP NUMBER)
PREDICATE, SUBR.
```

FIXP is a one-argument predicate. It is satisfied if the value of its argument is an integer:

```
(FIXP 5) ==> *T*
(FIXP 5.0) ==> NIL
```

FLOATP.

```
(FLOATP NUMBER)
PREDICATE, SUBR.
```

FLOATP is a one-argument predicate. It returns *T* when its argument evaluates to a floating-point number.

```
(FLOATP 5) ==> NIL
(FLOATP 5.0) ==> *T*
```

LESSP.

```
(LESSP NUMBER1 NUMBER2)
PREDICATE, SUBR.
```

LESSP is a two-argument predicate. It is satisfied when the value of the first argument is greater than the value of the second argument.

```
(LESSP 5 6) ==> *T*
(LESSP 5.4 5) ==> NIL
```

GREATERP.

```
(GREATERP NUMBER1 NUMBER2)
PREDICATE, SUBR.
```

```
(GREATERP 5 6) ==> NIL
(GREATERP 5.4 5) ==> *T*
```

EQN.

```
(EQN S1 S2)
PREDICATE, SUBR.
```

EQN is a two-argument predicate. Arbitrary S-expressions can be the values of its arguments. The predicate is satisfied if the arguments are EQ or are two numeric atoms with the same numeric value. The numbers can be of arbitrary type (the numbers with the difference less than $3 * 10^{-6}$ are signalized as equal).

```
(EQN 'A 'A) ==> *T*
(EQN '(A B) '(A B)) ==> NIL
(EQN 5 3) ==> NIL
(EQN 5 5) ==> *T*
(EQN 5 5.0) ==> *T*
(EQN 5 5.0000000001) ==> *T*
```

2.7.3 Composition

We will give now some examples, how to use the functions presented above.

Example 2.1

Write the Lisp functional expression which executes the following operation:
`2 * A + B / C - (A + 1) / 4`
on the values of variables `A`, `B`, and `C`.

Starting to solve this problem we select the operations that can be used as the first one. These are the operations: `2 * A`, `B / C`, and `A + 1`. These operations can be written in Lisp as:
`(TIMES 2 A)`, `(QUOTIENT B C)` and `(ADD1 A)`.

Next the division shall be applied, which is described in the form: `(QUOTIENT (ADD1 A) 4)`. Now we can describe the operations of addition and difference in arbitrary order, i.e.

```
(DIFFERENCE (PLUS (TIMES 2 A) (QUOTIENT B C))
            (QUOTIENT (ADD1 A) 4))
```

or

```
(PLUS (TIMES 2 A) (DIFFERENCE (QUOTIENT B C)
                      (QUOTIENT (ADD1 A) 4)))
```

Example 2.2

Write in Lisp the functional expression whose execution is done as follows:

- if the even natural number is the value of `A` then assign the value of the expression `C + A` as the value of variable `B`,
- if the value of variable `A` is the odd fixed-point number then assign the remainder from the division of this number by 5 as the value of variable `B`,
- if the floating-point number is the value of variable `A` then assign the value of the expression `C * (A - 1) + (A - 2)` as the value of variable `B`,
- if the value of variable `A` is not a number then do nothing.

The possible solution to this problem is the following functional expression.

```
(COND ((NUMBERP A)
       (SETQ B (COND
                 ((FIXP A) (COND ((ZEROP (REMAINDER A 2)) (PLUS C A))
                                 (T (REMAINDER A 5))))
                 ((FLOATP A) (PLUS (TIMES C (SUB1 A)) (DIFFERENCE A 2)))
                 (T NIL))))
      (T NIL))
```

Recognition of even numbers is done by checking if the remainder from the division by two equals zero. This checking is preceded by checking if the given number is the fixed-point number. The third expression in the second conditional ensures that this expression has the respective value when the value of the variable `A` is an octal number.

Example 2.3

Write in Lisp the functional expression whose behavior is the following:
If `A = 1, B = 1, C = 0` then add the values of `X` and `Y` and return their sum as the value of the functional expression.
If `A = 1, B = 0, C = 1` then the value of the functional expression equals the difference of the values of `X` and `Y`.
If `A = 0, B = 0` then this value equals `X * Y`.
If `A = 0, B = 1` then this value equals `X / Y`.
If `A = 1, B = 1, C = 1` then this value equals `X / 2`.

We assume that `A, B, C` = 0,1. The solution is the following functional expression:

```
(COND ((AND (ONEP A) (ONEP B) (ZEROP C)) (PLUS X Y))
      ((AND (ONEP A) (ZEROP B) (ONEP C)) (DIFFERENCE X Y))
      ((AND (ZEROP A) (ZEROP B)) (TIMES X Y))
      ((AND (ZEROP A) (ONEP B)) (QUOTIENT X Y))
      ((AND (ONEP A) (ONEP B) (ONEP C)) (QUOTIENT X 2))
      (T NIL))
```

The solution can be written in a simpler form:

```
(COND ((AND (ONEP A) (ONEP B) (ZEROP C)) (TIMES X Y))
      ((ZEROP A) (QUOTIENT X Y))
      ((AND (ONEP B) (ONEP C)) (QUOTIENT X 2))
      ((ONEP B) (PLUS X Y))
      ((ONEP C) (DIFFERENCE X Y))
      (T NIL))
```

Let us note that through respective placement of the factors inside `COND` the total number of predicates in the expression has been reduced from 18 to 9.

2.7.4 Problems

1. Write functional expressions which realize the operations

   a) `A + B + C + D`

   b) `A - B * C * D`

   c) `X / Y + Z`

   d) `- (A + B) + A * B`

   e) `(A + B) * (B - C) * (A + 2 * B)`

2. Write a functional expression that realizes the following operations:

   a) selection of the maximum absolute value among the variables `A, B, C` .

   b) calculation of the product of the maximal and minimal of the numbers being the values of variables `A, B, C, D` .

3. Write a functional expression that realizes the following operations:

   a) If the symbolic atom is the value of variable `A` then it is returned as the value of the function. If the value of `A` is the number then this number increased by one is returned for positive numbers and the number decreased by one for negative numbers. If the value of the variable `A` is a list or a dotted pair then the first element of the list or pair is returned as the value.

   b) If numbers are the values of variables `A` and `B` then as a value of variable `A` assigned is:

   `A + B` if 1 is the value of `C`,

   `A - B` if -1 is the value of `C`,

   `A * B` if 2 is the value of `C`,

   `A / B` if -2 is the value of `C`,

   `- A` if 0 is the value of `C`,

   `A + 1` for all other values of `C`.

   If only one of the values of `A` and `B` is a number, then this number is assigned as the value of variable `A`. If none of the values of variables is a number then assign zero as the value of variable `A`.

2.7.5 Solutions to Problems

1

a) `(PLUS A B C D)`

b) `(DIFFERENCE A (TIMES B C D))`

c) `(PLUS (QUOTIENT X Y) Z)`

d) `(PLUS (MINUS (PLUS A B)) (TIMES A B))`

e) `(TIMES (PLUS A B) (DIFFERENCE B C) (PLUS A (TIMES 2 B)))`

2

a)
```
(MAX (COND ((MINUSP A) (MINUS A)) (T A))
     (COND ((MINUSP B) (MINUS B)) (T B))
     (COND ((MINUSP C) (MINUS C)) (T C)))
```

b)
```
(TIMES (MAX A B C D) (MIN A B C D))
```

3

a)
```
(COND ((ATOM A) (COND ((NUMBERP A) (COND ((MINUSP A) (SUB1 A))
                                         (T (ADD1 A))))
                      (T A)))
      (T (CAR A)))
```

b)
```
(COND ((AND (ATOM A) (ATOM B))
       (COND ((AND (NUMBERP A) (NUMBERP B))
              (SETQ A (COND ((ONEP C) (PLUS A B))
                            ((EQN C -1) (DIFFERENCE  A B))
                            ((EQN C 2) (TIMES  A B))
                            ((EQN C -2) (QUOTIENT  A B))
                            ((ZEROP  C) (MINUS  A))
                            (T (ADD1 A)) )))
             ((NUMBERP B) (SETQ A B))
             ((NUMBERP A)  NIL)
             (T (SETQ A 0))))
      (T NIL))
```

Chapter 3

More List Processing, Representations and Processing of Discrete Data in Visual Hardware Lisp.

3.1 Functions and Predicates Processing List Structures

In this section we will present some of the most commonly used functions and predicates that check, compare and process s-expressions.

3.1.1 Predicate EQUAL

```
(EQUAL S1 S2)
(=     S1 S2)
PREDICATE, SUBR.
```

Predicate `EQUAL` has two arguments that evaluate to arbitrary S-expressions. `EQUAL` returns *T* if the evaluated values are identical expressions (in the sense that they would look the same when printed) and NIL otherwise. Two expressions are identical if their printouts are identical, which means, they can be transformed to identical structures through equivalent transformations from the list notation to the dot notation. The expressions are identical when the list structures corresponding to them are one-to-one mappings if each of them were converted to a tree by copying all shared nodes so that no more shared nodes will exist any more. For instance the S-expressions:

```
X1 ==> ((A B) C (D E))
```

and

```
X2 ==> ((A . (B . NIL)) . (C . ((D . (E . NIL)) . NIL)))
```

are identical because they are only different descriptions of the same structure. The reader is recommended to check identity of these two structures using three methods:

- type `(EQUAL X1 X2)` to the computer,
- assign respective values to variables `X1` and `X2` and compare printouts of their values,
- use the previously described rules for conversion.

The behavior of `EQUAL` is the following. First it is checked if the pointers to the values of arguments point to the same call or if they point to cells with the same numerical values. If any of these conditions is satisfied, *T* is returned as the function's value. If at least one of the pointers points to a cell subordinated to an atom and the previously specified conditions are not satisfied, then NIL is returned. In the remaining cases the above checking is done for the pointers coming respectively from `CAR` and `CDR` parts of the cells subordinated to the arguments, which can be formulated as the evaluation of the functional expression:

```
(AND (EQUAL (CAR A1) (CAR A2)) (EQUAL (CDR A1) (CDR A2)))
```

where A1 and A2 are the arguments of the initial functional expression (EQUAL A1 A2).

As we see, EQUAL is a *recursive function*, which in some conditions *calls itself* (is *self-referencing*). The simplified definition of EQUAL, together with the detailed trace of its execution will be given in 3.3.

Examples:

```
(EQUAL 2 2.0) ==> *T*
(EQUAL 'A 'A) ==> *T*
(EQUAL 'A 'B) ==> NIL
(EQUAL '(A B C) '(A B C)) ==> *T*
(EQUAL '(A B C) '(A . (B . (C . NIL)))) ==> *T*
(EQUAL '(A B) '(B A)) ==> NIL
```

3.1.2 Predicate NULL

```
(NULL S)
PREDICATE, SUBR.
```

NULL is a one-argument predicate. Its argument can evaluate to an arbitrary S-expression. If it evaluates to NIL then *T* is returned. Otherwise NIL is returned. NULL is usually used to detect empty lists. Let us note that functionally, NULL behaves exactly as NOT.

```
(NULL NIL) ==> *T*
(NULL 2) ==> NIL
(NULL (CDR '(A))) ==> *T*
(NULL (CAR '(A))) ==> NIL
```

3.1.3 Predicates MEMBER and MEMQ

```
(MEMBER S1 S2)
PREDICATE, SUBR.
```

MEMBER is a two-argument predicate. Both arguments evaluate to S-expressions. If the value of the first argument equals to one of the top-level elements of list being the second argument, then the part of the list starting from this element is returned. Otherwise NIL is a value of MEMBER. This function has then two applications:

- it is a predicate used to check if some S-expression is a member of the list of S-expressions,
- it can serve to find the tail of the list (or general S-expression) that begin from certain S-expression.

Examples.

```
(MEMBER 'A '(B A C)) ==> (A C)
(MEMBER 'A '(B A . (C . D)) ) ==> (A . ( C  . D))
(MEMBER 'A '(B C . (A . D)) ) ==> NIL
(MEMBER '(A B) '( (E R)  (R T)  (A B) ) ) ==> ((A B))
```

```
(MEMQ S1 S2)
PREDICATE, SUBR.
```

Like MEMBER but uses EQ instead of EQUAL.

3.1.4 Function LIST

```
(LIST S1 S2 ... SN)
NORMAL, FSUBR.
```

LIST is a special form with an arbitrary number of arguments. It returns a list of values of arguments. These values can be arbitrary S-expressions:

```
(LIST 'A 2 '( A B)) ==> (A 2 (A B))
(LIST 'A ==> (A)
(LIST  'A newAtom  'B) ==> UNASSIGNED VALUE newAtom
(LIST (CAR '(X Z)) 'S) ==> (X S)
```

3.1.5 Function APPEND

```
(APPEND LIST S)
NORMAL, SUBR.
```

APPEND is a two-argument function with lists as values of both arguments. The function returns the list whose elements are the successive elements of the two lists being the values of the first and the second argument:

```
(APPEND '( A B )  '( C D ) ) ==> (A B C D)
(APPEND '(A) (LIST 'A)) ==> (A A)
(APPEND (CDR '((Luis Veronica))) neighbors)
         ==> (Veronica Jane Mary Elizabeth)
```

3.1.6 Function PAIRLIS

```
(PAIR LIST1 LIST2 S)
NORMAL, SUBR.
```

PAIRLIS is a three-argument function. The arguments can evaluate to arbitrary lists, but the lists being the values of the first two arguments must include the same number of elements. PAIRLIS returns a list of dotted pairs, which pairs elements from the first list with the respective elements from the second list. The value of the third argument is appended at the end of the such created list:

```
(PAIRLIS '(A B) '(X Y) '(V Z)) ==> ( (A . X) (B . Y) V Z)
(PAIRLIS neighbors '(John Steve Felix) NIL) ==>
     ((Jane . John) (Mary . Steve) (Elizabeth . Felix))
```

3.1.7 Function ASSOC

```
(ASSOC S1 S2)
NORMAL, EXPR.
```

ASSOC is a two-argument function. The value of the first argument is an arbitrary S-expression. The list of dotted pairs is the value of the second argument. ASSOC returns first pair in this list, that has a CAR part equal to the value of the first argument. NIL is returned if such a pair does not exist:

```
(ASSOC 'A '((B . C) (A . A) (A . B))) ==> (A . A)
(ASSOC 'A '((B . A) (C . A) (AA . A))) ==> NIL
(ASSOC '(A N) '(((M B) . x1) ((A N) . x2) ((S D) . x4)))
                       ==> ((A N) . x2)
(ASSOC ' (A N) '(((C V)) ((A N)) ((S D))) ==> ((A N))
(ASSOC 'MacIntosh
  '((MacIntosh Compac IBMPC) (Apple2 Commodore)) ==>
       (MacIntosh  Compac IBMPC)
(ASSOC 'A '((A.D))) ==> NIL
(ASSOC 'A ((A . D))) ==> (A . D)
```

3.1.8 Function RPLACA

The list processing functions discussed thus far are limited to selecting some substructures (like CAR, CDR, ASSOC) or creating new structures by utilizing new memory cells (CONS, LIST, APPEND). In order to speed up list transformations we use some functions that change and modify the already existing structures.

```
(RPLACA NATS S)
PSEUDOFUNCTION, SUBR.
```

RPLACA is a two-argument pseudo-function. The first argument evaluates to an S-expression being not an atom and the second argument evaluates to an arbitrary S-expression. The modification consists in replacing the pointer outcoming from the CAR part of the cell subordinated to the value of the first argument with the pointer to the value of the second argument. Application of RPLACA is much more efficient than usage of a function that conses arguments. Let us consider for instance an expression:

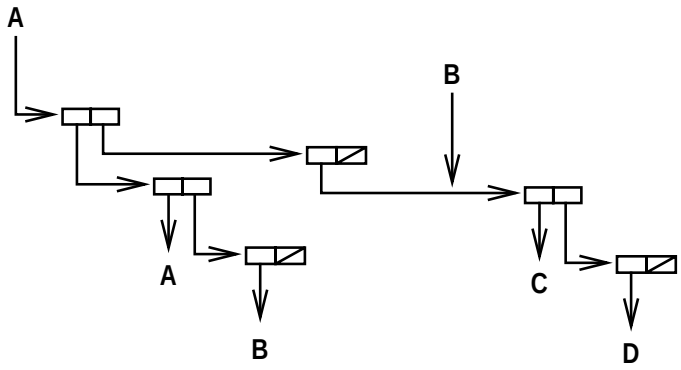


Figure 3.1: The initial state of the memory

```
(RPLACA A B)                                                    (3.1)
```

where the value of variable `A` is the S-expression `((A B) (C D))` and S-expression is the value of `B`. Let us also consider for comparison the following expression:

```
(SETQ A (CONS B (CDR A)))                                       (3.2)
```

As the result of evaluation of each of above expressions variable `A` is assigned a value

```
((C D) (C D))
```

and value `B` remains unchanged.

Another example:

```
A ==> (A . D)
B ==> (A . D)
(RPLACA A '(A . D)) ==> ((A . D) . D)
A ==> ((A . D) . D)

(SETQ X '(A . D)) ==> (A . D)
(SETQ  X (CONS B (CDR A)) ==> ((A . D) . D)
X ==> ((A . D) . D)
```

All the remarks about application of `RPLACA` hold also true for `RPLACD`.
To compare to the operation of previously known functions, let us investigate what transformations are executed by each of the functions. The initial state of the memory is schematically shown in Fig. 3.1.

The next transformations, executed by evaluating the functional expression 3.2 are presented in Figs. 3.2, 3.3, and 3.4. Fig. 3.5 presents the transformation executed by the evaluation of the functional expression 3.1.

Comparing Figs. 3.4 and 3.5 we can notice that in the first case a *new memory cell* is subordinated to variable `A` and in the second case only the *contents* of the cell subordinated to `A` is modified. Using function `RPLACA` we spare one memory cell and we spare a time necessary to access this cell, obtaining the same efeect. `RPLACA` is faster and uses less memory, but we must use it with caution and be always aware what transformation of structure it causes. For full clarity we will present the computer implementation of the discussed transformations as well. Let us assume that the cell of address 610 is subordinated to variable `A` and the cell of address 614 to variable `B`. The implementation before the transformations is:

```
610      611    612
611       A     613
612      614    NIL
613       B     NIL
614       C     615
615       B     NIL
```

a)

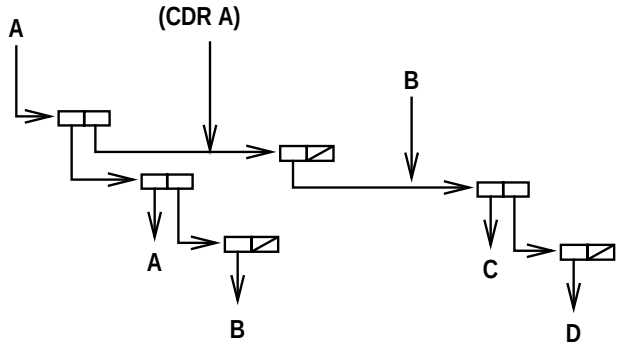


Figure 3.2: The transformations executed by evaluating the functional expression

b)

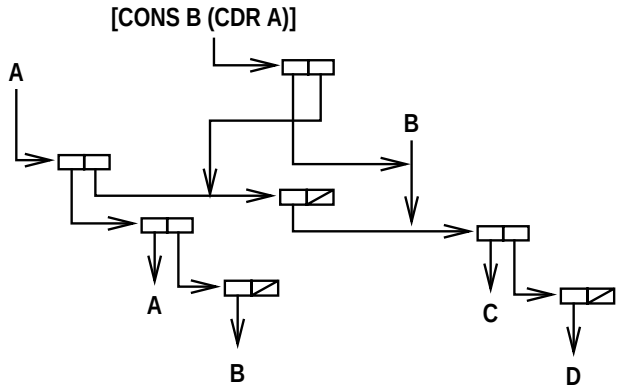


Figure 3.3: The transformations executed by evaluating the functional expression

c)

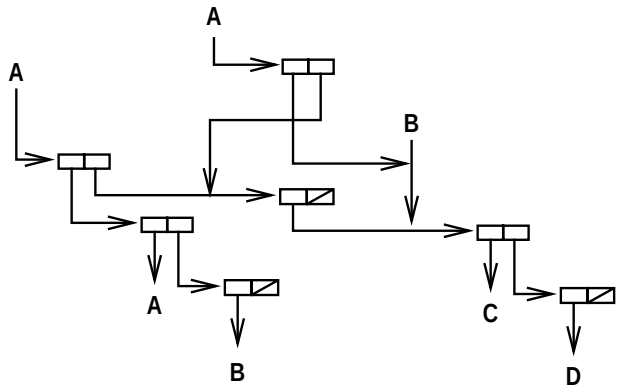


Figure 3.4: The transformations executed by evaluating the functional expression

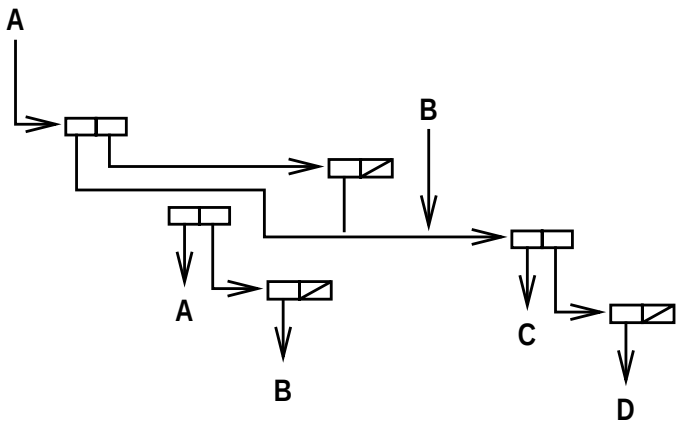


Figure 3.5: The transformations executed by evaluating the functional expression

The contents of the cells of addresses 610-614 do not change after evaluating the expression 3.2 and the new cell of address 616 is subordinated to variable `A`:

```
616    614   612
```

In a case, when the value of the functional expression 3.1 is calculated - only the contents of the cell 610 is modified but it still remains subordinated to the value of `A`

```
610   614    612
```

Let us observe, that some cells remain in the memory after transformations, such that no pointers would lead to them, i.e. the cells that are *not accessible*. In the first case this is the cell of address 610 and the connected to it cells of addresses 611 and 613. In the second case - the cell of address 611 and the connected to it cell of address 613. The interpreters of Lisp have some means to recollect and use such cells again.

Let us consider more examples of `RPLACA`. Assume that the value of variable `A` is

```
((A B) C)
```

and the value of variable `B` is

```
(X Y Z)
```

Let the computer implementation of these values be:

```
650      651   652
651       A    653
652       C    NIL
653       B    NIL
720       X    721
721       Y    722
722       Z    NIL
```

The cell of address 650 corresponds to variable `A` and the cell of address 720 to variable `B`. The state of the memory is schematically presented in Fig. 3.6. Let us consider four examples:

a) `(RPLACA  B  (CDR  A))`

b) `(RPLACA  (CDR  B)  A)`

c) `(RPLACA  (CDDR  B)  B)`

d) `(RPLACA  B  B)`

Each of these examples will be considered separately, i.e. in each of them the initial values of variables are the same. As the result of evaluating the above functional expressions we obtain the following structures:

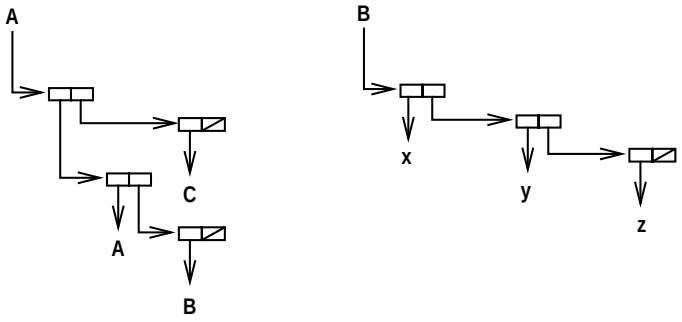


Figure 3.6: The state of the memory presented schematically

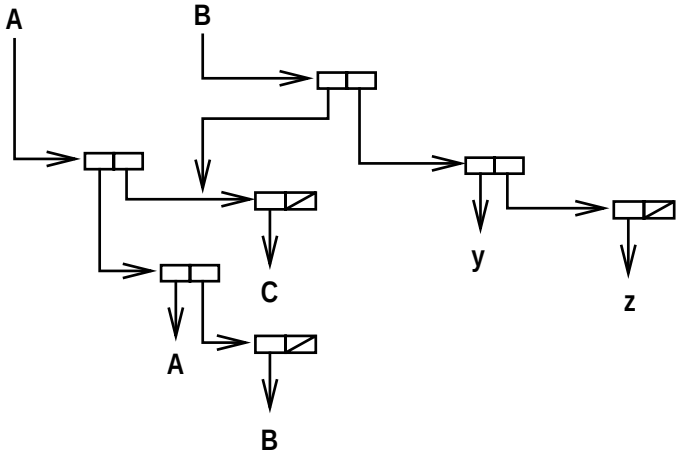


Figure 3.7: Structure presented schematically

a)
```
650     651   652
651      A    653
652      C    NIL
653      B    NIL
720     652   721
721      Y    722
722      Z    NIL
```

b)
```
650     651    652
651      A     653
652      C     NIL
653      B     NIL
720      X     721
721     652    722
722      Z     NIL
```

c)
```
720      X      721
721      Y      722
722     720     NIL
```

d)
```
720     720     721
721      Y      722
722      Z      NIL
```

These structures have been presented in Figs. 3.7, 3.8, 3.9 and 3.10.

Observe, that in the last two examples certain *looped* (called also, *circular*) structures have been created, which cannot be presented in a form of finite S-expressions (and hence cannot be printed). Let us try to print:

```
(RPLACA -> BBBBBBB.... (infinite sequence of symbols B)
```

The structures of this type can occur and exist only inside the computer memory. A big care is advised when using `RPLACA`. Its improper use can cause many difficulties, up to destroying the contents of the entire memory.

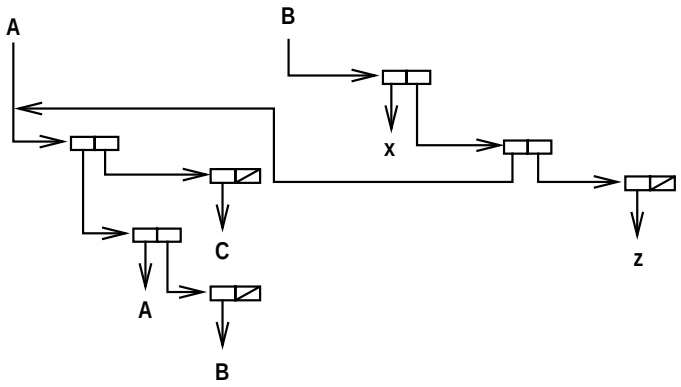


Figure 3.8: Structure presented schematically WRONG?.

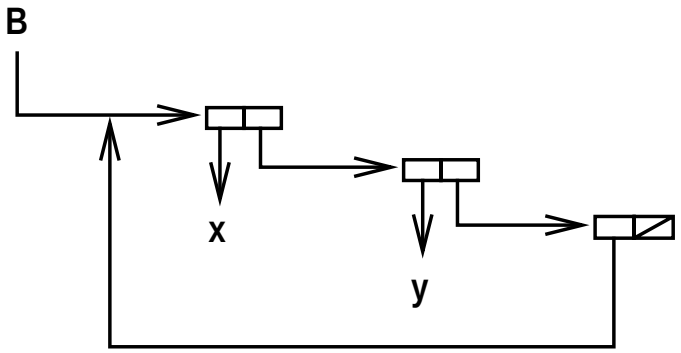


Figure 3.9: Structure presented schematically.

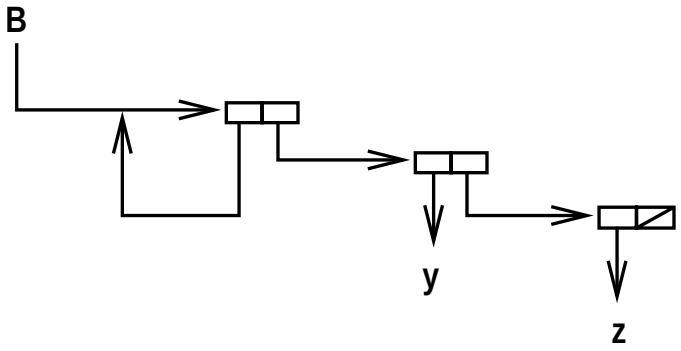


Figure 3.10: Structure presented schematically.

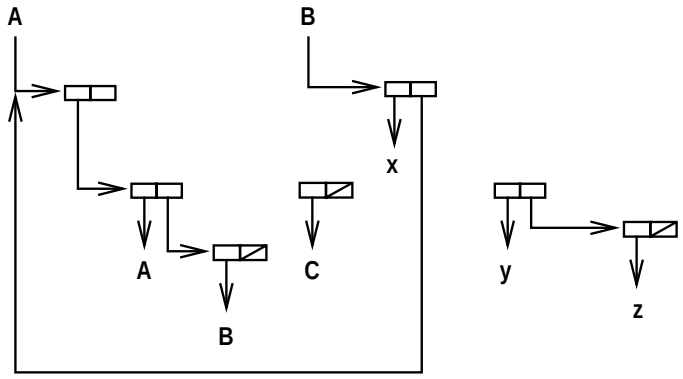


Figure 3.11: Structure shown schematically.

3.1.9 Function RPLACD

RPLACD.

```
(RPLACD NATS S)
PSEUDOFUNCTION, SUBR.
```

`RPLACD` is a pseudofunction of two arguments. The first argument evaluates to an arbitrary S-expression but not an atom and the second argument evaluates to arbitrary S-expression. `RPLACD` returns the value of the first, modified argument. The modification consists in replacing the pointer coming out from the `CDR` part of the cell corresponding to the first argument, with the pointer to the cell corresponding to the value of the second argument. `RPLACD` has similar properties to `RPLACA`. Its behavior can be described as follows:

```
(SETQ A (CONS (CAR A) B))
```

The reader is asked to verify the value of A for both cases on the computer. All the remarks about application of `RPLACA` hold also true for `RPLACD`.

Example:

```
(RPLACD A (RPLACD B A))
```

where the values of `A` and `B` are the same as in subsection 3.1.8. (Fig. 3.6).
After transformation the structure looks as follows:

```
650         651      720
651          A       653
652          C       NIL
653          B       NIL
720          X       650
721          Y       722
722          Z       NIL
```

This structure is shown schematically in Fig. 3.11.
Observe, that the values of both variables are circular structures.

3.1.10 Problems

1. Assuming that `A ==> (A  B  (C  D))` and `B ==> (B  Y  (C  D))`, evaluate values of the following expressions:

   a) `(EQUAL A B)`

   b) `(EQUAL (CAR A) (CAR B))`

   c) `(EQUAL (CDDR A) (CDDR B))`

   d) `(EQUAL (CAR A) (CADR B))`

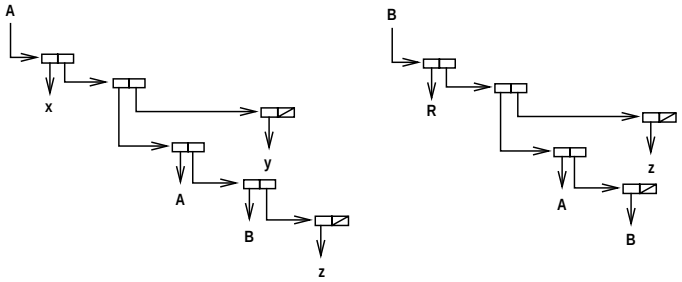


Figure 3.12: Structure to problem 2.

e) `(EQUAL (CADR A) (CAR B))`
f) `(NULL (CAR A))`
g) `(NULL (CDDR A))`
h) `(NULL (CDDDR A))`
i) `(NULL (CDDADDR A))`
j) `(MEMBER (CAR B) A)`
k) `(MEMBER (CAR A) B)`
l) `(MEMBER (CDDR A) B)`
m) `(MEMBER (CADDR A) B)`
n) `(LIST A B)`
o) `(APPEND A B)`
p) `(LIST A (CADDR B))`
r) `(APPEND A (CADDR B))`
s) `(LIST (CAR A) (CADR A) (CAR B) (CADR B))`
t) `(PAIRLIS A B NIL)`
u) `(ASSOC (CAR A) (PAIRLIS A B NIL))`
w) `(ASSOC (CAR B) (PAIRLIS A B NIL))`
z) `(ASSOC (CADR B) (PAIRLIS A B NIL))`

2. Assuming that variables `A` and `B` have respectively values:
`(X (A B Z) Y)` and `(R (A B) Z)` (see Fig. 3.12)
evaluate the values of the following functional expressions and the final values of variables `A` and `B`:

a) `(RPLACA A B)`
b) `(RPLACD A B)`
c) `(RPLACA (CADR A) (CDDR B))`
d) `(RPLACD (CADR A) (CDDR B))`
e) `(RPLACA (RPLACD (CADR A) (RPLACA (CDDR B) (CAR B))) (RPLACD A (CADDR A)))`

Illustrate solutions graphically. Check on your computer.

3.1.11 Solutions to Problems

1.

```
a) NIL     b) NIL    c) *T*   d) NIL       e) *T*        f) NIL      g) NIL
h) *T*     i) *T*    j) *T*   k) NIL       l)  NIL       m) *T*
n) ((A B (C D)) (B Y (C D)))  o) (A B (C D) B Y (C D))
p)  ((A B (C D)) (C D))       r) (A B (C D) C D)
s)  (A B B Y)                 t)  ((A . B) (B . Y) ((C D) . (C D)))
u)  (A . B)                   w) (B . Y)                z) NIL
```

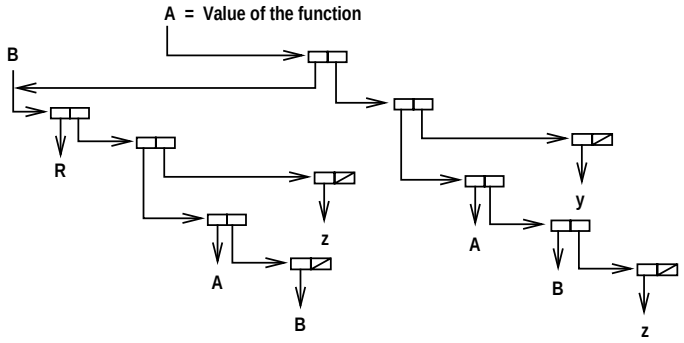


Figure 3.13: Structure to problem 2a.

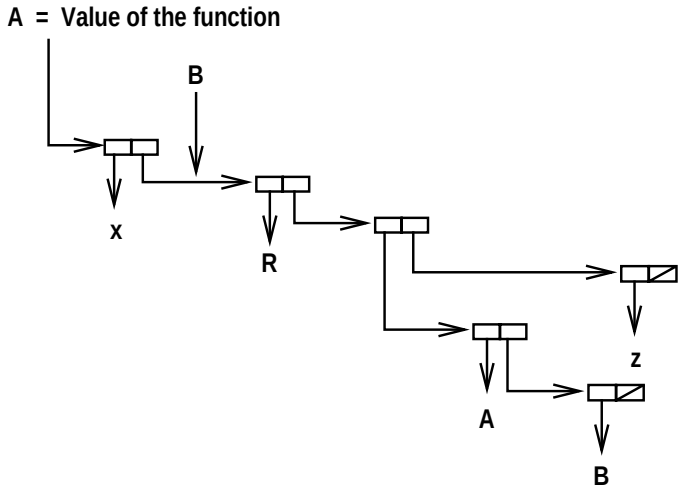


Figure 3.14: Structure to problem 2b.

2. a)

```
((R (A B) Z) (A B Z) Y), A ==> ((R (A B) Z) (A B Z) Y),
    B ==> (R (A B) Z) ,
```

see Figure 3.13.
b)

```
(X R (A B) Z) ,   A ==> (R Y (A B) Z)  ,   B ==> (R (A B) Z) ,
```

see Fig. 3.14.
c)

```
((Z) B Z)  ,  A ==> (X ((Z) B Z) Y) ,  B ==> (R (A B) Z)   ,
```

see Fig. 3.15.
d)

```
(A Z) , A ==> (X (A Z) Y) ,   B ==> (R (A B) Z)  ,
```

see Fig. 3.16.
e)

```
((X Y) Y)  ,  A ==> (X Y) ,  B ==> (Y (A B) Y)   ,
```

see Fig. 3.17

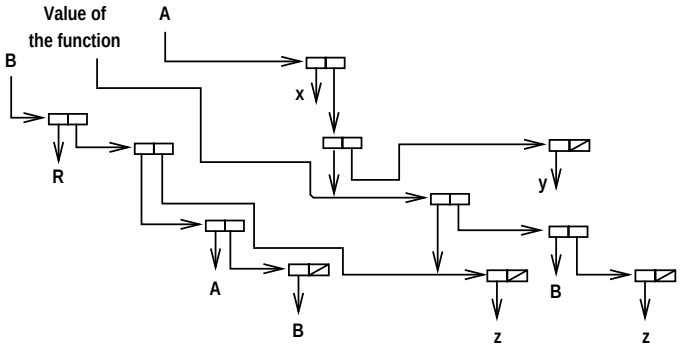


Figure 3.15: Structure to problem 2c.

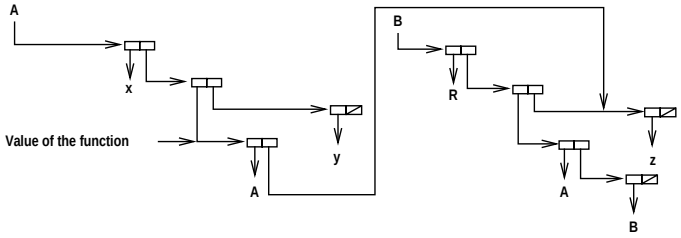


Figure 3.16: Structure to problem 2d.

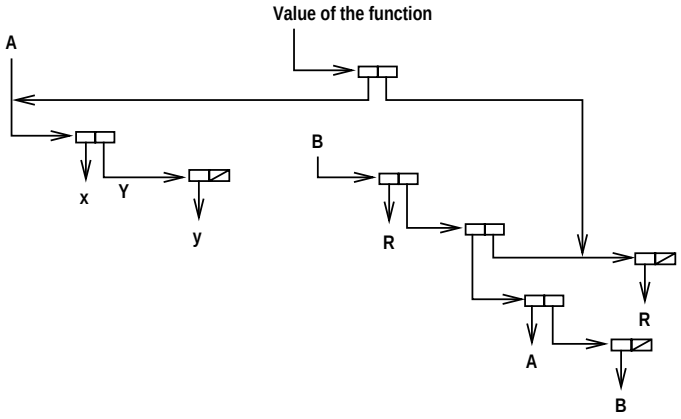


Figure 3.17: Structure to problem 2e.

3.2 Lambda Notation

It will be now necessary to introduce a new notation, called λ-notation, for defining new functions and for constructing more complex functional expressions. The founder of this notation was Alonzo Church. By writing programs we often use expressions like $(y^2 + x)$ whose values are calculated for the given values of arguments. The arguments, whose values are to be substituted for the variables from expression will directly follow the functional expression. Let us also observe, that in the expression $(x + y)$ the notation $((x + y)\ 5\ 3)$ treated as the expression $(x + y)$ with values of arguments 5 and 3, respectively, is complete: the value of the expression is always 8 and it does not depend on the order of assigning numbers 5 and 3 to its arguments, x and y. For the expression $(y^2 + x)$ the notation $((y^2 + x)\ 5\ 3)$ is not complete, because we don't know the order of assignment of assignment of 5 and 3 to `x` and `y`. The complete notation must then also specify the order according to which the values of the variables are evaluated. This is where λ-notation becomes necessary, the respective assignment of variables is ensured by this notation.

3.2.1 Lambda-Expression

λ-expression is a list. Its first element is the atom LAMBDA. The second element is the list of variables from this expression and the third element - the given expression.

Examples.

```
(  ( LAMBDA  (X Y) (PLUS X Y)) 5 3)
```

```
(  ( LAMBDA  (X Y) (PLUS (TIMES Y 2) X)) 5 3)
```

Evaluating the λ-expression is done as follows. The successive values of arguments of the λ-expression are assigned to the successive variables from the list of variables of λ-expression in such an order, that the value of the first argument is assigned to the first variable, the value of the second expression to the second variable, and so on. Next the functional expression of the λ-expression is evaluated with the values of variables taken as the values of the respective arguments. The list after atom LAMBDA will be called a list of λ-variables.

We will explain the evaluation of lambda with an example:

```
((LAMBDA (X Y) (PLUS (TIMES Y 2) X)) 5 3)
```

In this case value 5 is assigned to `X` and value 3 is assigned to variable `Y`. Evaluating the expression

```
(PLUS (TIMES Y 2)  X))
```

with these values of variables `X` and `Y` returns the value of expression equal 11. The same value is returned also by the λ-expression.
Let's check on the computer:

```
((LAMBDA (X Y) (PLUS (TIMES Y 2) X)) 5 3) ==> 11
```

Let us also observe, that change of variables' names in λ-expression does not affect the returned value:

```
((LAMBDA (A B) (PLUS (TIMES B 2) A)) 5 3) ==> 11
```

Therefore, such variables are called the *bounded variables of this λ-expression.* The bounded variables *keep their meaning only inside the given expression - their values are specified only while evaluating the λ-expression.* There are also other variables that can occur inside λ-expressions. They are called *free variables.* Their values are defined outside the expression and their values are preserved when the λ-expression is exited after its evaluation.

For instance, in the λ-expression

```
((LAMBDA (X Y) (PLUS (TIMES Y A) X)) 5 3)
```

variable `A` is a free variable. To calculate the value of the λ-expression the value of variable `A` must be previously defined. If we assume that value 2 was previously assigned to `A`, then number 11 is returned as a value of our expression:

```
A ==> UNDEFINED VARIABLE A
X ==> UNDEFINED VARIABLE X
((LAMBDA (X Y) (PLUS (TIMES Y A) X)) 5 3)
                                        ==> UNDEFINED VARIABLE A
(SETQ A 2) ==> 2
((LAMBDA (X Y) (PLUS (TIMES Y A) X)) 5 3) ==> 11
(SETQ A 3) ==> 3
((LAMBDA (X Y) (PLUS (TIMES Y A) X)) 5 3) ==> 14.
```

It is obvious that the names of the free variables cannot be replaced inside the λ-expression as can the names of the bounded variables. Therefore, using of bounded variables is more convenient and we shall use free variables only when absolutely necessary. Observe, that external `X`, undefined, is different from the internal variable `X`.

Examples.

```
((LAMBDA  (A B) (CONS A B))  'X 'Y) ==> (X . Y)
```

The same result is returned by `(CONS  'X  'Y)`.

```
((LAMBDA (X Y) (COND ((ATOM X) Y)
                     (T X)))      'A 5) ==> 5

((LAMBDA (X Y) (COND ((ATOM X) Y)
                     (T X)))      '(A) 5) ==> (A)
```

3.2.2 Binding of Variables

Since binding of variables is an important topic, we will devote more attention to it. Let us first consider two examples.

Example 3.1

```
((LAMBDA (A) 'A) 'X)
```

The value of this expression is an atom `A` and not an atom `X` as one can suspect for λ-notation. Value `X` of variable `A` was already evaluated but not used later, because of quote. Inside the λ-notation, as always in Lisp, quote does not evaluate the value of its argument, but only returns its name.

Example 3.2

```
((LAMBDA (A B) (SETQ A B)) 'X 'Y)
```

A value `Y` will be assigned to variable `A` and not to variable `X`. This results from the fact that the expression `(SETQ A B)` is a simplified form of the expression `(SET  'A  B)`.

Let us now analyze how the λ-expressions are evaluated. There are two bounded variables `A` and `B` in this expression. Before evaluating the expression `(SETQ  A  B)` the value of `A` is `X` and the value of `B` is `Y`. However, `SETQ` does not evaluate its argument, but takes its name. At this moment variable `A` is no more a variable related to this λ-expression and becomes the global variable, i.e. the variable available in the entire program. Atom `Y` is of course the value of `A`. Let us consider now the following expression:

```
(SETQ A 1) ==> 1
(SETQ B 2) ==> 2
X ==> UNBOUND VARIABLE X
((LAMBDA (A B) (SET A B)) 'X 'Y)
A ==> 1
B ==> 2
X ==> Y
Y ==> UNBOUND VARIABLE Y
```

As the result of evaluating this expression variable `X` obtains value `Y` and becomes a global variable. The above execution results from the fact that function `SET` evaluates the values of its arguments. Now we will present examples of λ-expressions with free variables.

Example 3.3

```
((LAMBDA (A B)
    (COND ((NUMBERP X)
           (SET A (COND ((ONEP X) (ADD1 B))
                        ((ZEROP X)   B)
                        (T NIL)))
          (T NIL)))
                    'Z 8)
```

In the above example the free variable `X` is used as a control variable. If its value equals 1 then the global variable `Z` obtains value 9 and if the value of variable `X` equals 0, then variable `Z` obtains value 8.

The following question can be asked at this point by a careful reader: *"if in the given program the variable bounded to some λ-expression has the same name as some free variable, defined previously in the same program, then which values will be used in this λ-expression: the value of the free variable or the value of the bounded variable assigned to it while evaluating the λ-expression?"*

For instance, if value 5 was assigned previously to variable `A` then 3 or 5 will be the value of variable `Z` after evaluating the λ-expression:

```
((LAMBDA (A B) (SET B A)) 3 'Z) ==> 3

Z ==> 3
```

The answer to this question depends on the actual implementation of the language. In most of the Lisp systems the value of the bounded variable is taken, i.e. 3 in this case. Assigning the same names to bounded and free variables must be used with care when you plan to transfer your programs to other Lisp systems, especially those having function `CSET` or equivalent, for assigning values to global variables. It is generally advised to use different names of variables in nested λ-expressions for ease of debugging, however using the same variable names is correct. Keep always in mind that λ-variables exist only while executing the given λ-expression and cannot influence expressions outside this λ-expression.

3.2.3 Problems

1. . Calculate values of the λ-variables and the values of the following expressions:

   a) `( (LAMBDA (X Y) X) 2 3)`

   b) `( (LAMBDA (X Y) 1) 2 3)`

   c) `( (LAMBDA NIL 1))`

   d) `( (LAMBDA (X Y) (PLUS X Y)) 3 4)`

   e) `( (LAMBDA (X Y) (DIFFERENCE Y X)) 2 3)`

   f) `( (LAMBDA (X Y) (TIME (ADD1 X) (SUB1 Y))) 2 4)`

   g) `( (LAMBDA (A) (CAR A)) ((QUOTE (A B C))))`

   h) `( (LAMBDA (A B) (CONS A B)) 'X 'Y)`

   i) `( (LAMBDA (A B C) (CONS (CDR A) (CONS C (CDR B)))) '(A . B) '(X Y Z) 'Z)`

2. . Evaluate values of the λ-expressions:

   a)
   ```
   ((LAMBDA (X Y Z) (COND ((ONEP X) (SETQ A Z))
                          ((ZEROP X) (SETQ A Y))
                          (T (PLUS Y Z))))            1 2 7)
   ```

   b)
   ```
   ((LAMBDA (A B C) (COND ((NULL A) B)
                          ((NULL (CDR A)) (CAR A))
                          (T (CONS A C)) ))
                                    '((A B C)) 'X 'Y)
   ```

3. . Write λ-expression realizing the following arithmetic expressions:

a) `(X + Y) * (Z - 2)`

b) `(Z + 1) * (X + Y / 2)`

4. . Write λ-expression to select the second element from the list being its argument. If the argument is not the list with at least two elements value NIL should be returned.

5. . Write λ-expression which executes the following expressions on arguments `X, Y, Z`:

```
if A = 0 then X + Y
if A = 1 then X - Y
if A = 2 then X + Y +Z
if  A = 3 then (X + Y) / Z
if A is a literal atom - (CONS X Y)
otherwise NIL.
```

A is a free variable.

3.2.4 Answers to Problems

1.

a) `X` ==> 2, `Y` ==> 3, value 2,

b) `X` ==> `Z`, `Y` ==> 3, value 1,

c) `NIL`,

d) `X` ==> 3, `Y` ==> 4, value 7,

e) `X` ==> 2, `Y` ==> 3, value 1,

f) `X` ==> 2, `Y` ==> 4, value 9,

g) `A` ==> `(A  B  C)`, value `A`,

h) `A` ==> `X`, `B` ==> `Y`, value `(X  .  Y)`,

i) `A` ==> `(A  .  B)`, `B` ==> `(X  Y  Z)`, `C` ==> `Z`, value `(B  Z  Y  Z)`.

2.

a) 7

b) `((A B C) . Y)`

3.

a) `(LAMBDA (X Y Z) (TIMES (PLUS X Y) (DIFFERENCE Z 2)))`

b) `(LAMBDA (X Y Z) (TIMES (ADD1 Z) (PLUS X (QUOTIENT Y 2))))`

4.

```
(LAMBDA (X)
  (COND ((NULL X) NIL)
        ((NULL (CDR X)) NIL)
        (T (CADR X))))
```

5.

```
(LAMBDA (X Y Z)
   (COND ((ATOM A)
          (COND ((NUMBERP A)
                 (COND ((ZEROP A) (PLUS X Y))
                       ((ONEP A) (DIFFERENCE X Y))
                       ((EQ A 2) (PLUS X Y Z))
                       ((EQ A 3) (QUOTIENT (PLUS X Y) Z))
                       (T NIL)))
                (T (CONS X Y))))
         (T NIL)))
```

3.3 Defining New Functions

Like in all programming languages there exist a need to define new functions by the user and to use them on the same terms as the system functions. This serves to simplify and shorten the program at the programmer's convenience. It should be however pointed out that Lisp has essentially more powerful possibilities with this respect than most of programming languages.

Defining new functions is done with pseudo-function `DEFUN` and `DEF`.

3.3.1 Function DEFUN

```
(DEFUN NAME ARGUMENTS SEXPR) %%%%%%%%%% XXXXX
PSEUDOFUNCTION, SUBR.
```

To define a new function, that selects the first element from the list we type:

```
(DEFUN FIRST (D) (CAR D))
```

The first argument is a name of the new function to be defined, next follows the list of its arguments and next the functional expression, which uses variables from the list of arguments as bounded variables.

```
(DEF FIRST (LAMBDA (D) (CAR D)))
```

The λ-expression which defines new function

The value of functions `DEFUN` and `DEF` depend on the system. Check them on your system. The side effect of evaluating this function is that the definition of the new function is stored permanently in system for further use. The function can be then used later in the programs on the same terms as the system functions.

We will now analyze an example of defining several functions.

Let us assume that we want to define three functions:

`FIRST` - that selects the first element from the list or dotted pair,

`SECOND` - that selects the second element from the list,

`THIRD` - that selects the third element from the list.

By writing definitions of these three functions it should be also considered, that their arguments can be atoms or lists that do not include a sufficient number of elements. We assume that NIL should be returned in all such cases.

Let us consider the first function. We have to consider two cases:

1) the argument of the function is an atom or an empty list (an empty list is of course also an atom - NIL, but for clearity we will distinguish the two cases). NIL is returned as a value of the function.

2) the argument is a nonempty list. The first element of this list is returned as a value of this function.

With respect to the above two cases a definition of function FIRST is:

```
(DEFUN FIRST (LAMBDA (X) (COND ((OR (ATOM X) (NULL X)) NIL)
                               (T (CAR X))) ))
```

Let us define now the second function. The following cases are possible:

1) The argument is an atom or an empty list. NIL should be returned.

2) The argument is a dotted pair with an atom as the second element. NIL is returned.

3) A nonempty list is the argument. The value of the function is the second element of this list (this can be a NIL).

Then the definition of `SECOND` is:

```
(DEF SECOND (LAMBDA (X) (COND
                         ((OR (ATOM X) (NULL X)) NIL)
                         ((OR (ATOM (CDR X)) (NULL (CDR X))) NIL)
                         (T (CADR X))) ))
```

Observe, that if we replace `(CDR  X)` with `X` in the second and in the third condition then we get the same conditions as in the definition of function `FIRST`. Applying this property, the definition of function `SECOND` is simplified to the form:

```
(DEF SECOND (LAMBDA (X) (COND
                         ((OR (ATOM X) (NULL X)) NIL)
                         (T (FIRST (CDR X)))) ))
```

Similarly, the definition of function `THIRD` will be:

```
(DEF THIRD (LAMBDA (X) (COND
                        ((OR (ATOM X) (NULL X)) NIL)
                        (T (SECOND (CDR X)))) ))
```

All these functions can be defined together, using function `DEFINE`, as follows:

```
(DEFINE '(
          (FIRST (LAMBDA (X) (COND ((OR (ATOM X) (NULL X))  NIL)
                                   (T (CAR X))) ))
          (SECOND (LAMBDA (X) (COND
                        ((OR (ATOM X) (NULL X)) NIL)
                        ((OR (ATOM (CDR X)) (NULL (CDR X))) NIL)
                        (T (CADR X))) ))
          (THIRD (LAMBDA (X) (COND
                       ((OR (ATOM X) (NULL X)) NIL)
                       (T (SECOND (CDR X)))) ))
  ))
```

3.3.2 Principles of Defining Functions

The presented in section 3.3.1 method of defining functions will be now followed with remarks.

1. The order of the defining function is arbitrary. In a definition of a functions the calls to other functions can exist, that either were already defined or that will be defined later. This is possible since function `DEF` only stores definitions of functions.

2. In all three definitions there exists a bounded variable of the same name `X`. This will not lead to any contradictions, while as we have already mentioned, the bounded variables have meaning only inside the λ-expressions to which they are assigned and do not exist outside these expressions. If inside the λ-expression there exists a call of a function that has the same names of variables, then the variables bounded to this λ-expression are not available while evaluating this function (they are *covered* by the internal variables of the function). This gives the possibility of using the bounded variables with the same names in many mutually calling functions.

3. Let us notice that in all three definitions there exist functional expressions of the type

   ```
   (OR (ATOM X) (NULL X))
   ```

   These expressions can be replaced with the predicate ATOMN :

```
(DEFUN THIRD (X) (COND ((ATOMN X) NIL) (T (SECOND (CDR X)))))
(DEFUN FIRST (X) (COND ((ATOMN X) NIL) (T (CAR X))))
(DEFUN SECOND (X) (COND ((ATOMN X) NIL) (T (FIRST (CDR X)))))
(DEFUN ATOMN (X) (OR (ATOM X) (NULL X)))
```

When we now call this program with a call

```
(THIRD '(one two three START!))
```

then the evaluation will be executed as follows.
Function `THIRD` is called. The value

```
(DEFUN THIRD (X) (COND ((ATOMN X) NIL) (T (SECOND (CDR X)))))
```

is assigned to the bounded variable X. Next evaluated is the value of the conditional from the definition of this function. Function ATOMN with an argument of X ==> (one two three START!) is called. This value is assigned to variable X from the definition of ATOMN. NIL is returned by the corresponding expression and then by ATOMN. Evaluation returns to THIRD and the expression (SECOND (CDR X)) is evaluated. The value of the argument, i.e. the value of (CDR X), is evaluated. While list (one two three START!) is the value of X then (two three START!) is the value of the argument of function SECOND. This value is assigned to variable X bounded to the λ-expression from the definition of function SECOND. Next the conditional from the definition of this function is evaluated. Because (ATOMN X) returns NIL then the expression

(FIRST (CDR X))

is evaluated. Since (two three START!) is the value of variable X then (three START!) is the value of the argument of function FIRST. This value is assigned to variable X bounded to λ-expression from the definition of function FIRST. The conditional from the definition of this function is evaluated. Since once more NIL is the value of (ATOMN X) then (CAR X) is evaluated. The value of variable X equals (three START!), then three is returned as the value of the expression and of the function FIRST. This value is returned to function SECOND and becomes the value of this function. Because the value of function SECOND is returned as the value of function THIRD, then

```
(THIRD '(one two three START!)) ==> third
```

The method of evaluating function THIRD can be schematically presented as follows:

```
0    function called          : THIRD
     argument                 : (one two three START!)
     argument's value         : (one two three START!)
     bound                    : X ==> (one two three START!)
1    function called          : ATOMN
     argument                 : X
     argument's value         : (one two three START!)
     bound                    : X ==> (one two three START!)
1    function's value         : NIL
1    function called          : SECOND
     argument                 : (cdr X)
     argument's value         : (two three START!)
     bound                    : X ==> (two three START!)
2    function called          : ATOMN
     argument                 : X
     argument's value         : (two three START!)
     bound                    : X ==> (two three START!)
2    function's value         : NIL
2    function called          : FIRST
     argument                 : (cdr X)
     argument's value         : (three START!)
     bound                    : X ==> (three START!)
3    function called          : ATOMN
     argument                 : X
     argument's value         : (three START!)
     bound                    : X ==> (three START!)
3    function's value         : NIL
2    function's value         : three
1    function's value         : three
0    function's value         : three
```

The numbers from the left denote the levels on which the given function is called. The external level is denoted with number 0 and the subsequent internal levels by numbers 1, 2, and 3. If the function called at level n calls some function then this one is called on level n+1. Let us notice that function ATOMN is called by function THIRD on level 1, by function SECOND on level 2 and by function FIRST on level 3.

3.3.3 Simple Recursive Functions

In the above example function THIRD has called function SECOND and this one has in turn called function FIRST. One of the most important properties of Lisp is *recursion*, which is *the ability of a function to call itself.* First we will

discuss recursion with use of simple examples. Let us assume that we have to write a two-argument function PLUS2 with integers as values of arguments and their sum as a value. Addition shall be implemented by adding one to the first number and subtracting one from the second number until the second argument becomes 0. Having this general idea let us now distinguish all possible cases:

1. The second number equals zero. Then the first number is returned as a value.
2. The second argument equals one. Then the first argument increased by one is returned.
3. The second argument is neither zero nor one. In such a case we must add one to the first argument, subtract one from the second argument and call again function PLUS2 with the new values of arguments:

```
(DEFUN PLUS2 (A B)
        (COND ((ZEROP B)  A)
              ((ONEP B)  (ADD1 A))
              (T (PLUS2 (ADD1 A) (SUB1 B) ))))
```

We can also notice that the second clause in the conditional is redundant, while such a case is covered by the third clause. At last we will obtain:

```
(DEFUN PLUS2 (A B)
    (COND ((ZEROP B) A)
          (T (PLUS2 (ADD1 A) (SUB1 B) ))))
```

To analyze the behavior of this function let us call

```
(PLUS 5 4)
```

We will trace the arguments and the values as previously:

```
0     function              : PLUS2
      arguments             : 5  4
      arguments' values     : 5  4
      bound                 : A ==> 5 , B ==> 4
1     function              : PLUS2
      arguments             : (ADD1 A)  (SUB1 B)
      arguments' values     : 6  3
      bound                 : A ==> 6 , B ==> 3
2     function              : PLUS2
      arguments             : (ADD1 A)  (SUB1 B)
      arguments' values     : 7  2
      bound                 : A ==> 7 , B ==> 2
3     function              : PLUS2
      arguments             : (ADD1 A)  (SUB1 B)
      arguments' values     : 8  1
      bound                 : A ==> 8 , B ==> 1
4     function              : PLUS2
      arguments             : (ADD1 A)  (SUB1 B)
      arguments' values     : 9  0
      bound                 : A ==> 9 , B ==> 0
4     function's value      : 9
3     function's value      : 9
2     function's value      : 9
1     function's value      : 9
0     function's value      : 9
```

As we see, function PLUS2 calls itself repeatedly with the changing argument until reducing to the simple case of the second argument's value equal to zero.

The two principles useful by defining recursive functions are the following:

Principle RECURSION-1.
Start from the simplest case. The simpliest case is for such values of arguments that the function's value can be easily formulated. Such case is usually an empty list, a single symbolic atom or a number. In case of numeric calculations this is usually the number zero. Let us remark here that since an empty list is identical with an atom NIL than *T* is a value of predicate ATOM with an argument whose value is an empty list:

(ATOM NIL) ==> *T*

Empty list as a specific case must be then sometimes recognized separately with use of predicate NULL.

Principle RECURSION-2.
More complex cases should be reduced to simpler cases by calling the same function with new arguments. The arguments should be changed in such a way that after some number of calls the problem is reduced to one of the simple cases. For symbolic expressions this usually results by taking CAR or CDR of the given expression. In the case of numbers it results from adding or subtracting some values (for instance one) from the arguments. The recursive calls must always terminate at certain level of nesting, not to be repeated infinitely. The values of the arguments must be changed in such a way that the function is never called with the same values of all arguments twice. This is to ensure finite execution and not the infinite sequence of recursive calls.

While defining a function an attention must be also payed to consider all possible cases of the arguments' values, and to provide all the function's arguments with the values which are accepted by this function. We will consider now some more examples of defining recursive functions.

Example 3.4

Write a definition of function FACTORIAL, which calculates factorial of the value of the argument. If the value of the argument is not a natural number, then NIL should be returned. There are two simple cases in this problem: when argument's value equals 0 and when it equals 1. Before the evaluation we shall eliminate the cases when the function has value NIL. The complex cases are reduced to the recursive call using the rule:

```
  n!   =  n * (n - 1) !
```

```
(DEFUN  FACTORIAL (X)
    (COND ((OR (NOT (NUMBERP X))
                     (FLOATP X)
                     (MINUSP X))     NIL)
          ((OR (ZEROP X) (ONEP X))     1)
          (T (TIMES X (FACTORIAL (SUB1 X))))  ))
```

Let us analyze the order of predicates in the first clause of the conditional. If the argument is not a number then (NOT (NUMBERP X)) ==> *T* and the next arguments of OR are not evaluated. There is then no error when a non-numeric argument is given. Similarly, next arguments of OR make the following arithmetic predicates and functions safe from evaluation with arguments which are not positive fixed-point numbers. Let us also notice that the function defined as above is not efficient, because checking of the argument is done on each level of the recursive call. We better rewrite the definition to one in which the checking is done only at the very beginning:

```
(DEFUN FACTORIAL (X)
     (COND ((OR (NOT (NUMBERP X))
                      (FLOATP X)
                      (MINUSP X))  NIL)
           (T (FACTORIAL1 X))))

(DEFUN FACTORIAL1 (X)
        (COND ((OR (ZEROP X) (ONEP X))   1)
              (T (TIMES X (FACTORIAL1  (SUB1 X))))  ))
```

Example 3.5

Write a definition of a two-argument function ASK. The value of the first argument is a list and the value of the second argument is an atom. The function searches the list being its first argument. If one of the elements of this list is an atom identical to the value of the second argument then *T* is returned as a value of the function. If such element does not exist, NIL is returned.

The simple case in this example is when an empty list is a value of the first argument. It shall be checked if the successive elements of the list are atoms. We obtain:

```
(DEFUN ASK (X Y) (COND ((NULL X) NIL)
                       ((AND (ATOM  (CAR X))
                                (EQ (CAR X)  Y))    T)
                       (T (ASK  (CDR X)  Y))))
```

Let us trace the evaluation of ASK for the call:

```
(ASK '(ALFA BETA GAMMA) 'BETA)

0     function              :    ASK
      arguments             :    (ALFA BETA GAMMA)    ,   BETA
      values                :    (ALFA BETA GAMMA)    ,   BETA
      bound                 :    X ==> (ALFA BETA GAMMA)  ,  Y ==> BETA
1     function              :    ASK
      arguments             :    (CDR X)  ,  Y
      values                :    (BETA GAMMA)    ,   BETA
      bound                 :    X ==> (BETA GAMMA)  ,  Y ==> BETA
1     function's value      :    *T*
0     function's value      :    *T*
```

Let us notice, that this function checks if the first element of the list is the atom sought. If this condition is not satisfied it calls itself with an argument whose value is the given list without the first element. This is repeated until the sought atom is found or the list gets empty.

Example 3.6

Write a definition of the two-argument function EQUAL. The arguments are arbitrary. The value of EQUAL is *T* if the arguments are identical S-expressions and NIL otherwise.

We have already described behavior of this function in section 3.1.1.

The simple case is when one or both arguments have atomic values. The complex cases are reduced to simple cases using CAR and CDR:

```
(DEFUN  EQUAL (A B)
     (COND ((AND (ATOM A) (ATOM B))
            (EQ A B))
           ((OR (ATOM A) (ATOM B))
            NIL)
           ((EQUAL (CAR A) (CAR B))
            (EQUAL (CDR A)  (CDR B)))
           (T NIL)))
```

Let us trace the evaluation for the call

```
(EQUAL '((A B) C D) '((A B) C X))



0      function               :  EQUAL
       values of arguments    :  ((A B) C D)    ,    ((A B) C X)
1      function               :  EQUAL
       values of arguments    :  (A B)          ,         (A B)
2      function               :  EQUAL
       values of arguments    :  A   ,   A
       value of function      :  *T*
2      function               :  EQUAL
       values of arguments    :  (B)      ,      (B)
3      function               :  EQUAL
       values of arguments    :  B     ,     B
3      value of function      :  *T*
3      function               :  EQUAL
       values of arguments    :  NIL  , NIL
3      value of function      :  *T*
2      value of function      :  *T*
1      value of function      :  *T*
       values of arguments    :  (C D)     ,      (C X)
```

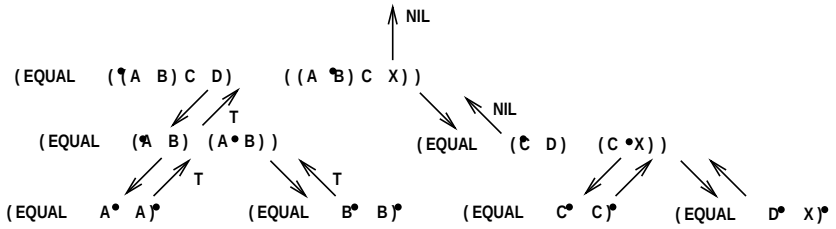


Figure 3.18: Illustration of Recursive Calls for function EQUAL

```
2      function                :  EQUAL
       values of arguments     :  C      ,    C
2      value of function       :  *T*
2      function                :  EQUAL
3      values of arguments     :  (D)   ,  (X)
3      function                :  EQUAL
       values of argument      :  D    ,     X
3      value of function       :  NIL
2      value of function       :  NIL
1      value of function       :  NIL
0      value of function       :  NIL
```

As we see in the above example, function EQUAL is nesting down the structures of its arguments, reducing comparison of more complex expressions to many comparisons of simplier expressions. The comparison of the expressions: ((A B) C D) and ((A B) C X) is reduced at the first level to the comparison of expressions: (A B) and (A B) and (C D) and (C X). There are four comparisons at the second level: A and A, (B) and (B), C and C, (D) and (X). The first and the third comparison are already the simple cases. The second and the fourth comparisons are reduced to the third-level comparisons: B and B, NIL and NIL, D and X, NIL and NIL. The fourth comparison is not executed, since the third one returns NIL. Let us note that if at any level a comparison returns NIL then the next comparisons are not executed. The evaluation of the function can be presented with use of a tree whose levels correspond to the levels of calls.

The calls are illustrated in Figure 3.18.

It is also advised the reader to manually trace the evaluation of EQUAL using graphical diagrams like one above for this and other calls. Follow with the same done in box notation. Let us also consider the execution of this function using the computer implementation of arguments for the call (EQUAL '(A B) '(A B)).
Let the memory implementation be as follows:

```
911    A      912
912    B      NIL
```

and

```
920      A    921
921      B    NIL
```

The pointers of arguments of EQUAL point to cells 911 and 920. While none of these memory cells corresponds to an atom - the addresses from the first and the second parts of these cells are compared. The addresses from the first parts pint to the cell corresponding to atom A. The addresses from the second parts od these cells point to correspond to atoms.

Next the addresses from the first and the second parts of these cells are compared. The addresses from the first parts point to the cell assigned to atom B, and the addresses from the second parts point to the cells corresponding NIL. The comparison was then completed with the positive result. In standard Lisp function EQUAL is more complex and enables also comparison of numbers. As an example write please a definition of EQUAL which uses function EQN to compare numbers.

Example 3.7

Write a definition of function LAST, that selects the last element from the list, being the value of its argument. There are two simple cases for this example:

1. A value of the argument is an atom or an empty list.

2. A value of the argument is a list with one element.

The remaining cases can be reduced to one of the above by calling function recursively to a list with removed first element:

```
(DEFUN LAST (X)
  (COND ((ATOM X) NIL)
        ((NULL (CDR X)) (CAR X))
        (T (LAST (CAR X)))))
```

The reader is asked to trace the evaluation of this function with each of the following arguments:

- an atom,

- a dotted pair,

- a list (normally ending with nil),

- a list ending with a dotted pair.

NCONC.

```
(NCONC LIST S)
NORMAL, SUBR.
```

NCONC is similar to APPEND in that it joins its two arguments into a single S-expression. NCONC actually modifies LIST to replace the terminal NIL with a pointer to S. NCONC returns the value of its first argument, modified according to its behavior. This can be contrasted to APPEND which does not change values of its arguments.

CONC.

```
(CONC LIST1 LIST2  ... LISTN)
NORMAL, FSUBR.
```

CONC is similar to NCONC in that it concatenates its arguments into a list. However, CONC accepts an arbitrary number of lists and returns LIST1 internally modified to be the concatenation of all the lists in the order of their presentation. CONC modifies the structures of all lists except the last one.

EFFACE.

```
(EFFACE S LIST)
NORMAL, SUBR.
```

EFFACE removes the first occurrence of the item S from LIST. EFFACE modifies the existing structure of the list. EFFACE is called REMOVE in some systems, check it on yours, the behavior may slightly differ.

```
(REMOVE 'F '(E F R F)) ==> (E R)
(REMOVE '(R T) '((R T) (R T))) ==> NIL
```

LENGTH.

```
(LENGTH S)
NORMAL, SUBR.
```

The argument of LENGTH may be either an atom or a list. In the first case value zero is returned. The length of list (number of elements) is returned otherwise.

NTH.

```
(NTH S)
NORMAL, SUBR.
```

NTH returns the FIXNUMBER top-level element in LIST. FIXNUMBER must be a positive number less than or equal to the number of elements in LIST. If it is greater, then NIL is returned.

```
(NTHELEM 3 '(one two three four)) ==> three
(NTH '(A B) 1) ==> A
```

COPY.

```
(COPY S)
NORMAL, SUBR.
```

COPY copies its argument. It means that the entirely new list structure is returned, that occupies different memory locations, but is functionally equivalent (EQUAL) to the value of the argument.

```
A ==> (one more list)
(EQUAL A A) ==> *T*
(EQUAL A (COPY A)) ==> *T*
(EQ A (COPY A)) ==> NIL
```

REVERSE.

```
(REVERSE LIST)
NORMAL, SUBR.
```

REVERSE returns a new list with the same top-level elements as in the value of the argument, but in the reverse order. Sublists of elements are not reversed.

```
(REVERSE '(1 2 3 4)) ==> (4 3 2 1)
(REVERSE '((ONE 1) (TWO 2) (THREE 3))) ==> ((THREE 3) (TWO 2) (ONE 1))
```

REVERSIP.

```
(REVERSIP LIST)
PSEUDOFUNCTION, SUBR.
```

REVERSIP performs an in-place reversal of a list. It returns the same value as REVERSE - reversed list, but it is not a new list but the same list as the value of the argument. It was internally modified in the memory such the top-level elements appear in reverse order.

SUBLIS.

```
(SUBLIS  LIST S)
NORMAL, SUBR.
```

The value of the first argument must be a list of the form:
((S1 . S2) (S3 . S4) ... (SN-1 . SN)).
The value of the second argument is an arbitrary S-expression. SUBLIS returns an expression S in which each occurrence of the left part of a dotted pair is replaced with the right part of the corresponding pair. SUBLIS permits for simultaneous substitution of an arbitrary number of arbitrary expressions. It does not modify the existing structure but creates a new one.

```
(SETQ sentence '(A GOOD MAN KISSED A POLICEWOMAN IN THE PARK))
==>   (A GOOD MAN KISSED A POLICEWOMAN IN THE PARK)

(SUBLIS
'((GOOD . BAD) (KISSED . KILLED) (POLICEWOMAN . POLICEMAN))
  sentence)
==>   (A BAD MAN KILLED A POLICEMAN IN THE PARK)

sentence  ==> (A GOOD MAN KISSED A POLICEWOMAN IN THE PARK)
```

3.3.4 Problems

1. Write the definition of the two-argument function. The value of the first argument is the list of elements, where each of the elements is the list of three atoms, for instance

   ```
   ((A 2 3) (B C X) (W Y Z))
   ```

   The value of the second argument is an atom. Consider the following variants to calculate the value of the function:

a) The three-element list whose first element is identical to the value of the second argument of the function. This list is selected among the elements of the list being the value of the first argument.

b) Third element of the list as in point a.

c) Second element from the triple, whose third element equals the value of the second argument.

d) List composed of the first and the third element of the triple, whose second element is identical as the value of the second argument of the function.

2. Write the definition of function `ADD`. The value of the argument of this function is the list of numbers. The sum of these numbers is returned as the value.

3. Write the definition of function `SUMOFPROD`. The argument of the function is the list of lists of numbers. As the value of the function returned is the number obtained by multiplication of the numbers in all sublists and adding of all the results of multiplications.

4. Write the definition of function `LENGTH`. The value of the function's argument is an arbitrary list. The number of elements in this list is returned as the function's value.

5. Write the definition of function `LENGTHATOM`. The value of the function's argument is an arbitrary list. The number of atoms from all levels of the expression is returned as the value.

6. Write the definition of function `SETPRODUCT`. The value of both arguments are lists of different atoms (i.e, they represent sets). The value of the function is the set of all atoms included in both lists.

7. Write the definition of function `SETSUM`, of arguments as in the previous problem. The value of the function is a list of atoms that occur at least in one of the arguments (this is the sum of sets).

8. Write the definition of function `SETDIFFERENCE`, of arguments as in the previous problem. The value of the function is a list of atoms that occur exactly in one of the lists (this is the symmetric difference of sets).

3.3.5 Solutions to Problems

1 a)

```
(DEFINE '(
           (ASS1 (LAMBDA (X A)
                   (COND ((NULL X) NIL)
                         ((EQ (CAAR X) A) (CAR X))
                         (T (ASS1 (CDR X)  A ))) ))
        ))
```

b)

```
(DEFINE '(
           (ASS2 (LAMBDA (X A)
                   (COND ((NULL X) NIL)
                         ((EQ (CAAR X) A) (CADDAR X))
                         (T (ASS2 (CDR X)  A ))) ))
        ))
```

c)

```
(DEFINE '(
           (ASS3 (LAMBDA (X A)
                   (COND ((NULL X) NIL)
                         ((EQ (CADDAR X) A) (CADAR X))
                         (T (ASS3 (CDR X)  A ))) ))
        ))
```

d)

```
(DEFINE '(
           (ASS4 (LAMBDA (X A)
                   (COND ((NULL X) NIL)
                         ((EQ (CADAR X) A) (CONS (CAAR X) (CDDAR X)))
                         (T (ASS4 (CDR X)  A )) )) )
        ))
```

2

```
(DEFINE '(
           (ADD (LAMBDA (X)
                   (COND ((NULL X) 0)
                         (T (PLUS (CAR X) (ADD (CDR X))) )) ))
        ))
```

3

```
(DEFINE '(
           (SUMOFPROD (LAMBDA (X)
                   (COND ((NULL X) 0)
                         ((NULL (CAR X)) (SUMOFPROD (CDR X)))
                         (T (PLUS (PROD (CAR X))
                                  (SUMOFPROD (CDR X)) ))) ))
           (PROD (LAMBDA (X)
                   (COND ((NULL X) 1)
                         (T (TIMES (CAR X)
                                   (PROD (CDR X)) ))) ))
         ))
```

4

```
(DEFINE '(
           (LENGTH (LAMBDA (X)
                   (COND ((NULL X) 0)
                         (T (ADD1 (LENGTH (CDR X)) ))) ))
         ))
```

5

```
(DEFINE '(
           (LENGTHATOM (LAMBDA (X)
                   (COND ((NULL X) 0)
                         ((ATOM X) 1)
                         (T (PLUS
                                  (LENGTHATOM (CAR X))
                                  (LENGTHATOM (CDR X)) )) )))
         ))
```

6

```
(DEFINE '(
           (SETPRODUCT (LAMBDA (X Y)
                   (COND ((NULL X) NIL)
                         ((MEMSET (CAR X) Y)
                             (CONS (CAR X)
                                   (SETPRODUCT (CDR X) Y)))
                         (T (SETPRODUCT (CDR X) Y)) )))
           (MEMSET (LAMBDA (X Y)
                (COND ((NULL Y) NIL)
                      ((EQ (CAR Y) X) T)
                      (T (MEMSET X (CDR Y)) )) ))
         ))
```

7

```
(DEFINE '(
           (SETSUM (LAMBDA (X Y)
                   (COND ((NULL X) Y)
                         ((MEMSET (CAR X) Y)
                          (SETSUM (CDR X) Y))
                         (T (SETSUM (CDR X)
                                    (CONS (CAR X) Y)) )) ))
         ))
```

8

```
(DEFINE '(
          (SETDIFFERENCE (LAMBDA (X Y)
                  (COND ((NULL X) Y)
                        ((MEMSET (CAR X) Y)
                         (SETDIFFERENCE
                             (CDR X)
                             (US (CAR X) Y)))
                        (T (SETDIFFERENCE
                             (CDR X)
                             (CONS (CAR X) Y)) ))))
          (US (LAMBDA (X Y)
                 (COND ((NULL Y) NIL)
                       ((EQ (CAR Y) X) (US X (CDR X)))
                       (T (CONS (CAR X) (US X (CDR Y)) )) )))
        ))
```

3.4 Special Form PROG

In the previous section we have described the method to define new functions in Lisp. This method is usually used for creating rather simple functions of not very complex structure. The difficulties can, however, arise when one attempts to define complex functions that have operations executed one by one and with the possibility of creating loops and jumps, thus changing the execution order, as it is possible for instance in FORTRAN or PASCAL. The form `PROG` is used to combine sub-programs in such a way.

3.4.1 Pseudofunction PROG

```
(PROG LAT S1 S2 ... SN
PSEUDOFUNCTION, FSUBR.
```

`PROG` is the function of an arbitrary number of arguments. The first of them is the list of the so-called program variables, i.e. auxiliary variables, which are available only in the scope of this `PROG`. These variables can be used by the programmer in any method convenient to him or her. They are assigned the value NIL at the time of their defining i.e. when the first argument of `PROG` is evaluated). The values of these variables can be changed later on by use of the already known to us functions `SET` and `SETQ`, whose behavior will be detailly explained in the sequel.

If the program variables are not necessary – the first argument of `PROG` is NIL (equivalently, `()`). The next arguments of `PROG` can be auxiliary S-expressions. They are evaluated one by one, from left to right, according to the following rules:

1) If the literal atom stands as the next argument then its value is not evaluated and this atom is treated as a label. Only the literal atoms can be the labels in Lisp.

2) The function `GO` with the argument being the label will cause jump to this label, i.e. from the expression directly following the respective label.

3) If function `RETURN` is encountered, the value of its argument is evaluated and returned as the value of `PROG`. This completes the evaluation of `PROG`.

4) If all the arguments of `PROG` have been evaluated and `RETURN` was not encountered then NIL is returned as the form's value.

5) If the conditional expression `COND` is an argument of form `PROG` and none of the first elements of its propositions evaluates to non-NIL, then the argument following `COND` is evaluated. In other cases, the evaluation of the conditional expression is executed exactly as in ordinary functional expression.

6) The remaining functional expressions are evaluated in the standard way.

Now we will present the method of evaluating the form `PROG` using simple examples.

Example 3.8

```
(PROG (A B)
      (SETQ A (SETQ B 0)))
```

A and B are the program variables as the elements of the list being the first argument of PROG. This form includes only one expression for evaluation.

```
(SETQ A (SETQ B 0))
```

As the result of its evaluation, the variables A and B, which have been assigned values NIL while declared, are assigned value 0. NIL is returned as a value of this form PROG.

Example 3.9

```
    (PROG (A)
          (SETQ A 0)
       AA (SETQ A (ADD1 A))
          (GO AA)   )
```

In this example, A is the program variable. The evaluation of the form PROG proceeds as follows: NIL is assigned to A when declared as the program variable. Next, the expression (SETQ A 0) is evaluated and variable A is assigned the value 0. Expression AA is treated as a label. As the result of evaluating the expression (SETQ A (ADD1 A)) value 1 is assigned to A. As a result of evaluation of the next expression (GO AA) the jump to the segment of the program directly following label AA is executed. The next evaluated expression is (SETQ A (ADD1 A)).

In the loop the variable A will be increased by one. The loop has no exit, then the operation will be repeated until overflow. Of course, using of loops with no exit is in most cases an error.

Example 3.10

```
    (PROG (A)
    (SETQ A 0)
 AA (SETQ A (ADD1 A))
    (COND ((LESSP A 11) (GO AA)))
    (RETURN A))
```

The nonconditional jump was replaced with the conditional jump in this example. While A has a value less than 11, the jump to label AA is executed. After value 11 is assigned to variable A, the program transits to the calculation of the expression following COND, i.e. (RETURN A). As a result of calculation of this expression, the value of variable A (i.e. 11) is returned as a value of form PROG and its execution is completed.

The examples considered thus far have, of course, no practical application, their only reason was to explain how PROG is evaluated.

Now some functions will be presented which use the form PROG and have practical applications. Some of them have been already defined previously. As the first example, we give the PROG counterpart of function FACTORIAL

```
(DEFINE '(
    (FACTORIAL (LAMBDA (X)
    (PROG (A)
    (SETQ A 1)
 AA (COND ((ZEROP X) (RETURN A)))
    (SETQ A (TIMES A X))
    (SETQ X (SUB1 X))
    (GO AA) )))
    ))
```

Note that variable X is not declared as the program variable.

The next example is function LAST which selects the last element from the list being its argument. The definition of this function is:

```
(DEFINE '(
    (LAST (LAMBDA (X) (PROG (Y)
          (SETQ Y X) (COND ((NULL Y) (RETURN (QUOTE EMPTY))))
```

```
   AA (COND ((NULL (CDR Y)) (RETURN (CAR Y))))
      (SETQ Y ( CDR Y))
      (GO AA)
          )))   ))
```

The definition assumes that atom `EMPTY` is returned when the empty list is given as an argument.
Function `LENGTH` gives as its value the number of elements in the list being its argument.

```
(DEFINE '(
   (LENGTH (LAMBDA (X)
      (PROG (A)
         (SETQ A 0)
      AA (COND ((NULL X) (RETURN A)))
         (SETQ A (ADD1 A))
         (SETQ X (CDR X))
         (GO AA) )))
             ))
```

Let us note that the given in the previous section recursive definitions of the same functions are more elegant, simpler and compact but not necessarily faster (depends on Lisp implementation).

Example 3.11

Write the definition of function `ZERLIST`. The argument of this function is the list of variable's names. The operation of the function consists in zeroing global values of all variables from the list being its argument. The iterative definition of this function is the following:

```
(DEFINE  '(
   (ZERLIST (LAMBDA (A)
        (PROG NIL
     AA (COND ( (NULL A) (RETURN NIL)))
        (SET (CAR A) 0)
        (SETQ A (CDR A))
        (GO AA)  )))
           ))
```

The recursive definition of this function is:

```
(DEFINE '(
   (ZERLIST  (LAMBDA (A)
      (COND ((NULL A) NIL)
            ( T (CONS (SET (CAR A) 0)
                      (ZERLIST (COR A)) )))
                      ))    ))
```

The `CONS` function was artificially applied to evaluate both its arguments. This will additionally produce a list of zeros which is returned as the value of `ZERLIST`.

Examples of definitions as above, in which several expressions are evaluated in one function, occur often. To enable writing such functions without the necessity of using artificial constructs, the functions `PROG2` and `PROGN` are introduced to Lisp.

Some Lisp authors treat PROG as obsolete and do not recommend to use it. For several reasons that will be explained in the sequel, but are mostly related to hardware descriptions and theory, we will still use PROG and GO, often in a restricted or limited way.

3.4.2 Pseudofunctions PROG2 and PROGN

```
(PROG2 EXP1 EXP2)
PSEUDOFUNCTION, SUBR.
```

`PROG2` is the two-argument pseudofunction with arbitrary expressions as arguments. The operation of the function consists in evaluating both its arguments while the value of the second one is returned as the function's value.

Function `ZERLIST` can be now defined as follows:

```
(DEFINE '(
   (ZERLIST (LAMBDA (A)
      (COND ((NULL A) NIL)
            (T (PROG2 (SET (CAR A) 0) (ZERLIST (CDR A)))))
           ))
           ))
```

Thanks to the use of PROG2, the unnecessary list of zeros is not produced.

```
(PROGN EXP1 EXP2 ... EXPN)
PSEUDOFUNCTION, FSUBR.
```

PROGN evaluates all its arguments from left to right and returns the value of the last one as the value of PROGN.

3.4.3 More Details on Functions inside PROG

1) The statements of PROG (i.e., the arguments of PROG except the first one) can be arbitrary Lisp expressions such as conditionals, λ-expressions, LABEL expressions (see section 7 PROG expressions, etc. These expressions can be nested in an arbitrary way: PROG in COND, COND in SETQ, etc. The value of the expressions on the lowest level inside PROG is not used - they are used only with respect to their behavior. It is often convenient to use the forms like:

```
a) (SETQ X (COND ... ))
b) (COND ( (PROG ... ) (PROG ... )))
c) (GO   (COND ...))
d) (GO   (PROG ...))
e) (SET  (COND ...) (COND ...))
```

2) Forms GO and RETURN cannot be used outside PROG. In such a case, the following error is signalized:

```
XXXX RETURN OR GO FUNCTION MAY BE USED
ONLY WITHIN A PROG FUNCTION****
```

3) If PROG forms are nested within other PROG forms, then statements GO, RETURN, SETQ and SET (for PROG variables of internal PROG) have local scope. RETURN decreases nesting level to the PROG one level lower.

```
(GO LITATOM)  or (GO S)
PSEUDOFUNCTION, FSUBR.
```

The argument of GO is usually a symbolic atom. It can also be an arbitrary form being not an atom, and evaluating to an atom which is EQ to one of the labels. GO has no value. The number of labels before each form inside PROG can be arbitrary. The multiple occurrence of the same label is not signalized. When some of the labels occur more than once, the jump is executed to its first occurrence within the most recently occurring PROG. GO has only local scope, i.e., jumps to other PROG are not allowed. For instance:

```
(PROG  (    )
 A  (GO  (PROG ( )
         X (SETQ ..)
           . . . .
           (RETURN (COND (( ... ) (QUOTE A))
                         (T (QUOTE B))))))
B  (COND (( ... ) (PROG (    )
                  Y (SETQ ...)
                            )))
C  . . . . . .
        )
```

In the PROG of the lowest level we can jump to A, B and C. In the first PROG on the internal level we can jump to X (jumps to Y, A, B and C cannot be executed). The jump from this PROG is executed through evaluation of function RETURN. Value of RETURN is transferred as the value of PROG to statement GO on the lowest level. The forms:

```
    (PRINT (GO A))
```

and

```
    (PRINT (RETURN A))
```

are not correct. They are not signalized as errors but they don't cause printout. Their proper counterparts are:

```
    (GO (PRINT (QUOTE A)))
```

and

```
    (RETURN (PRINT A))
```

The jump controlled with the value of variable X which prints the label to which it jumps can be written as:

```
    (SETQ   X  (FUN ... ))

          ...

    (GO (PRINT X))

         ...
```

The value of `(FUN ... )` shall be an atom corresponding to one of the program's labels.
If the result of evaluation of `GO` is not a label within the `PROG` including this `GO`, the error diagnostics occurs

```
    XXXX AN ATTEMPT WAS MADE TO GO TO A NON-EXISTENT LABEL
LOCATION.  EVALQUOTE WILL RETURN THE LABEL NAME ****
```

and the unknown label will be printed as the value of EVAL (or interpreter).

RETURN.

```
(RETURN S)
PSEUDOFUNCTION, SUBR.
```

`RETURN` evaluates its argument and causes Lisp to exit a `PROG` expression. The value of the argument is returned as the value of `PROG`. `RETURN` deletes from the stack everything which was written to it while execution of `PROG` inside which it was called. The old values of `PROG` variables are used in the returned value and next the values of these variables are removed from the top of the stack. The arguments of `PROG` following the form `RETURN` are not executed. Return can be evaluated in the non-`PROG` expression which is evaluated inside the expression `PROG`. In such case `PROG` is exited. If exit from `PROG` is not caused by `RETURN`, then its value is NIL.

3.4.4 Problems

Solve problems 1, 2, 3, 6, 7, 8 from section 3.3 using `PROG`.

3.4.5 Solutions to Problems

1 a.

```
(DEFINE  '(
    (ASS1 (LAMBDA (X A)
               (PROG (B)
               (SETQ B X)
              AA (COND ((NULL B) (RETURN NIL))
                       ((EQ (CAAR B) A) (RETURN (CAR B))))
               (SETQ B (CDR B))
               (GO AA) )))
              ))
```

b.

```
(DEFINE  '(
    (ASS2 (LAMBDA (X A) (PROG (B)
              (SETQ B X)
            AA (COND ((NULL B) (RETURN NIL))
                     ((EQ (CAAR B) A) (RETURN (CADDAR B))))
              (SETQ B (CDR B))
              (GO AA) )))
            ))
```

c.

```
(DEFINE  '(
    (ASS3 (LAMBDA (X A) (PROG (B)
             (SETQ B X)
           AA (COND ((NULL B) (RETURN NIL))
                    ((EQ (CADDAR B) A) (RETURN (CADAR B))))
              (SETQ B (CDR B))
              (GO AA) )))
           ))
```

d.

```
(DEFINE  '(
   (ASS4 (LAMBDA (X A) (PROG (B)
                 (SETQ B X)
               AA (COND ((NULL B) (RETURN NIL))
                        ((EQ (CADAR B) A) (RETURN
                                  (CONS (CAAR B) (CDDAR B)))))
                 (SETQ B (CDR B))
                 (GO AA) )))
          ))
```

2

```
(DEFINE  '(
  (ADD (LAMBDA (X) (PROG (A B)
               (SETQ A X)
               (SETQ B 0)
           AA  (COND ((NULL A) (RETURN B)))
               (SETQ B (PLUS (CAR A) B))
               (SETQ A (CDR A))
               (GO AA) )))
               ))
```

3

```
(DEFINE  '(
   (SET_OF_PRODUCTS (LAMBDA (X) (PROG (A B)
            (SETQ A X)
            (SETQ B 0)
       AA  (COND ((NULL A) (RETURN B)))
            (SETQ B (PLUS (PROD (CAR A)) B))
            (SETQ A (CDR A))
            (GO AA) )))

     (PROD (LAMBDA (X) (PROG (A B)
            (SETQ A X)
            (SETQ B 1)
      AA    (COND ((NULL A) (RETURN B)))
            (SETQ B (TIMES (CAR A) B))
            (SETQ A (CDR A))
            (GO AA) )))
            ))
```

4

```
DEFINE  ((
     (LENGTH (LAMBDA (X) PROG (A B)
          (SETQ A X)
          (SETQ B 0)
     AA   (COND ((NULL A) (RETURN B)))
          (SETQ B (ADD1 B))
          (SETQ A (CDR A))
          (GO AA) ))
           ))
```

5

```
(DEFINE  '(
  (SETPRODUCT (LAMBDA (X Y) (PROG (A B)
          (SETQ A X)
     AA   (COND ((NULL A) (RETURN B))
                ((MEMBER (CAR A) Y) (SETQ B (CONS (CAR A) B))))
          (SETQ A (CDR A))
          (GO AA) )))
          ))
```

6

```
(DEFINE  '(
   (SETSUM (LAMBDA (X Y) (PROG (A B)
          (SETQ A X)
          (SETQ B Y)
     AA   (COND ((NULL A) (RETURN B))
                ((NOT (MEMBER (CAR A Y))
                      (SETQ B (CONS (CAR A) B)))))
          (SETQ A (CDR A))
          (GO AA) )))
           ))
```

7

```
(DEFINE  '(
   (SETDIFFERENCE
         (LAMBDA (X Y) (PROG (A B)
         (SETQ A X)
         (SETQ B Y)
     AA  (COND ((NULL A) (RETURN B))
               ((MEMBER (CAR A) Y) (SETQ B (US (CAR A) B)))
               (T (SETQ B (CONS (CAR X) B))))
         (SETQ A (CDR A))
         (GO AA) )))

    (US (LAMBDA (X Y) (PROG (A B)
        (SETQ A Y)
    AA  (COND ((NULL A) (RETURN Y))
              ((NOT (EQ X (CAR A))) (SETQ B (CONS (CAR A) B))))
        (SETQ A (CDR A))
        (GO AA) )))
        ))
```

Chapter 4

Input/Output, Graphics, User Interface

4.1 Basic Input/Output Functions

The Lisp input/output functions have another meaning than in standard programming languages. This is a result of the Lisp's possibility of introducing data to the program in the form of functional expressions with arguments and obtaining results as the function values. This property is especially useful for interactive programming.

The reading functions read data from files.

4.1.1 Pseudofunction READ

(READ)
PSEUDOFUNCTION, SUBR.

READ is a non-argument pseudofunction. Its operation consists in reading subsequent S-expressions from the input line of the currently selected read file. Value of READ is the S-expression read. Using READ one can read arbitrarily complex S-expressions: from literal atoms to complex list structures such as whole programs being compiled or paragraphs being translated from language to language (standard text is often preprocessed to S-expressions for easier handling by Lisp). If the S-expression does not fit to one line it can be transferred to the next line by splitting it in an arbitrary place but not splitting atoms.

More than one S-expression for READ can stand on one line for reading by different READ functions or subsequent calls of the same READ (this is not so in interactive usage).

Example 4.1

If line STOP is followed by the lines

```
ABC (X Y Z) (JOHN

 MARY CAT)

 LIST PASCAL
```

then successive calls of READ will produce the following values:

```
(READ) ==> ABC
(READ) ==> (X Y Z)
(READ) ==> (JOHN MARY CAT)
(READ) ==> LIST
(READ) ==> PASCAL
```

If there is no next S-expression in the file, then atom
EOF is returned as the value of READ. This can be utilized to terminate further reading from this file.

If the
EOF

```
IN UNMATCHED LEFT PARENTHESIS
```

is printed.

4.1.2 Pseudofunction PRINT

Pseudofunctions PRINT and WRITE

```
(PRINT S BOOLEAN)
(WRITE S (BOOLEAN)
PSEUDOFUNCTION, SUBR.
```

Function PRINT serves for printing S-expressions. It is a one-argument or two-argument pseudofunction. Arbitrary S-expression can be its argument. The printout of this expression is the result of PRINT execution. PRINT prints the S-expression on the currently selected write file. The printed S-expression is also returned as the value of PRINT. The transition to the new line does not occur before writing of the new line. This means that if some S-expression was printed previously and the line was not changed then the actual S-expression will be printed in the same line (directly after the one previously printed). PRINT causes, however, transition to the next line after the S-expression was printed. The next printout will start then from the new line. If the printed S-expression doesn't fit in a single line, then it is split into more lines in such a way that none of the atoms is split.

The printouts of function PRINT occur between the printout of call of the function on the highest level (these calls which include function PRINT) and the printout of the function's value.

Example 4.2

```
    EVAL
    (PRINT `(A B C))
    (A B C)
    VALUE
    (A B C)
    EVAL
    (DEFINE `(
           (TEST (LAMBDA (X)
                 (COND ((NULL X) (NIL))
                       (T (PRINT (TEST (PRINT (CDR X))))))
           ))
           ))
  VALUE
  (TEST)
  EVAL
  (TEST `(A B C D))
  (B C D)
  (C D)
  (D)
  NIL
  NIL
  NIL
  NIL
  NIL
  VALUE
  NIL
```

The above example shows how the recursive calls of the function can be *"traced"* by the use of function PRINT. It is advised the reader to analyse carefully in this example, which printout comes from which PRINT call in the TEST function.

If the second argument is false, the S-expression will be printed using the print names of its component atoms:

```
  EVAL
  (PRINT `$$$A B$ NIL)
  A B
  VALUE
  A B
```

If the second argument is true, the atoms with nonstandard print names will have the appropriate delimiters inserted.

```
EVAl
(PRINT $$$A B$ '(ACB))
$A B$
VALUE
A B
```

The results can then be subsequently read by the Lisp input functions.

4.1.3 Pseudofunction PRIN1

```
(PRIN1 S1 BOOLEAN)
PSEUDOFUNCTION, SUBR.
```

PRIN1 is a one-argument pseudofunction of operation similar to PRINT. As PRINT, it prints the value of its argument being an atom. The difference is that PRIN1 does not terminate line after printing the atom. This permits for printing many atoms side-by-side in one line.
Execution of the sequence on internal level:

```
(PRINT (QUOTE (A B C)))
(PRIN1 (QUOTE ABCD))
(PRINT (QUOTE X))
(PRIN1 (QUOTE O))
(PRIN1 (QUOTE AB))
(PRIN1 (QUOTE C))
(PRINT 23)
```

will give the printout:

```
(A B C)
ABCD X
ABC 23
```

Let us note the spaces that precede the printouts of function PRINT (before atoms X and 23) and lack of these spaces in the case of printouts of function PRIN1 (followed by atom C in the last line). This function may be implementation dependent, check it on your system.

4.1.4 Pseudofunction PPRINT

```
(PPRINT S BOOLEAN)
PSEUDOFUNCTION, SUBR.
```

PPRINT works exactly like PRINT, the only difference is that the printout is prettyprinted. This means shifting nested S-expressions and special recognition of PROG, LAMBDA and COND to improve the readability.
If LENGTH were 50 then

```
(PPRINT //OUTPUT AT)
```

would print:

```
(LAMBDA (=====/////)
              (OR
               (AND (NOT (ATOM =====//////))
               (EQ (CAR =====/////) 'DEFINE))
              (OUTPUT //SYSOUT =====////////)))
```

4.1.5 TERPRI

```
(TERPRI)
PSEUDOFUNCTION, SUBR.
```

TERPRI is a non-argument pseudofunction, returning value NIL. Its execution causes that the next printout will start from the new line on the currently selected file.

The call:

(PROG2 (PRIN1 A) (TERPRI))

is equivalent to (PRINT A).

Example 4.3

```
(PRIN1 234)
(TERPRI)
(PRIN1 (QUOTE ABC))
(PRIN1 23)
(TERPRI)
(TERPRI)
(PRINT 23)
```

causes the printout:

```
234
ABC23

23
```

TERPRI is usually used to skip blank lines. Some examples of useful functions:

```
(DEFINE '(
     (NEWPAGE (LAMBDA NIL
              (PROG2 (PRIN1 1) (TERPRI))
       ))
     (SKIPLINES (LAMBDA (N)
                (COND((ONEP N) (TERPRI))
                     (T (PROG2 (TERPRI) (SKIPLINES (SUB1 N)))))
       ))
     (PRINCHARS (LAMBDA (N CHAR)
                (COND ((ONEP N) (PRIN1 CHAR))
                      (T (PROG2 (PRIN1 CHAR) (PRINCHARS (SUB1 N)
                                              CHAR))))
       ))
     (PRINTOP (LAMBDA (X)
              (COND((NULL x) NIL)
                   (T (PROG2 (PRINT (CAR X))
                             (PRINTOP (CDR X)))))
       ))
))
```

Function NEWPAGE causes continuation of further printouts from the new page (this is only in Lisps where the character 1 is used as a control character).

Function SKIPLINES causes leaving N free lines. N must be a positive natural number.

Function PRINCHARS will cause printing N characters CHAR, one by one, and does not change the line. The positive natural number is the value of N and CHAR is an arbitrary alphanumeric character.

Example 4.4

The call:

```
(PRINCHARS  5  'X)
```

causes printout

```
     XXXXX
```

Function PRINTOP prints successive elements of list X one under one. Arbitrary list can be the value of the argument.

Example 4.5

```
(PRINTOP  '(A B (C D) (A B ( C D))))
```

causes printout

```
A
B
(C D)
(A B (C D))
```

4.1.6 Problems

1. What are the values of successive functions: (READ) (READ) (READ) if the lines for reading look as follows:

```
a) A BC DC E G H

b) (A BC
      CD
      )A (BC) Z

c) ((A B) (X Y)

   (Z X V)

   (X Y)) Z(X V

   (Z X)))

d) 1.00

   (2.10 0.01E2)

   22Q
```

2. What printouts will cause the sequence of functions:

```
a)  (PRIN1 128) (PRIN1 (QUOTE ABC)) (PRIN1 25) (TERPRI)
    (PRINT 128)
b)  (PRIN1 228) (PRIN1 (QUOTE ABC)) (PRINT 25) (TERPRI)
    (PRINT 128)
c)  (PRINT (QUOTE Z)) (PRIN1(QUOTE A))
                      (TERPRI)(PRIN1 (QUOTE AB))
    (PRINT 121)
```

3. Write the definition of a function that:

   a) prints N times atom AT in the same line and not changing the line. K spaces shall stand among successive printouts of atom AT.

   b) prints N times atom AT in the same line and changing the line. K spaces shall stand among successive printouts of atom AT.

   c) prints successive atoms from the list of atoms LISAT in the same line and separates them with K spaces. After printing the last atom the N spaces shall be printed and the line is not changed.

   d) prints successive atoms from list of atoms LISAT, N of them in one line, separating them with K spaces. The line is changed after printing all the atoms.

Caution:
The space is the value of atom BLANK. The printout of the space can be then obtained by call (PRIN1 BLANK).

4.1.7 Solutions to Problems

1

a) A, BC, DC

b) (A BC CD), A, (B C)

c) ((A B)(X Y)(Z X V)(X Y)), Z, (X V(Z X))

2

a)
```
<from new page>
28ABC25
128
```

b)
```
28ABC 25

128
```

c)
```
 Z

 B 121
```

3

a)
```
(TEST1 (LAMBDA(N AT K)
              (COND((ONEP N)(PRIN1 AT))
                   (T(PROG2(PRIN1 AT)(PROG2(PRINCHARS K BLANK)
                                           (TEST1(SUB1 N) AT K)))))
              ))
```

b)
```
(TEST2 (LAMBDA(N AT K)
            (COND((ONEP N)(PROG2(PRIN1 AT)(TERPRI)))
                 (T (PROG2(PRIN1 AT)(PROG2(PRINCHARS K BLANK)
                                          (TEST2(SUB1 N) AT K)))))
       ))
```

Using function PRINT in the second line would cause printing one space more before the last atom.

c)
```
(TEST3(LAMBDA(LISAT K N)
            (COND((NULL(CDR LISAT))
                  (PROG2(PRIN1(CAR LISAT)) (PRINCHARS N BLANK)))
                 (T(PROG2(PRIN1(CAR LISAT))
                         (PROG2(PRINCHARS K BLANK)
                         (TEST3(CDR LISAT) K N)))))
            ))
```

d)
```
(TEST4(LAMBDA (LISAT N K) (PROG (X Y)
            (SETQ X LISAT)
         AA (SETQ Y N)
         BB (PRINT (CAR X))
            (SETQ X (CDR X))
            (SETQ Y (SUB1 Y))
            (COND ((NULL X) (RETURN (TERPRI)))
                  ((ZEROP Y) (PROG2 (TERPRI) (GO AA) )))
            (PRINCHARS K BLANK)
            (GO B)
            )))
```

4.2 Making Your Programs to Run Correctly

4.2.1 Tracing Functions

By running the functions that we are not sure of their correctness or functions including the difficult to place error it is advised to use functions TRACE and TRACESET.

Function TRACE is especially useful for tracing recursive functions and the functions which are called in many places of the program and in many *"loops"* whose execution depends on satisfaction of some conditions. The argument of function TRACE is the list of function names to be traced. NIL is returned as the value of TRACE. The side effect of TRACE is printing of the names of functions traced and the values of their arguments each time when they are called, and printing their values after each evaluation of the traced functions. The levels of nesting are enumerated with the successive letters of the alphabet and are printed inside the brackets. This is especially useful in the case of tracing recursive functions.

Example 4.6

If we insert the sequence

```
(TRACE '(FACTORIAL))
(FACTORIAL 5)
```

after the definition of FACTORIAL in the previous example, then the printout is created.

```
Function EVALQUOTE has been entered, arguments...
TRACE
((FACTORIAL))
END OF EVALQUOTE, VALUE IS...
NIL
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS...
FACTORIAL
(5)
[:A] ARGUMENTS OF FACTORIAL
5
[:B] ARGUMENTS OF FACTORIAL
4
[:C] ARGUMENTS OF FACTORIAL
3
[:D] ARGUMENTS OF FACTORIAL
2
[:E] ARGUMENTS OF FACTORIAL
1
[:E] VALUE OF FACTORIAL
1
[:D] VALUE OF FACTORIAL
2
[:C] VALUE OF FACTORIAL
6
[:B] VALUE OF FACTORIAL
24
[:A] VALUE OF FACTORIAL
120
END OF EVALQUOTE, VALUE IS...
120
```

As this simple example shows, tracing of the function essentially increases the amount of the program's printouts. We shall then use it with caution and limit the number of the traced functions, otherwise too much paper is wasted.

If tracing of some function is already unnecessary in the next part of the program, we can remove it using function UNTRACE. The argument of UNTRACE is the list of functions from which we want to remove tracing. These must be, of course, the functions which were previously traced with TRACE. UNTRACE returns NIL and its side effect is removing of the tracing from the functions.

If we, for instance, use the following sequence:

```
(TRACE '(FACTORIAL))
(FACTORIAL 5)
(UNTRACE '(FACTORIAL))
(FACTORIAL 8)
```

then only the first of the calls will be searched.

Function TRACESET is used for tracing functions including form PROG on the highest level (so called *"program functions"*). The argument of TRACESET is the list of function names for tracing. The same list is returned as a value of TRACESET. The side effect of TRACESET is printing the results of the execution of functions SET and SETQ in the traceseted functions in the form

```
<name of the program variables> =
                  <value of the program variable>
```

If the name of the function without form PROG on the highest level is encountered in the list being the argument of TRACESET then the error is signalized.

Example 4.7

In section 3.4.1 we have defined function FACTORIAL as the program function. If, after defining this function, we will place the following sequence in the program:

```
(TRACESET '(FACTORIAL))
(FACTORIAL 5)
```

then we will obtain the printout:

```
EVAL:
(TRACESET '(FACTORIAL))
VALUE:
(FACTORIAL)
EVAL
(FACTORIAL 5)

A = 1
A = 5
A = 20
A = 60
A = 120
A = 120

VALUE:
120
```

Let us note that the tracing concerns only the value of the program variable A and doesn't concern the variable X.

For removing tracing from the program variables serves function UNTRACESET. Its argument is the list of names of program functions which were previously traceseted with the use of TRACESET. The value of the argument is returned as the value of UNTRACESET. The side effect of evaluating the functional expression with function UNTRACESET is removing the TRACESET from the functions whose names are included into the list being the argument of UNTRACESET.

The tracing functions can be also used at the internal level, like:

```
(COND (A (TRACE '(FUN1)))
      ((NOT B) (TRACE '(FUN2)) (UNTRACE '(FUN1))))
```

While new programs are created, it is useful to create auxiliary printouts with the use of function PRINT (see section 4.1 for its description). We remind here only that PRINT prints the *value* of its argument. For instance placing anywhere in the program the expression:

```
(PRINT A)
```

causes printout of the *value* of variable A. In that way we can print values of the variables which we suspect to have other values than expected. Application of the following expressions is advised

```
(PROG2 (PRINT 'A) (PRINT A))
```

which prints the name of the variable and then its value. **PRINT** can be used also for writing *"flags"* of program's control transition through some program's segment. For instance, placing in some location of the program the expression:

```
(PRIN1 "1")
```

will cause printing 1 as many times as this segment was executed.

4.2.2 How to Run Your Programs

Placement of the Program

Layout of the program on lines is arbitrary from the system's point of view. We can, however, make use of this property and place the programs in clear form which simplifies detection and correction of eventual errors. It is then advised:

1) Write the calling of function **DEFINE** and all other calls of functions on the external level from the first or second column.

2) Call of function **DEFINE** should be done in the form

```
(DEFINE '(
          <definition of functions>
         ))
```

3) The definitions of functions shall be written in such a form that the first line includes the function's name, **LAMBDA** and arguments. The next line shall eventually include **PROG** and program variables. The description in the first line should be essentially shifted to the left with respect to other lines (for instance the description on the first line from the third column and the rest from the sixth column).

4) The successive S-expressions shall be printed on *separate lines*. Long S-expressions should be splitted in such a way that their readability is preserved, shifting at the same time the beginning of the description in the second and the next lines several columns to the right.

5) The conditional **COND** shall be written in the form: the first line includes the left parenthesis, atom **COND** and, eventually, the first proposition.

   The next propositions are placed on new lines, they begin under the beginning of the first proposition.

   If the description of any of the pairs is too long, then rule 4 is applied.

6) The parentheses terminating the function definition (two for non-program functions, and three for program functions) are placed on separate lines ending this definition or are at least separated with spaces from other parentheses.

7) In the remaining cases the description shall be as clear and readable as possible.

Application of the above rules enables avoiding parentheses errors as well as detecting and correcting the errors. If you are lazy to use these rules, call the pretty-printing function PP, but if you do too many errors, this will not help you much. It is good for larger programs to both obey these rules, and next use PP and compare the two printouts. This way many errors can be catched very early.

Running of Programs

The following is advised the beginning programmer:

1) Run at first simple functions to become acquinted with Lisp, its interpreter, editor and operating system.

2) Write big programs as the hierarchies of not too large of functions. Run these functions separately, starting from the simplest and when they already run, run the functions which call them, etc., until the entire program is working. The auxiliary functions can be also defined which enable the running of some functions.

3) Test the named functions on all possible cases.

Error Corrections

When you are correcting a program, assume that:

1) The incorrect program of yours, and not the interpreter's error, is the reason of the incorrect program's operation.

2) Consider carefully what was the error, where it occurred, and how it can be corrected, instead of using to the excess of the diagnostic tools.

3) If you have difficulties with error placement, use the tools from section 4.1.2. Use them with care and thought, however.

4) Pay attention not to introduce new errors, for instance, parentheses errors while correcting your initial errors.

5) In the case of non-self-explanatory effects or error diagnostics, analyze carefully the entire program - especially in the case of correcting the very unfriendly *"TIME LIMIT"* errors.

4.2.3 Interactive Usage of Lisp

In this section we will explain some slight peculiarities in the interactive usage of Lisp.

In such case, the files `SYSIN` and `SYSOUT` are equated to file `TTY` and are associated with the same buffer area. You cannot declare other files `SYSIN` or `SYSOUT`. The size of the buffer is reduced to 10 memory words and the output line is set to 70 characters, then the user sees the results in synchrony with what Lisp is actually doing. Lisp outputs information when the buffer for a file is full or whenever a read operation occurs after a write operation for a given file. It means that all output is sent to the terminal before the next input is read. The typed input is visible to the user.

The next top level S-expression should not be typed until the output of the previous one is entirely sent to the terminal.

Lisp reads one S-expression at a time and responds with printout. It means that if the input line is READ THIS and the program contains

```
(PROG2 (PRINT (READ)) (PPRINT (READ)))
```

then

```
READ
```

is printed and program will request more input instead of reading `THIS`.

The above remarks apply only to terminal input. Reading and writing to other files is as in batch mode.

4.2.4 Interrupts

Interrupts suspend the normal execution of the program and allow to execute some other actions in batch or interactive mode.

Interrupts in Lisp can be done:

- from outside, by pressing the `BREAK` key,
- from inside, by calling the `INTERRUPT` function.

By using function `DEFINT` the user can define a unique expression to be evaluated for each interrupt.

The interrupt expressions may do anything valid in the context in which they are evaluated, like:

- interaction with user (for instance calling `SYSIN`),
- changing values of variables (also //ZAP),
- changing function definitions.

4.2.5 Uses for Lisp Interrupts

The main purposes for using interrupts are:

- to query the program status,
- to determine cause of errors and to cure the errors,
- to effect special controls.

When an expression evaluated in the response to an interrupt communicates something to the user, he can determine some information about the status of his program.
He can:

- execute backtrace,
- look at the function definitions,
- look at the property lists,
- evaluate the variables or some functions on them.

When the user leaves the interrupt expression, Lisp continues execution of the interrupted process. Note, however, that any changes made in variable values, etc. are effective when the process resumes.

Example 4.8

We want to look into values of variables, etc. in some point of interactive program.

```
(SETQ A (ASS 1   2))
 .
 .
 .
(DEFINT 8  '(PROG (X)
              A (COND ((EQ(SETQ X (READ))
                         'END)
                         (PRINT "END OF INTERRUPT 8"))
                       (T (EVAL X)))
                (GO  A)  ))
 .
 .
 .
(SETQ R (FUNI A))
B (INTERRUPT 8)
```

`DEFINT` specifies interrupt number 8 and defines what is done in the case of such interrupt. The user expressions are evaluated until he will type END. In the place of program with label `B` interrupt 8 is called. `DEFINT` can be used either inside the tested program, or separately.

Before we will discuss other applications of interrupts, we present functions `DEFINT` and `INTERRUPT` with more details.

DEFINT

```
(DEFINT FIXNUMBER EXP)
PSEUDOFUNCTION, SUBR.
```

`DEFINT` binds `EXP` to the system variable $//INT1$,... ,$//INT12$ respectively to the value of `FIXNUMBER`. `EXP` is subsequently evaluated when the interrupt `FIXNUMBER` occurs. The default values of expressions are:

```
    Interrupt                    Expression

     1 and 2              (PROGN (PRINT $$$/// INTERRUPT$
                                 (SYSIN 'TTY)))
     3, 4, 5              NIL
     6                    (PROGN (SETQ //INPUT '(INPUT //SYSIN))
                                 (SETQ //MODE '(EVAL . 1)))
     7 - 12               NIL
```

If the user loses control of Lisp by *//MODE* and *//INPUT* get set to bad values, interrupt 6 may be used to recover control.

INTERRUPT

```
(INTERRUPT FIXNUMBER)
PSEUDOFUNCTION, SUBR.
```

INTERRUPT evokes the simulated interrupt

```
(FIXNUMBER
(INTERRUPT NIL)
```

is equivalent to (INTERRUPT 1).

As a result, the expression bound to //INT FIXNUMBER is evaluated.

ERROR DIAGNOSIS

If system variable //ZAP is non-NIL when an error occurs, an interrupt will occur. The value of //ZAP determines which interrupt will be used. Before the interrupt occurs, the error message will be printed so the user will know what error happened. The cause of an error can be often found by checking the status of the program. Later on it is possible to correct the cause of an error and resume the computation as though the error had not occurred. The function RETFROM is used with that respect.

NTHFNBK.

```
(NTHFNBK ATOM FIXNUMBER)
PSEUDOFUNCTION, SUBR.
```

NTHFNBK searches the stack, starting from the current top, for the FIXNUMBER-th occurrence of the function name ATOM, and returns the stack index of that entry.

RETFROM.

```
(RETFROM FIXNUMBER EXP)
PSEUDOFUNCTION, SUBR.
```

FIXNUMBER is the stack index found by NTHFNBK. RETFROM causes the stack to be *"peeled"* back to that point, and then exits the indexed function with the value of EXP as the value of the function. Note, that EXP is evaluated after the stack has been "peeled" back in the environment that held when the function was invoked. Also, the effects of TRACE and TRACESET may be affected by a call to RETFROM.

Example 4.9

Suppose we have a program

```
(DEFINE '(
   (MYFUNCTION (LAMBDA (X)
         (CAR X)  ))
   (FUN1 (LAMBDA () (PROG()
      .
      .
      .
      .
      (SETQ R (MYFUNCTION 'AT))

      )))
      ))
```

and we do

```
(SETQ //ZAP 8)
```

After error diagnosis we define new function MYFUNCTION

```
(MYFUNCTION (LAMBDA (X)
     (OR (ATOM X) (CAR C))))
```

and call the function

```
(RETFROM (NTHFNBK 'MYFUNCTION 1)
            (MYFUNCTION 'AT) )
```

4.3 Examples of Programs in Lisp

Example 4.10

Function `ART` evaluates the value of arithmetic expression like 2 + 5 * (3 - 4.0 / 8). The operators + , - * and / as well as parentheses can be included in the expression. The expression can be of arbitrary complexity. The precedence of operators * and / with respect to + and - is respected. For instance in the above noted expression the division is executed at first, next subtraction, multiplication and addition.

The program's is presented below. The reader is asked to match parantheses and trace using `TRACE`.

```
EVAL:
(DEFINE '(
 (ART (LAMBDA (X)
  (COND ((ATOM X) X)
   ((NULL (CDR X)) (CAR X))
   ((EQ (CADR X) (QUOTE X))
        (ART (CONS (TIMES (ART (CAR X))
             (ART (CADDR X))) (CDDDR X))))
   ((EQ (CADR X) (QUOTE /))
        (ART (CONS (QUOTIENT (ART (CAR X))
                (ART (CADDR X))) (CDDDR X))))
   ((MEMBER (CADR X) (QUOTE (+ -)))
     (COND ((AND (NOT (NULL (CDDDR X))) (MEMBER (CADDDR X)
                                        (QUOTE (X /))))
            (ART (CONS (CAR X) (CONS (CADR X) (CONS (ART (LIST
               (CADDR X)(CADDDR X) (CADDDR X)))
                           (CDDDDDR X)))))
                                        )
          ((EQ (CADR X) (QUOTE +)) (ART
            (CONS (PLUS (ART (CAR X))(ART (CADDR X)))
                                         (CDDDR X))))
          (T (ART (CONS (DIFFERENCE (ART (CAR X))
                                 (ART (CADDR X)))
                          (CDDDR X))))))
  (T 0)) ))     ))
```

The characters +, -, * and / are used as literal atoms. Therefore, they must be separated with spaces from other atoms.

In the case of operator / at least one of its arguments must be a floating-point number, because otherwise the result of the division will be rounded to the fixed-point integer (property of function `QUOTIENT` - see section 3).

Consider from the printout how the function ART recursively nests down analyzing the expression.

Project 4.1

Modify the above program such that instead of evaluating arithmetical expressions it will evaluate Boolean expressions. Use values `true` and `false` of variables, for 1 and 0, respectively. Use Boolean operations AND, OR, NOT, NAND, NOR, EXOR, and XNOR. Next extend for multiplexers with one and two address inputs, and majority functions of 3 arguments.

Project 4.2

Modify this program to evaluate Boolean expressions in which every argument can take values 0, 1, and X, which means a don't care (unspecified logic value). The value of the circuit should be a specified logic value whenever possible.

Example 4.11

Functions `CREATE` and `FIND` serve to create and search a book catalog. To each book corresponds a list, that includes:

- author's name
- title in parentheses

- domain in parentheses

- identification number

- number of shelf with book

The identification number is assigned by the system, the remaining data are read by one of the functions, `CREATE`. Before creating the catalog 0 is assigned to variable `NRK` and NIL to variable `KAT`.

Function `FIND` serves for finding the position of the catalog of the specified values of parameters: `AUTHOR`, `TITLE`, `DOMAIN`, `NUMBER` and `SHELF`. Below we give the program's listing as well as some examples of application of the defined functions. The reader should test it and trace it on his Lisp system.

```
(DEFINE '(
   (CREATE (LAMBDA (X)
      (PROG2
         (SETQ CAT(CONS(LIST(CAR X)(CADR X)(CADDR X)
                     (SETQ NRK (ADD1 NRK))
                  (CADDDR X))        CAT))
         (CAR CAT)) ))
(FIND (LAMBDA (PAR VAL)
   (COND ((EQ PAR 'AUTHOR) (AUT VAL CAT))
         ((EQ PAR 'TITLE) (TIT VAL CAT))
         ((EQ PAR 'DOMAIN) (DOM VAL CAT))
         ((EQ PAR 'NUMBER) (NUM VAL CAT))
         ((EQ PAR 'SHELF) (SHELF VAL CAT))
         (T NIL)) ))
(AUT (LAMBDA (X Y)
   (COND ((NULL Y) NIL)
         ((EQ (CAAR Y) X)
              (PROG2 (PRINT (CAR Y))
                     (AUT X (CDR Y))))
         (T (AUT X (CDR Y))))
   ))
(TIT (LAMBDA (X Y)
   (COND ((NULL Y) NIL)
         ((EQUAL (CADAR Y) X)
             (PROG2 (PRINT (CAR Y))
                    (TIT X (CDR Y))))
         (T (TIT X (CDR Y))))
   ))
(DOM (LAMBDA (X Y)
   (COND ((NULL Y) NIL)
         ((EQUAL (CADDAR Y) X)
             (PROG2 (PRINT (CAR Y))
                    (DOM X (CDR Y))))
         (T (DOM X (CDR Y))))
  ))
(NUM (LAMBDA (X Y)
   (COND ((NULL Y) NIL)
         ((EQUAL (CADDAR Y) X)
             (PROG2 (PRINT (CAR Y))
                    (NUM X (CDR Y))))
         (T (NUM X (CDR Y))))
  ))

(SHELF (LAMBDA (X Y)
   (COND ((NULL Y) NIL)
         ((EQUAL (CADDDAR Y) X)
             (PROG2 (PRINT (CAR Y))
                    (SHELF X (CDR Y))))
```

```
            (T (SHELF X (CDR Y))))
      ))
))
    (SET 'NRK 0)
    (SET 'CAT NIL)
    (CREATE '(SHAKESPEARE (ROMEO AND JULIET) (DRAMA) 1))
    (CREATE '(SHAKESPEARE (SUMMER'S NIGHT DREAM) (DRAMA) 2))
    (CREATE '(IONESCO (WAITING ON BECKETT) (DRAMA) 2))
    (CREATE '(WEINER (CYBERNETICS) (SCIENCE) 1))
    (CREATE '(ANDERSEN (TALES) (CHILD LITERATURE) 2))
```

Project 4.3

Modify this program such that it will search the data base of Lisp functions corresponding to various hardware blocks, such as adder, multiplier, logic gates, flip-flops with different powers consumptions, rising and falling delay times, and other useful characteristics. In addition to creating and searching functions, the reader is asked to write some statistic calculating functions for this data base of electronic components. TTL series can be a good starting point.

4.4 Basic Graphics

4.5 Other User Interface

4.6 Visualization of Boolean Functions

4.7 Visualization of Graphs

4.8 Graphic Interaction with the System

Chapter 5

Solving Satisfiability and Petrick Function Problems

This section algorithms for solving the following problems. Given is a product of terms, each term being a Boolean sum of literals, each literal being a Boolean variable or its negation.
Problem 1 (Satisfiability): Find any product of literals that satisfies all terms or prove that such product does not exist.
Problem 2 (Optimization): Find a product with a minimum number of variables that satisfies all terms or (option) prove that such product does not exist.
Problem 3 (Optimization of the Generalized Petrick function): Find such product of literals that satisfies all terms and in which a minimum number of literals is not negated or (option) prove that no product exists, that satisfies all terms.
Problem 4 (Partial Satisfiability): Find such set of literals that satisfies maximum number of terms.
Problem 5 (Complementation of Boolean function Given is a Boolean function in a Sum of Products Form. Find its complement in a Sum of Product form.

Other problems such as Tautology Checking, Convertion from Sum of Product Form to Products of Sums Form and Convertion from Product of Sums Form to Sum of Products Form are also mentioned.

The central role of the first problem in Computer Science is well established. Many reductions of practically important problems to problems 2 and 3 were shown, including problems from VLSI Design Automation, especially in logic and state machine design. We discuss tree searching algorithms implemented in Lisp.

5.1 Introduction

Given is a product of terms, each term being a Boolean sum of literals, each literal being a Boolean variable or its negation. We are interested in the following problems.
Problem 1 (Satisfiability):
Find any product of literals that satisfies all terms or prove that such product does not exist.
Problem 2 (Optimization):
Find a product of literals that satisfies all terms and which has a minimum number of variables or prove that such product does not exist.
Problem 3 (Optimization of Generalized Petrick function):
Find a product of literals that satisfies all terms and which has a minimum number of not negated literals or (option) prove that no such product exists.
Problem 4 (Partial Satisfiability):
Find a product of literals that satisfies most terms.
Problem 5 (Complementation of Boolean function):
Given is a Boolean function in a Sum of Products Form. Find its complement in the same form.
Problem 6 (Tautology Checking):
Verify whether a function in a Sum of Products Form is a Boolean Tautology.
Problem 7 (Convertion from a Sum of Products Form (SOP) to the Product of Sums Form (POS)):
Convert a Boolean function from a Sum of Products Form to the Product of Sums Form.
Problem 8 (Convertion from Product of Sums Form (POS) to Sum of Products Form (SOP)):
Convert a Boolean function from a Product of Sums Form to the Sum of Products Form.

In problems 2, 3 and 4 we can look for all solutions or for a single optimal solution. Problem 2 with all solutions looked for corresponds to the well-known Boolean Complementation Problem that occurs in the minimization of Boolean functions.

All NP-complete combinatorial decision problems are equivalent to a Satisfiability Problem [?]. The reductions of many practically important NP-hard combinatorial optimization problems can be also found in the literature.

For instance the minimization of Boolean functions can be reduced to the Covering Problem [73] and the Covering Problem can be further reduced to the Petrick Function Optimization Problem (PFOP) [?]. Many other problems, like test minimization can be also reduced to the Covering Problem [?, 73, ?].

The problem of minimization of Finite State Machines includes the Maximum Clique Problem and the problem of finding minimum closed and complete subgraph of a graph (Closure/Covering Problem) [?]. The first of these problems can be reduced to the Petrick Function Optimization Problem. The problem of the optimum output phase optimization of PLA [?] can be reduced to PFOP.

The second problem can be reduced to the Generalized Petrick Function Optimization Problem (GPFOP), introduced below. Many other problems, like AND/OR graph searching [360] or TANT network minimization [?] were reduced to Closure/Covering Problem.

A number of problems (including Boolean Minimization [?, ?], Layout Compaction [?], and minimization of number of register in compilation [?]) can be reduced to a Minimal Graph Coloring Problem. Minimal Graph Coloring can be reduced to the Problem of Finding Maximum Independent Sets and next the Covering Problem. The Problem of Finding Maximum Independent Sets can be reduced to PFOP. The PFOP is a particular case of the GPFOP.

As we then see, all the above problems can be reduced to to a Generalized Petrick Function Optimization Problem .

A role and importance of Complementation [?], Tautology [?] and Convertions from SOP to POS and vice versa are well known.

Decomposition of Boolean functions can be done in an algorithm that repeatedly applies Satisfiability, Tautology and Complementation. These operations are also of fundamental importance in most algorithms for Boolean minimization, factorization, and multi-level design.

Problem of Partial Satisfiability and its applications are discussed by K. Lieberherr [294, ?].

Many other reductions to the formulated above problems are discussed in papers [?, ?, ?, ?].

We will be using here a design automation system [?] in which many problems are first reduced to the few selected "generic" combinatorial optimization problems. These problems include the eight problems introduced above.

We will be looking to various methods to implement these generic combinatorial algorithms with the goal of finding ones that are as efficient as can be realistically achieved for NP-hard problems with current software and hardware technologies. Systolic processors and hardware accelerators will be presented in the last chapters of the book.

In this chapter we will systematically analyse the existing sequential algorithms for these and similiar problems. The results of this analysis will be later used to construct parallel algorithms.

5.2 Tree-Search Algorithms for Basic Boolean Problems

5.2.1 A General Characteristics of the Existing Approaches

The analysis of various algorithms for satisfiability can be found in the papers of Davis-Putnam [?], Goldberg, Purdom and Brown [188], Franco [167], Haralick [?], Lieberherr [294, 295], and Perkowski.

Although it is not generally acknowledged, the Boolean complementation problem [?, ?] is basically the same as the Problem 2 with all solutions looked for.

Various algorithms for solving the Covering Problem and Boolean minimization , by Slagle [?] and by Schmidt and Druffel [466], are basically the algorithms to solve the PFOP, and can be easily adapted to solve the GPFOP.

All algorithms from literature known to us are **sequential**.

All algorithms for the above , related strongly problems, can be basically divided into the following categories:

1. tree searching algorithms (for instance, those by Slagle, Schmidt, Purdom, and Davis).
2. array algorithms (for instance Quine-McCluskey algorithm to solve the covering problem).
3. transformational algorithms (for instance the original method to solve the Petrick functions).

The tree search can be differently organized and various tree searching strategies are used. We will use the terminology from Nilsson [360]. The tree is composed of nodes and arrows. New nodes are created from the nodes by application of operators. Arrows are labelled by operators. In our case operators correspond to literals or sets of literals. Various search strategies are used to expand trees, they use the cost and heuristic functions to select the

nodes for expansion and for the ordering of operators. We assume that the order of expanding the tree in the figures is from left to right.

The non-optimum algorithms can be divided into the following categories:

1. greedy algorithms,
2. random search algorithms,
3. incomplete tree-search algorrithms (they search only a subset of the solution space),
4. simulated annealing,
5. genetic algorithms.

In all these algorithms **three representations** of a General Petrick Function (GPF) are applied, as well as **three basic methods of branching**. This gives nine basic algorithm variants, out of which only few have been investigated in the literature. In this book we will apply only few of these choices, but the careful reader is encouraged to investigate all possible trade-offs.

Let us take the GPF (POS):

$$F1 \;=\; (a+\bar{b}+\bar{c}+d)(\bar{a}+d+e)(\bar{b}+\bar{c}+e)$$

The first representation is a list of terms. A term is a Boolean sum of literals. Each term can be also represented as a list. Function $F1$ can be then represented as a Lisp list:

$$F1 \;=\; ((a\,(b)\,(c)\,d)\,((a)\,de)\,((b)\,(c)\,e))$$

The same representation will be used for a Sum of Products Form:

$$F2 \;=\; a\bar{b}\;\bar{c}d \;+\; \bar{a}\;de \;+\; \bar{b}\;\bar{c}\;e$$

The possible methods of tree branching for our problems are the following:

1. nonbalanced binary tree, where each branching is done for a single variable and its negation. Various variables can be selected for branching in different nodes of the same depth of the tree [407] and additional rules are used for terms being single literals [131].
2. arbitrary number of successors in each node, branching is done according to the selected term in this node, all literals from this term lead to some successor nodes [73, ?]
3. method based on a standard tree of subsets of a set of all literals used in the function [?]. This method modifies the standard tree by removing literals that are not present in each current node of the tree.

Let us now concentrate on the first branching method only. The following decisions affect speed and quality of solutions obtained from this method.

1. How to select the branching variable?
2. What other rules (like Davis-Putnam) can be applied for creating the operators?
3. How to order the branches of the tree?
4. How to terminate the branches of the tree?

The modification of the first branching method is shown in Figure 5.1. Nodes of the tree correspond to the function and simplified functions that are created after substitutions The leafs of the tree are the solutions. Usually they correspond to products of operators along the branches. This method is used when all solutions are searched for.

A variable is selected for branching according to some rule. For instance, a variable can be selected that occurs most often (in both affirmative and negative forms) in the function. The branching with operators variable = 0 and variable = 1 is done by substituting in the function the values 0 and 1 for this variable, respectively. Two new nodes are created, in which the corresponding functions are simplified by removing terms with selected literal and removing the negations of the selected literal from other terms. Whenever a term being a single literal is created, it is immediately used for substitution, as in the Davis-Putnam procedure and the algorithms based on it.

Figure 5.1: Fig.1.

5.2.2 Selection of a Branching Variable

The rules for selecting a variable are:

1. Select a variable that occurs in most terms, in both positive and negative forms.
2. If there are more than one such variables selected in step 1 then select among them variables that occur in the shortest term. If there is only one variable selected in step 2 then return it.
3. If there are more than one variable selected in step 2 then select among them a variable v that maximizes the value of the function:

$$CV(v) = \frac{number\ of\ terms\ in\ which\ variable\ v\ occurs}{total\ number\ of\ literals\ in\ these\ terms}$$

4. Otherwise return random variable from step 3.

5.2.3 Additional Operator Selecting Rules

All literals from terms consisting of single literals are selected and no branching is done.

5.2.4 Ordering of Branches

For each variable v selected according to 5.2.2 we can apply one of two operators: $\{v\}$ and $\{\not(v)\}$ as the first operator in branching. The literal that occurs more often in the terms is applied. This leads to solutions being generated earlier when the depth-first search strategy with successors ordering is used, in which the successors of a node are ordered according to the above rule.

5.2.5 Termination of Tree Branches

First variant of branches terminating.
Two new additional rules are used:

1. When a function in a node consists of a single term, the solutions are created for all literals from this term. No branching is done and the current branch is terminated.
2. When all terms in a function include the same literals $L1, ...Ln$ and only single other literal each, then the solutions are created for $L1, ..., Ln$ and the product of the remaining literals from the terms. No branching is done and the current branch is terminated. When all terms in a function include the same literals $L1, ..., Ln$ and one or more from these terms include only those literals and no other literals, then the solutions are created for $L1, ..., Ln$. No branching is done and the current branch is terminated (this is unlike in the well-known procedures).

Second variant of branches termination.
In the optimization problems when only a single optimum solution or some optimum solutions are looked for with a minimum number of positive literals a speed-up can be obtained by using the rules:

1A. When a function in a node consists of a single term any literal from this term is selected to the solution. No branching is done and the current branch is terminated.

2A. When all terms in a function include the same literals $L1, ..., Ln$ and only single other literal each then the solution is created for $L1$. No branching is done and the current branch is terminated. When all terms of the function include the same literals $L1, ..., Ln$ and one or more from these terms include only those literals and no other literals then the solution is created for $L1$. No branching is done and the current branch is terminated.

Additionally, in this type of optimization search problems the cut-off rules are used to backtrack when the costs of partial solutions are equal or higher than the current minimum cost (cost of the best solution found until now).
This is presented in Figure 5.8.

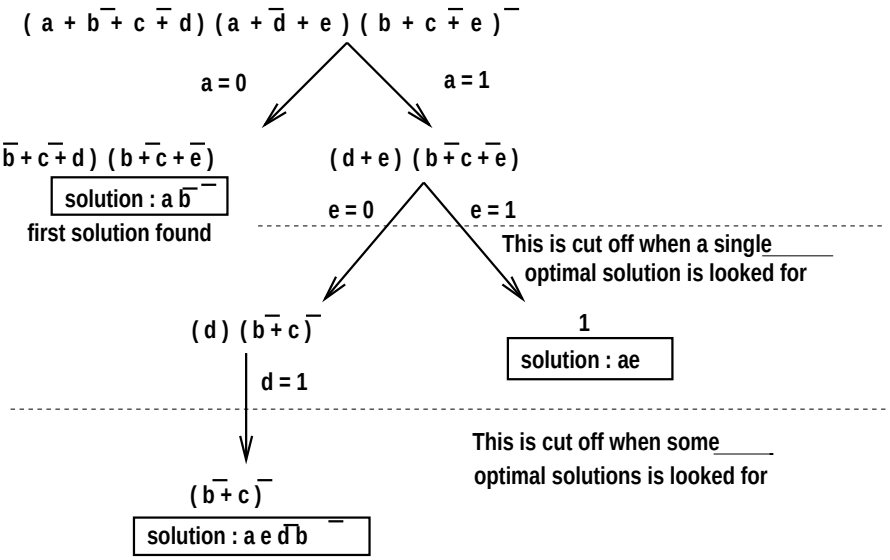


Figure 5.2: fig.2.ps MASIO

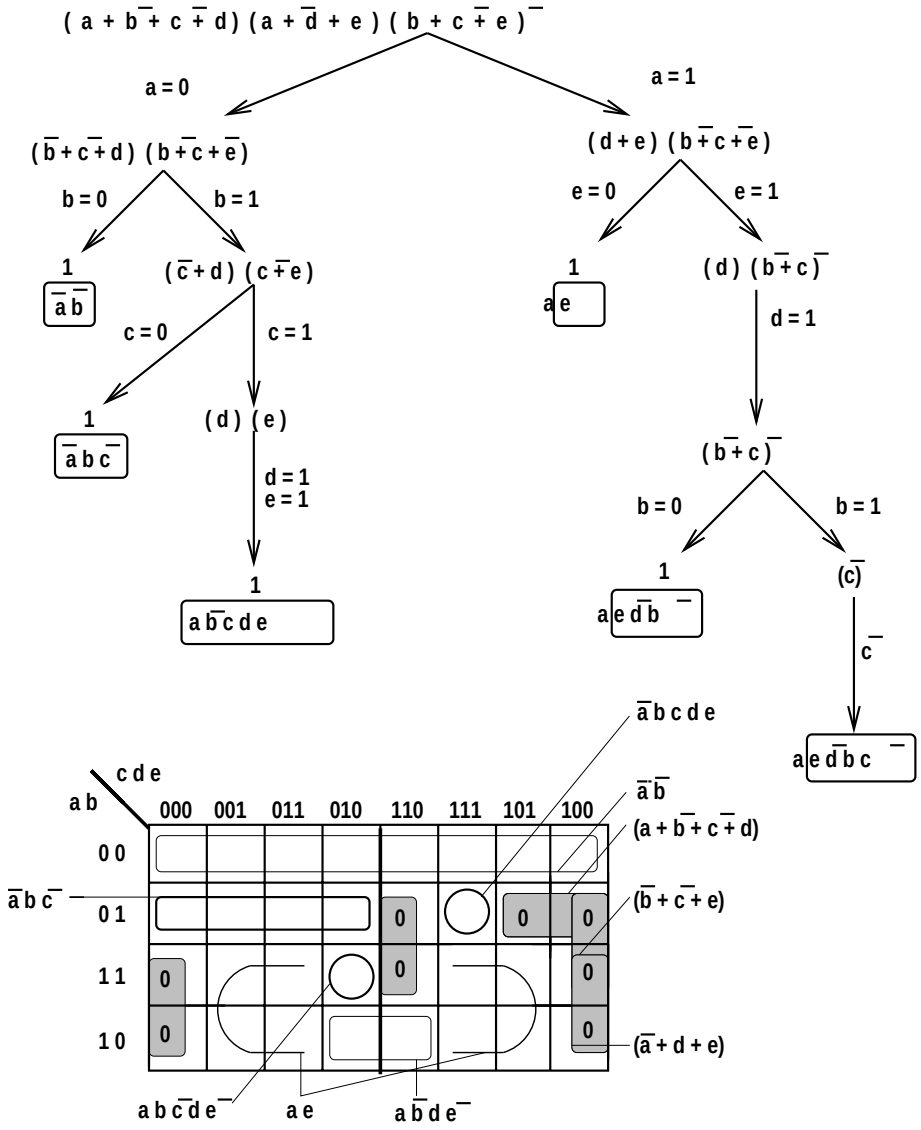
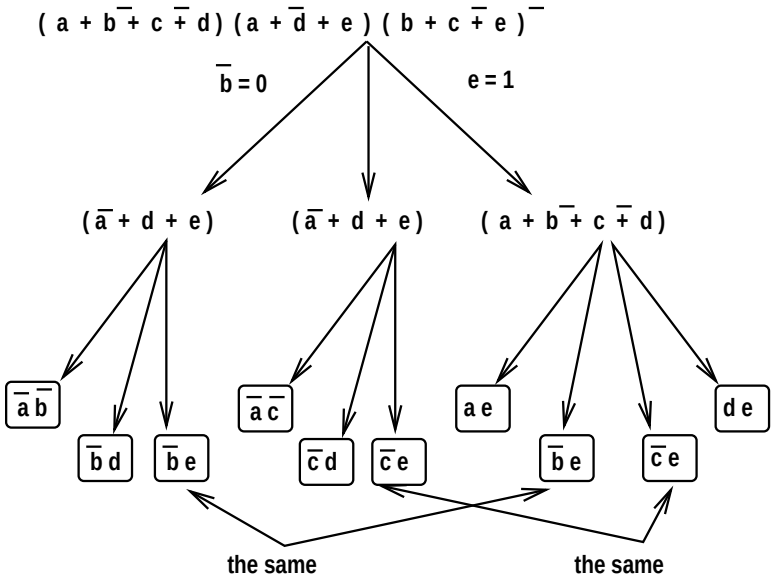


Figure 5.3: fig.x13.ps



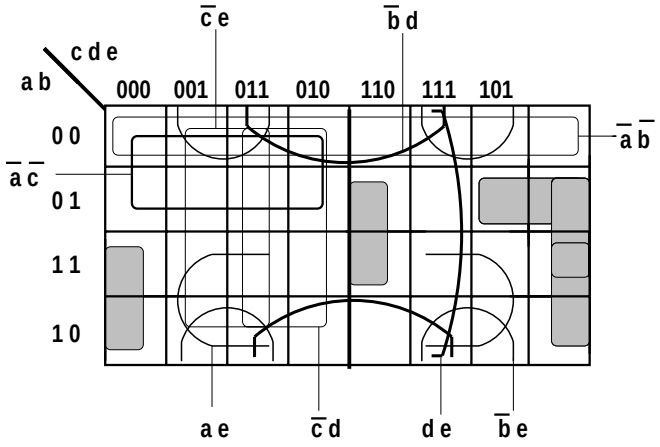


Figure 5.4: fig.x14.ps

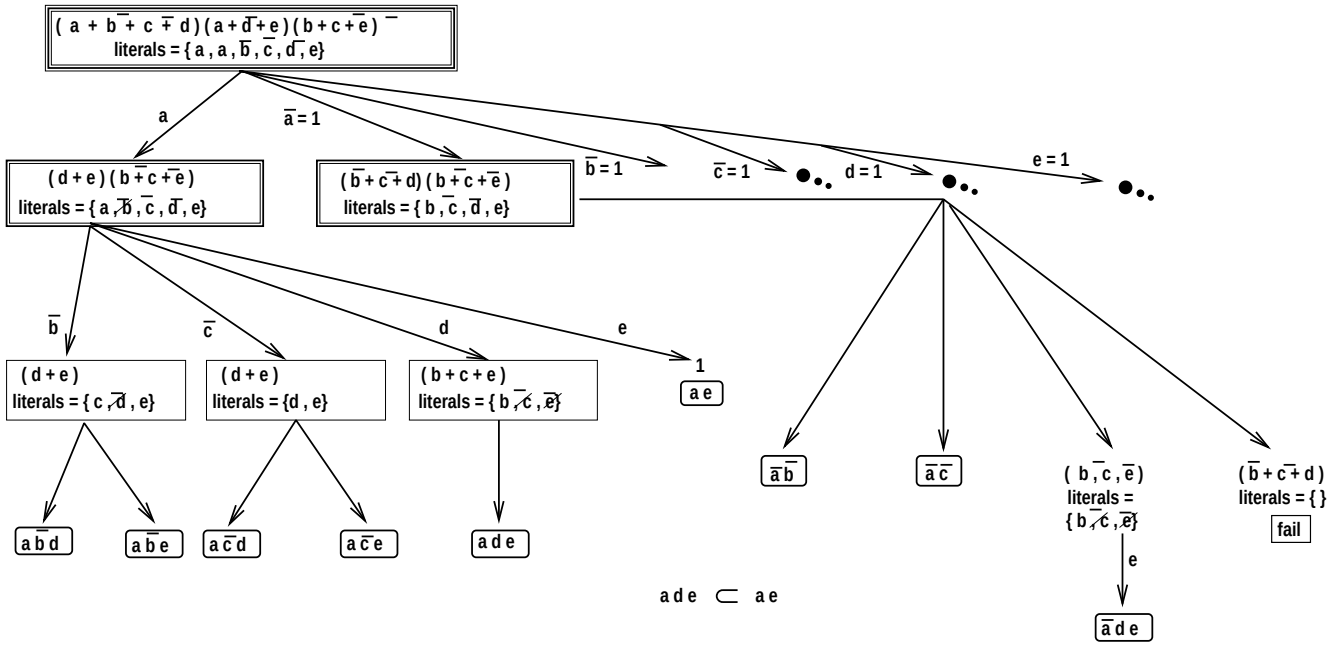


Figure 5.5: fig.x15.(COMP).ps

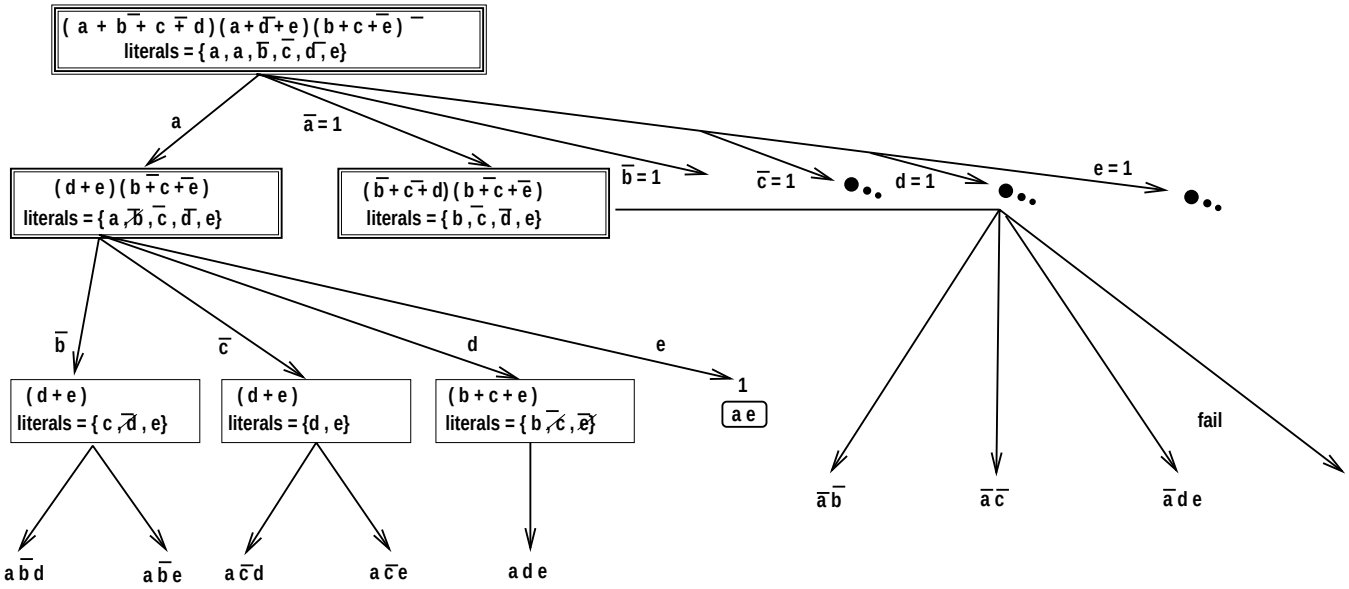


Figure 5.6: fig.x15(INC).ps

Figure 5.7: Fig.1b.

Figure 5.8: Fig.2.

5.2.6 Discussion

The created solutions (so-called implicants of $F1$) are all different. Comparison of the created products with the Karnaugh map of $F1$ from Fig. 5.7 allows us to observe the following properties of this branching method:

1. the complemented function has products which **are not neccessarily prime implicants**,
2. the implicants in the complemented function **are overlapping (are not disjoint)**.
3. the number of implicants is smaller than the number of all prime implicants of the function (all prime implicants are generated in many methods).

It results from the above that this branching method is well suited for complementation of Boolean functions and for finding of single solutions to the optimization problems. This method is not able to find all optimal solutions to such problems, however, it can produce a subset of quasi-optimal solutions, which in practice can be quite sufficient.

The advantage of this method is that some good solutions can be found with very limited search (using for instance the depth-first tree-search strategy [360], and the cut-off in the tree can be applied soon. Another advantage is that each solution is generated only once in the search process.

The main disadvantage of this method is that it may not produce the optimal solution to Problem 2, when the variables are selected in a wrong order. Although in the investigated by us practical examples the solutions were always optimal, they depended on heuristics. It does however provide optimum solution to problem 3, which has more practical applications.

The solution to problem 2 is with this branching variant not necessarily optimum since not all combinations of literals are created as branches of the tree. The solution to problem 3 is optimal since all combinations of positive literals are generated as branches.

The other branching methods are compared in [?]. Only the implementations of the above two variants of the first method will be discussed below.

5.3 Reductions

In this section we will show how some of the problems investigated by us can be reduced to other problems, in oreder to decrease the number of necessary generic programs in our library of useful CAD routines.

Reduction 1.

Let us assume taht we dispose an algorithm **Find_Solutions (Product_of_Sums, Number_of_Solutions, Type_of_Solutions)** that finds solutions (the solutions are products of literals) to **Product_of_Sums**. **Product_of_Sums** is a Boolean function in product of sums form (GPFOP). **Number_of_Solutions** and **Type_of_Solutions** are some user-specified parameters.

When

1. **Number_of_Solutions = all**, then all solutions are generated.
2. **Number_of_Solutions = all_optimal**, then all optimal solutions are generated.
3. **Number_of_Solutions = some_optimal**, then some optimal solutions are generated.
4. **Number_of_Solutions = one_optimal**, then one optimal solution is generated.

When

1. **Type_of_Solutions = literals**, the solutions to minimize the number of literals are looked for.
2. **Type_of_Solutions = positive_literals**, the solutions to minimize the number of positive literals are looked for.

When this parameter equals nil the type of the function is irrelevant.

To find a complementation of a Boolean function in a Sum of Products Form it is sufficient to find all solutions to a dual function. Therefore a Lisp definition is:

```
(defun Complementation (Sum_of_Products)
 (Find_Solutions (Dual Sum_of_Products) 'all nil))
```

With respect to the representation of Boolean functions shown above the finding of the dual function is trivial. It consists only in negating of all literals in the Sum of Products Form.

Example 5.1

$$f \;=\; ab \;+\; cd \;+\; \bar{a}\;\bar{b} \tag{5.1}$$

$$\begin{aligned}\bar{f} \;&=\; \overline{ab \;+\; cd \;+\; \bar{a}\;\bar{b}} \;=\; \overline{ab}\;\overline{cd}\;\overline{\bar{a}\;\bar{b}} \\ &= (\bar{a}+\bar{b})\;(\bar{c}+\;\bar{d})\;(a+b) \\ &= a\bar{b}\bar{c} + a\;\bar{b}\;\bar{d} + \;\bar{a}\;b\;\bar{c} \;+\; \bar{a}b\bar{d}\end{aligned} \tag{5.2}$$

Function f in SPF is represented as $((ab)(cd)((a)(b)))$. (Dual f) in PSF form is:

$$(((a)(b))((c)(d))(ab)) \tag{5.3}$$

When we generate all solutions for 5.3 using the first branch terminating variant and next the set of solutions generated by the program is treated as a SPF then the returned by it complement function will be the same as in 5.2.

Reduction 2.

To convert a Sum of Products to Product of Sums Form it is sufficient to find all solutions to this sum of products treated as a product of sums. The result is treated as a sum of products. In Lisp:

```
(defun Convert_Sum_of_Products (Sum_of_Products)
  (Find_all_solutions Sum_of_Products 'all nil))
```

Example 5.2

Let us take the function from the previous example: $f \;=\; ab + cd + \bar{a}\;\bar{b}$. Applying de Morgan Theorem to the right side of the formula 5.2 we get:

$$\begin{aligned}f \;&=\; \overline{a\bar{b}\bar{c}} \;\cdot\; \overline{a\;\bar{b}\;\bar{d}} \;\cdot\; \overline{\bar{a}\;b\;\bar{c}} \;\cdot\; \overline{\bar{a}b\bar{d}} \\ &=\; (\bar{a}+b+c)(\bar{a}+b+d)(a+\bar{b}+c)(a+\bar{b}+d)\end{aligned} \tag{5.4}$$

This is a PSF of function f. We treat the SPF of $f \;=\; ((a\;b)\;(c\;d)\;((a)\;(b)))$ as a PSF and find solutions with the first branch terminating variant. Later we treat the set of solutions as the product of sums. The same solution as in 5.4 is found.

Let us verify this. PSF corresponding to SPF is:

$$(a+b)(c+d)(\bar{a}+\bar{b}) \;=\; \bar{a}bc \;+\; \bar{a}bd \;+\; a\bar{b}c \;+\; a\bar{b}d \tag{5.5}$$

The result 5.5 must be treated as PSF, then we have:

$$(\bar{a}+b+c)(\bar{a}+b+d)(a+\bar{b}+c)(a+\bar{b}+d) \tag{5.6}$$

This is the same result as in 5.4.

Reduction 3.

Let us assume now that we dispose an algorithm `Satisfiability (Product_of_Sums)` that answers `YES` or `NO`, depending if there is a solution to a `Product_of_Sums`. Let us assume now that we want to check whether some SPF is a tautology. SPF is a tautology when its complement is zero or in other words when the answer to the Satisfiability Problem for the complement is `NO`.

```
(defun Tautology (Sum_of_Products)
 (not (Satisfiability (Dual Sum_of_Products))))
```

Example 5.3

	C1	C2	C3
a	1	0	0
$\bar{a}$	0	1	0
b	0	0	1
$\bar{b}$	1	0	0
c	0	0	0
$\bar{c}$	1	0	1
d	1	1	0
$\bar{d}$	0	0	0
e	0	1	1
$\bar{e}$	0	0	0

Table 5.1: Tabular Representation of Function $F1$

SPF of function f is:

$$f \;=\; ab + a\bar{b} + \bar{a}b \;+\; \bar{a}\bar{b} \tag{5.7}$$

f is represented as $((a\ b)\ (a\ (b))\ ((a)\ b)\ ((a)\ (b)))$. It can be easily checked that 5.7 is a tautology:

$$\begin{aligned} \bar{f} \;&=\; \overline{ab + a\bar{b} + \bar{a}b \;+\; \bar{a}\bar{b}} \\ &= \overline{ab}\ \overline{a\bar{b}}\ \overline{\bar{a}b}\ \overline{\bar{a}\bar{b}} \\ &= (\overline{a} \;+\; \overline{b})\ (\overline{a} \;+ b)\ (a \;+\overline{b})(a \;+\; b) \end{aligned} \tag{5.8}$$

The right side of 5.8 can be calculated by

$$(DualSum\_of\_Products) \;=\; (((a)\ (b))\ ((a)\ b)\ (a\ (b))\ (a\ b))$$

This is given as an argument to program `Satisfiability`. Now searching the tree for 5.8 shows that there is no product of literals that satisfies this function. The answer for function from 5.8 produced by the `Satisfiability` program will be `NO`. Therefore the answer to the `Tautology` will be `YES`.

In conclusion, we will need only three programs to solve all eight problems:

1. `Satisfiability(Product_of_Sums)`,
2. `Find_Solutions(Product_of_Sums,Number_of_Solutions,Type_of_Solutions)`,
3. `Partial_Satisfiability(Product_of_Sums,Number_of_Solutions,Type_of_Solutions)`.

5.4 Other structures

Let us take the GPF (POS):

$$F1 \;=\; (a + \bar{b} + \bar{c} + d)(\bar{a} + d + e)(\bar{b} + \bar{c} + e)$$

The first representation is a list of terms. A term is a Boolean sum of literals. Each term can be also represented as a list. Function $F1$ can be then represented as a Lisp list:

$$F1 = ((a\,(b)\,(c)\,d)((a)\,de)((b)\,(c)\,e))$$

Its variants use computer words or Boolean cubes [495] to represent terms.

The second representation uses an array of symbols 0 and 1 to describe the GPF: $F1 =$

This representation uses often arrays of binary words to store rows or columns of the array. The variant of this representation uses half the number rows, but more symbols to be stored in an array are now required. Two bits per symbol (0, 1, X, auxiliary E) are used.

$F1 =$

	C1	C2	C3
a	1	0	X
b	0	X	1
c	0	X	0
d	1	1	X
e	X	1	1

Table 5.2: Second Method for Tabular Representation of Function $F1$

Figure 5.9: The second branching method

The third representation uses lists corresponding to rows of the above arrays:
$La = \{C1\}$,
$L\bar{a} = \{C2\}$,
$Lb = \{C3\}$,
$L\bar{b} = \{C1\}$,
$Lc = \{\}$,
$L\bar{c} = \{C1,C3\}$,
$Ld = \{C1,C2\}$,
$L\bar{d} = \{\}$,
$Le = \{C2,C3\}$,
$L\bar{e} = \{\}$,
The possible methods of tree branching are the following:

1. irregular binary tree, where branching is done for a variable and its negation. Various variables can be selected in different nodes of the tree on the same search depth [Purdom, Haralick],

2. arbitrary number of successors in each node, branching is done according to the selected term in this node , all literals from this term lead to successors [Breuer, Perkowski],

3. standard tree of subsets of a set of all literals used in the function [Perkowski]. This method modifies the standard tree by removing literals that are not present in each current node of the tree.

The **first branching method** is shown in Fig. **??**. Whenever terms of single literals are created, they are immediately used for substitution, as in the Davis-Putnam procedure and all its successors. The created solutions (implicants of $F1$) are all different. Comparing the created products with the Karnaugh map of the $F1$ function (Fig. **??**) permits us to note two properties of this branching method:

1. the complemented function has products which **are not prime implicants,**

2. the implicants in the complemented function **are not overlaping.**

It results from the above that this branching method is well suited for complementation of Boolean function and for finding of single solutions to problems. It is not able to find all optimal solutions, however it can produce a subset of quasi-optimal solutions, which in practice can be quite sufficient.

The advantage of this method is that some good solutions are found with small search (using for instance the depth-first tree-search strategy (Nilsson, [360]) and the cut-off in the tree can be applied soon. Another advantage is that each solution is generated only once.

The main disadvantage of this method is that it can produce not the optimal solution, when variables are selected in wrong order. Although in the investigated by us practical examples the solutions were always optimal, they depended on the heuristic. the optimum solution a is found when variable a is selected in the first level (it is selected because variable a occurs most often). When variable c is selected on the first level the best solution found has two literals and is not optimum.

The **second branching method** is presented in Figure 5.9. At each node of the tree one term is selected

1. according to some heuristic,

2. randomly,

3. as the first one.

The literals from this term are taken for branching. This method can incorporate not only Davis-Putnam heuristics but also many methods used to solve the covering problem, like dominance of rows or symmetry. As we see in the corresponding Karnaugh map (Fig. ??) the created products are prime implicants of the function and they also overlap. The method is then good to generate all prime implicants while complementing and to generate all optimal solutions to a function. The disadvantage of this method is that some solutions are generated many times (like !be in our example). The advantage of this method is that it can generate **all optimum solutions.**

The **third branching method** uses the standard method of generating subsets of a set of all literals (see Fig. ??). The advantage of this method is that each solution (implicant) is generated only once. However implicants included into other implicants are generated, which makes this method applicable only if the implicants with costs higher than the cost of the actually minimum solution are cutted-off. This method can generate some optimum solutions. It cannot be used to generate all optimum solutions or to complement a Boolean function.

The approximate methods expand some subset of the described above trees.

The greedy algorithms find one depth-first path in the tree, and if necessary, iterate.

The random search algorithms find single random depth-first path and (sometimes) iterate.

The incomplete search algorithms search with some heuristic strategy, that searches only a subspace of the entire solution space. The strategies include:

1. depth-first with limited number of backtracks,
2. ordered-search with not-admissible quality function (Nilsson, [360]),
3. branch-and-bound with no-admissible quality function,
4. any combination of the above.

While designing a parallel hypercube algorithms two problems must be taken into account:

1. general problem decomposition and organization of parallelism,
2. algorithm for each node.

We have designed several general parallel architectures for our three problems. They include :

1. tree,
2. inverse tree,
3. ring,
4. 2-dimensional mesh,
5. 3-dimensional mesh with tree,
6. trellis tree.

Although all of them show some interesting properties for parallel processing, some are better suited for systolic processors and MIMD architectures with smaller grain than iPSC and many more processors [Perkowski].

For current available size of iPSC computer we selected two architectures:

1. ring,
2. tree.

First architecture can be used to generate quasi-optimum solutions only. The advantage of this architecture is a nice combination of randomness, problem decomposition and pipelining on the top level and vector processing on the bottom level. Let us note that extremely large functions can be optimized with this approach. Let us assume that each node can have 500 kWords memory for partial function. Assuming that we use first representation and literals encoded as numbers are placed in successive words of a memory, we can store in it a function with for instance 1000 terms and 2000 literals assuming density 0.25. The entire function in 16-processor configuration can have then 16,000 terms.

Assuming the bit representation of a function we can store for instance 5000 terms of 3000 variables (60 bits in a word, 2 bits per variable) in every single node.

The second architecture can be used both for quasi-optimum and absolutely optimum solutions. However, only the functions which are 16 times smaller than in the ring architecture can be solved, taking the memory restriction alone.

5.5 The Parallel Ring Algorithm

The given below algorithm solves approximately Problems number 2 and number 3. The only difference is how the cost function is calculated. If there is no solution to the problem algorithm will loop infinitely. In some cases a program will loop, even if the solution exists (it depends on the quality of random number generator). Since we are mostly interested in solving the practical problems where many solutions exist, and not the decision problems, this approach is acceptable. Also, as it results from theoretical investigations (Franco [167], Lieberherr [294], even for the decision problems this approach is superior for large functions (small functions can be solved with current sequential algorithms).

Let us assume that there are n terms and m variables altogether. The number of processors is k. The processors are configured to a ring with k processors. Initially a set of terms is splitted to processors: a set of terms in each processor of number me is denoted by $FUNCTION(me)$.

The ring architecture algorithm is the following.

```
1. Load FUNCTION(me) ;
2. ;;;;;; Comment: New_Process
   Find good solution SOLUTION(me) to FUNCTION(me) with
       algorithm optimize(FUNCTION(me));
   ;;;;;;  this includes random choice
   number(SOLUTION) := me ;
   COST(SOLUTION) := COST(SOLUTION(me)) ;
   go_to 7 ;
3. Receive a message from Predecessor(me);
   The message has a structure:
   TRIPLE :=
    [ number of SOLUTION, number_of_elements, SOLUTION, COST of SOLUTION ];
        ;;;;;   SOLUTION is a partial solution  SOLUTION from Predecessor(me) ;
        ;;;;;;  (This is a set of literals).
4. Substitute assignment of values from SOLUTION to  FUNCTION(me)
   FUNCTION(me) := substitute_values( FUNCTION(me) , SOLUTION(me) ) ;
5. If there is a term in FUNCTION(me)
   that is equal logically 0 then go_to 2 ;
6. Find good solution SOLUTION(me) to FUNCTION(me) ;
   Receive minimum cost from all hypercube neighbors ;
   MINIMUM_COST := min (MINIMUM_COST, minimum costs from nodes) ;
   Send MINIMUM_COST to all hypercube neighbors ;
   COST(SOLUTION) := COST + COST(SOLUTION(me)) ;
7. If COST(SOLUTION) >= MINIMUM_COST
   then go_to 2 ;
8. If number(SOLUTION) = Successor(me)
    then
  begin
    send SOLUTION to cube manager
    send COST(SOLUTION) to all hypercube neighbors ;
go_to  3 ;
  end
    else
  begin
send message TRIPLE = [ number(SOLUTION), SOLUTION(me), COST(SOLUTION)]
       to processor  Successor(me);
go_to 3
      end
```

Example 5.4

Let a Petrick function be the following:

$$PF = (a+\bar{b}+c+d)(a+b+\bar{c})(b+c+d+\bar{e})(d+e+f)(a+\bar{b}+\bar{e})(\bar{c}+d+e)$$

After initial splitting the functions in processors are:

$$FUNCTION(1) \ = \ (a + \bar{b} + c + d)(a + b + \bar{c});$$

$$FUNCTION(2) \ = \ (b + c + d + \bar{e})(d + e + f);$$

$$FUNCTION(3) \ = \ (a + \bar{b} + \bar{e})(\bar{c} + d + e).$$

The processors are connected in a ring as in Figure **??**. The first three "macro-pulses" of the ring architecture are shown in Fig. **??**. The architecture is asynchronous and the transfer of partial solution to the successor processor in ring occurs when it is free. The speed of the pipeline depends then on the speed of the slowest actually processor. The processors shall find solutions in approximately the same time. The splitting of the function into functions of approximately the same complexity and selection of the algorithms in nodes must ensure this.

```
TIME = 1

me=1
 FUNCTION(1) =  (a + !b + c + d)(a + b + !c);
 SOLUTION(me) = {a}
 COST(me) = 1


me=2
 FUNCTION(2) =  (b + c + d + !e)(d + e + f);
 SOLUTION(me) = {d}
 COST(me) = 1


me=3
 FUNCTION(3) =  (a + !b + !e) (!c + d + e).
 SOLUTION(me) = {a,d}
 COST(me) = 2




TIME = 2









me=1
 FUNCTION(1) =  (a + !b + c + d)(a + b + !c);
 TRIPLE = [3, {a,d}, 2]
 FUNCTION(1) = 1
 COST = 2
 TRIPLE = [3, {a,d}, 2]


me=2
 FUNCTION(2) =  (b + c + d + !e)(d + e + f);
 TRIPLE = [1, {a}, 1]
```

```
 FUNCTION(2) =  (b + c + d + !e)(d + e + f);
 SOLUTION(me) = {d}
 COST(me) = 1
 COST = 2
 TRIPLE = [1, {a,d}, 2]


me=3
 FUNCTION(3) =  (a + !b + !e) (!c + d + e).
 TRIPLE = [2, {d}, 1]
 FUNCTION(3) = (a + !b + !e)
 SOLUTION(me) = {!b}
 COST(me) = 1
 COST = 2
 TRIPLE = [2, {d,!b}, 2]




TIME = 3




me=1
 FUNCTION(1) =  (a + !b + c + d)(a + b + !c);
 TRIPLE = [2, {d,!b}, 2]
 FUNCTION(1) = (a + !c)
 SOLUTION(me) = {a}
 COST(me) = 1
 TRIPLE = [2, {d,!b,a}, 3]
 SEND TO CUBE MANAGER !


me=2
 FUNCTION(2) =  (b + c + d + !e)(d + e + f);
 TRIPLE = [3, {a,d}, 2]
 FUNCTION(2) =  1
 SOLUTION(me) = {}
 COST(me) = 0
 COST = 2
 TRIPLE = [3, {a,d}, 2]
 SEND TO CUBE MANAGER !


me=3
 FUNCTION(3) =  (a + !b + !e) (!c + d + e).
 TRIPLE = [1, {a,d}, 2]
 FUNCTION(3) =  1
 SOLUTION(me) = {}
 COST(me) = 0
 COST = 2
 TRIPLE = [2, {a,d}, 2]
 SEND TO CUBE MANAGER !
```

Fig.4.

Suppose that the solution from processor 1 arrived first in Cube Manager. It is printed and its cost 3 is stored.

Assume now that next the solution from processor 3 arrives. Its cost 2 is smaller than the actual minimum cost 3, so it is printed and its cost is stored. When the solution from the second processor arrives its cost is the same as the previous cost. Depending on the algorithm of the Cube Manager (this algorithm is straightforward and is not discussed here) this solution is printed or not. The Cube Manager prints either all even better solutions or all solutions not worse than ones previously generated. Depending on the selected problem's type it also selects the final solution or the set of solutions. In the case of decision problem it answers "yes" when a first solution is found or answers "no" with certain calculated probability.

The process of solving partial problems in processors and transferring partial solutions to the successors in ring is infinite. The solutions in each processor are generated with random algorithms, so a large subspace of the space of all solutions is investigated and the same solutions are rarely generated for large examples.

5.6 Vector Processor Algorithms for Auxiliary Functions

Function `substitute_values` simplifies $FUNCTION(me)$ by substituting values from $SOLUTION(me)$.

Algorithm of this function is the following. Operations on sets are assumed.

By $NEGATED(set)$ we denote negations of literals from set.

```
defun substitute_values(FUNCTION(me) , SOLUTION(me))
for i := 1 to card(FUNCTION(me)) do
   begin
    term := FUNCTION(me) [i]
    if SOLUTION(me) ∩ term ≠ nil
    then remove term from FUNCTION(me)
    term := term - NEGATED(SOLUTION(me)) ;
    if term = nil then return "no solution"
   end
```

Assuming that each term is stored in *number_of_words* words of the memory the above algorithm is rewritten to the given below form. Now each operation like ∩ and - are executed in one pulse, because the set operations are replaced with operations on 60 bit (in our case) Boolean words.

```
begin
j := 0 ; ;;;;; counter of new terms in FUNCTION(me)
for i := 1 to card(FUNCTION(me)) do
   begin ; loop for single term
      FLAG_NO := 0 ;
   for word := 1 to number_of_words do
      begin
     term[word] := FUNCTION(me) [i] [word]
     if term[word] ∩ SOLUTION(me)[word] ≠ 00...0
        ;;;;;; product done in one pulse as bit-by-bit and of words
      then go_to next_term ;
    term[word] := term[word] - NEGATIONS_FROM_SOLUTION [word]
          ; executed in one pulse
        if term[word] ≠ 00...0 then FLAG_NO := 1
   end ;
  if FLAG_NO = yes then return "no solution"
     begin
          for word := 1 to number_of_words do
              begin
            FUNCTION(me) [j] [word] := term [word]
                ;;;;;;; rewriting back a term to array FUNCTION(me)
              end ;
      j := j + 1
   end
   next_term
   end ; ;;;;;; loop for term
   j is new card(FUNCTION(me));
end.
```

Figure 5.10: Abstract structure of inverted binary tree

Figure 5.11: Shuffle-exchange architecture

5.7 Vector Processor Algorithm for Quasi-Optimal Optimization

As discussed in one of previous sections, there are many algorithms to find approximate solution to $FUNCTION(me)$.

We want to investigate and compare the three branching schemes and the three representations for the given three problems. We will also investigate various heuristics and tree-search strategies.

For instance, the quasi-optimum algorithm for the first branching method is:
defun optimize (FUNCTION, nil)

0. If $FUNCTION = 1$ then return nil; If $FUNCTION = 0$ then return 'false ;
1. Select a variable V according to some heuristics (the heuristics are: a variable that occurs most often, a variable that often occurs in short terms, etc) ;
2. $variant1 := substitute(FUNCTION, V = 0)$
3. $variant2 := substitute(FUNCTION, V = 1)$
4. $solution1 := optimize(variant1) \ \cup \ \{\bar{V}\}$;
5. $solution2 := optimize(variant2) \ \cup \ \{V\}$;
6. return better of solutions $solution1$ and $solution2$.

This recursive algorithm can be rewritten a to stack algorithm. If we limit the number of backtracks in the stack algorithm or apply the random choice in it - the fast approximate solutions are generated. Other serial algorithms for tree-search to solve our three problems are given in [Perkowski].

5.8 The Parallel Tree Search Architecture

The parallel tree search architecture uses all processors for tree search.

The upper levels of the tree are expanded in each processor with a breadth-first search strategy until the list OPEN with n nodes is created. Next each processor with number me takes me-th element from this list and expands its own tree with this node as an initial node. Arbitrary search strategy and branching method can be used. The solutions found in each processor are communicated to a Cube Manager which selects the best one. This method applies to all branching methods described above, all representations of functions and all search strategies in nodes.

5.9 Transformational Algorithms

The transformational algorithms use the levelized Boolean multiplication and simplification of a function. They use abstract structure of inverted binary tree (Fig. 5.10). This tree can be mapped differently into hypercube. First method uses all processors to store next levels of the tree in a kind of shuffle-exchange architecture (Fig. 5.11). The second method uses inverse binary tree. In the first architecture parts of solution are transfered to the next level when they are calculated. When any solution is found its cost is calculated and transmitted to all processors. All transitiory solutions that have larger costs are removed. Partial result of multiplication is transfered when all multiplications are done In this approach the solutions to partial process are transmitted when all they have been found.

In the second approach whenever single solution is found in leaf processors it is transmitted to its father.

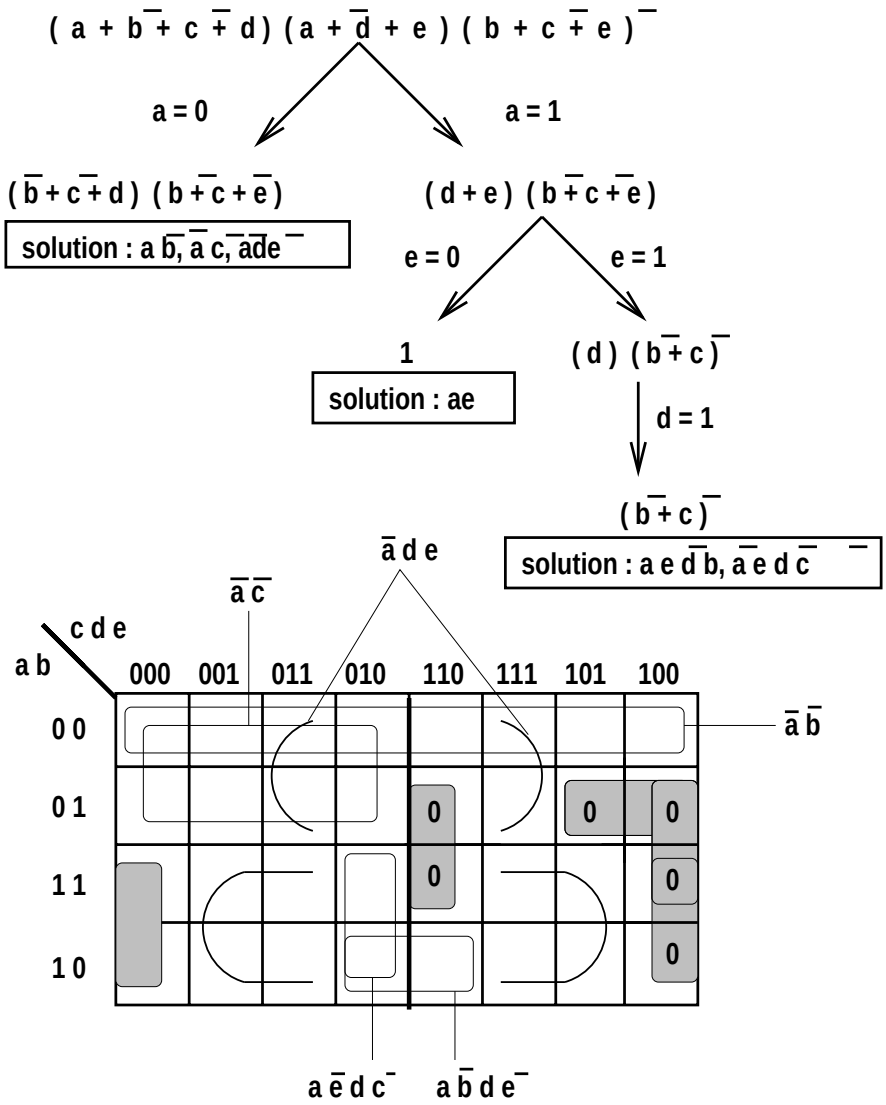


Figure 5.12: fig.1.ps MASIO Satisfiability 1

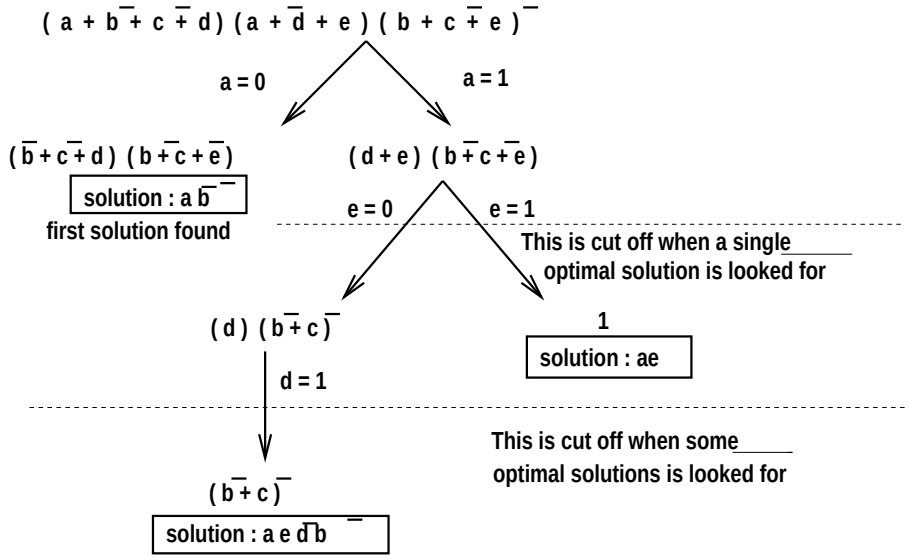


Figure 5.13: fig.2.ps MASIO Satisfiablity 2

5.10 Figures to Satisfiability

5.11 Tautology

5.12 Set Covering: Binate and Unate

5.13 Maximum Clique

5.14 Lisp Logic Simulator by Dave Mattson extended to MV

Chapter 6

Two Powerful Lisp Ideas

6.1 Functional Arguments

6.1.1 Functionals

Functions in Lisp always evaluate its arguments. The exceptions are the special forms (FEXP or FSUBR) for which the arguments can be evaluated *inside the function*, according to the requirements of the programmer. Particularly, they can be not evaluated at all, or only part of them is evaluated.

When a function or a functional expression is an argument of a function - they cannot be evaluated. They must then be given as arguments of QUOTE, FQUOTE or FUNCTION.

When only the interpreter is used, we shall always (with the exceptions specified below) use the form QUOTE. FQUOTE shall be used for functional arguments when the compiler is used.

```
(FQUOTE S)
NORMAL, FSUBR.
```

FQUOTE causes that its argument is compiled by the computer. FQUOTE should then appear only before λ-expressions of compiled functions.

FUNCTION.

```
(FUNCTION FUNCTION)
PSEUDOFUNCTION, FSUBR.
```

FUNCTION is a special form of one argument. The argument is the name of a function or a λ-expression. Form FUNCTION is evaluated very longly, QUOTE and FQUOTE are then usually used instead. FUNCTION is then necessary only in cases when the function includes free variables which change values during the time between the definition and the execution of the function. FUNCTION is used in cases when the environment must be prepared for a functional argument being passed to another function. Its value is the so-called FUNARG *expression.*

Example 6.1

```
(FUNARGTEST (LAMBDA (LIST FUN)
     (COND ((NULL (CDR LIST)) (FUN (CAR LIST)))
           (T (FUNARGTEST (CDR LIST)
                (CONS (CAR LIST) (FUN (CAR LIST))))) )))
```

Let us notice that function FUNARGTEST includes a free variable LIST, which changes itself from the moment of defining and the moment of execution. FUNCTION must be then applied rather than FQUOTE.
Test on your computer.

```
(FUNARGTEST '((A B) (C D)) (FUNCTION CAR))
(FUNARGTEST '((A B) (C D)) (FQUOTE CAR))
```

6.1.2 Standard Functionals

We will present now the standard functionals of Lisp:
MAP, MAPLIST, MAPCON, SEARCH, SASSOC.

```
MAP, MAPLIST, MAPCON, SEARCH, SASSOC
(MAP LIS FNEXP)
PSEUDOFUNCTION, SUBR, FUNCTIONAL.
```

`MAP` applies a one-argument functional expression `FNEXP` to list `LIS`, next to `(CDR LIS)` and to subsequent `CDR` of this list until the list will reduce to the single atom (usually NIL).

The value of `MAP` is `NIL`. `MAP` is *always* used because of its *side effects* rather than its value.

```
(DEF (MAP (LIS FNEXP)
       (PROG ()
         A (COND ((ATOM LIS) (RETURN LIS)))
           (FNEXP LIS)
           (SETQ LIS (CDR LIS))
           (GO A) )))
```

Example 6.2

```
EVAL:

    ((LAMBDA (L) (MAP L 'PRINT)) '(CAT RUNS (DOG BOY)) )

VALUE:

    (CAT RUNS (DOG BOY))
    (RUNS (DOG BOY))
    ((DOG BOY))
    VALUE:
    NIL
```

Check on your Lisp: `(MAP '(A . B) 'PRINT)`, and also: `(MAP '(2.3) 'PRINT)`

MAPLIST.

```
(MAPLIST LIS FNEXP)
NORMAL, SUBR, FUNCTIONAL.
```

`MAPLIST` does the same as `MAP` but its value is the list of values of the one-argument functional `FNEXP` applied to the subsequent `CDR` of `LIS`.

The value of `(MAPLIST LIS FNEXP)` is then equivalent to

```
    (LIST (FNEXP LIS) (FNEXP (CDR LIS))
                      .... (FNEXP (CD....DR LIS)))

    (DEF
        (MAPLIST (LIS FNEXP)
           (COND ((NULL LIS) NIL)
                 (T (CONS (FNEXP LIS)
                          (MAPLIST (CDR LIS) FNEXP))) )))
```

Example 6.3

```
(DEF (SQUARECAR (X) (TIMES (CAR X) (CAR X)) ))

((LAMBDA (J) (MAPLIST J 'SQUARECAR))
            '(1 2 3 4 5)) ==> (1 4 9 16 25)

((LAMBDA (J) (MAPLIST J 'CDR)) '(A B C))
                          ==> ((B C) (C) NIL)

(MAPLIST '(A B C) (FQUOTE (LAMBDA (X) (CONS X NIL))))
                ==> (((A B C)) ((B C)) ((C)))
```

MAPCON.

```
(MAPCON LIS FNEXP)
NORMAL, SUBR, FUNCTIONAL.
```

Function `MAPCON` joins the results of the application of one-argument functional expression `FNEXP` subsequently to `LIS`, `(CDR LIS)`, etc., in such a way that NIL which terminates the list being the value of the subsequent application of `FNEXP` (but not the last one) is replaced with the value of this expression `FNEXP` in its next call. The value of `MAPCON` is the list created in such a way. When list `LIS` is empty, `MAPCON` takes value NIL. It results from the operation of `MAPCON` that each value returned by `FNEXP` must be a list, because otherwise the error would occur during an attempt of concatenation.

```
(DEF (MAPCON (LIS FNEXP)
       (COND ((NULL LIS) NIL)
             (T (NCONC (FNEXP LIS)
                       (MAPCON (CDR LIS) FNEXP))) )))
```

Example 6.4

```
(MAPCON '(A B C) (LAMBDA (X)(CONS X NIL))) ==> ((A B C)(B C)(C))

(MAPCON '(A B C) (LAMBDA (X)(LIST (LENGTH X)))) ==> (3 2 1)
```

MAPCAR.

```
(MAPCAR LIS FNEXP)
NORMAL, EXPR, FUNCTIONAL.
```

`MAPCAR` is similar to `MAPLIST`. It creates a list from the values of the one-argument functional expression `FNEXP` applied subsequently to the elements of list `LIS`. It differs from `MAPLIST` in that it applies `FNEXP` to each element of `LIS`, i.e. to the `CAR` of the structure to which `FNEXP` is applied in `MAPLIST`.

```
    (DEF (MAPCAR (LIS FNEXP)
         (COND ((NULL LIS) NIL)
               (T (CONS (FNEXP (CAR LIS))
                        (MAPCAR (CDR LIS) FNEXP)) ))))
```

Example 6.5

```
((LAMBDA (J) (MAPCAR J (FQUOTE (LAMBDA (L)
      (COND ((NUMBERP L) (TIMES L L)) (T L)) ))) )
            '(A 1 B 2 C 3)) ==> (A 1 B 4 C 9)

(MAPCAR '(A B C) (FQUOTE (LAMBDA (X) (CONS X NIL))))
                            ==> ((A) (B) (C))
```

MAPC.

(MAPC LIS FNEXP)
PSEUDOFUNCTION, E

MAPC operates as MAPCAR with respect to effect and as MAP with respect to value. MAPC applies functional expression FNEXP to each of the elements of LIS. The value of MAPC is the last element of LIS, usually NIL.

```
(DEF (MAPC (LIS FNEXP)
     (PROG ( )
        A (COND ((ATOM LIS) (RETURN LIS)))
          (FNEXP (CAR LIS))
          (SETQ LIS (CDR LIS))
          (GO A) )))
```

Check on your computer: `(MAPC '(A.B)  'PRINT)`

Example 6.6

```
((LAMBDA (L) (MAPC L 'PRINT)) '(I AM A TEACHER))
```

causes the printout

```
I
AM
A
TEACHER
VALUE
NIL
```

SEARCH.

(SEARCH LIST FNEXP1 FNEXP2 FNEXP3)
NORMAL, SUBR.

SEARCH applies FNEXP1 to LIST and successive CDR's of LIST until it happens that the value of FNEXP1 is true. If this condition occurs, then SEARCH applies FNEXP2 to the remainder of LIST at that time. If the entire list is exhausted without any application of FNEXP1 giving a true value, then FNEXP3 is applied to NIL. SEARCH is therefore used to apply some function to a portion of the list depending on a condition which must be met by some elements of the list. Each FNEXP must be a function of only one argument.

```
(DEF
   (SEARCH (LIS FNEXP1 FNEXP2 FNEXP3)
     (COND ((NULL LIS)
            (FNEXP3 NIL))
           ((FNEXP1 LIS)
            (FNEXP2 LIS))
           (T (SEARCH
                 (CDR LIS) FNEXP1 FNEXP2 FNEXP3 )) )))
```

Example 6.7

```
(SEARCH '(A B C) 'NUMBERP 'ADD1 'LENGTH) ==> 0

(SEARCH '(A B C) 'CAR 'CDR 'GENSYM) ==> (B C)

(SEARCH '(A B C) 'CDR 'CAR 'GENSYM) ==> A

(SEARCH '(A B C) 'NUMBERP 'ONEP 'NULL) ==> *T*

(SEARCH '((A.1) (B.2) (C.3))
   '(FQUOTE (LAMBDA (X) (EQ (CAAR X) 'B) ))
   '(FQUOTE (LAMBDA (X) (CDAR X)))
   '(FQUOTE (LAMBDA (X) (ERROR '(SEARCH FAILURE)))) ) ==> 2
```

SASSOC.

```
(SASSOC S LIS FNEXP)
NORMAL, SUBR, FUNCTIONAL.
```

This function searches list LIS of non-atoms and selects from it the element whose CAR is EQ to S. If such an element exists, then the pointer to it is returned as a value of SASSOC. If LIS is empty or if there is no such element in LIS, then the zero-argument functional expression FNEXP is executed and its value is returned as the value of SASSOC. Function SASSOC is applied for selecting from the list of dotted pairs the pair, whose first element is atom S, but can be also applied for selecting analogously the elements from the list of lists. If S is not an atom but LIS does not include a list or dotted pair with S at the beginning, then FNEXP is executed.

```
    (DEFINE '(
       (SASSOC (LAMBDA (S LIS FNEXP)
         (COND ((NULL LIS) (FNEXP))
               ((EQ (CAAR LIS) S) (CAR LIS))
               (T (SASSOC S (CDR LIS) FNEXP)) )))
               ))
```

Example 6.8

```
(SASSOC 'A '((A B) (A C) (B D)) 'GENSYM) ==> (A B)

(SASSOC 'A '((B C D) (A G H)) 'TERPRI)   ==> (A G H)

(EVAL '(SASSOC 'A '((A1 B1) (A2 B2)) '(NO A IN LIST)))
                                        ==> (NO A IN LIST)
```

It is sometimes useful to define function ASSOC which is like SASSOC but uses EQUAL instead of EQ.

6.1.3 Examples of Programs

SUBSTOP replaces SEX1 with SEX2 on the top level of list LIS.

```
(SUBSTOP (LAMBDA (SEX1 SEX2 LIS)
     (PROG (C)
        (MAP LIS (FQUOTE (LAMBDA (SUBLIS)
             (COND ((NULL SUBLIS) NIL)
                   ((EQUAL SEX1 (CAR SUBLIS))
                     (RPLACA SUBLIS SEX2))
                   (T SUBLIS)) )))
        (RETURN LIS) )))
```

REMOVELAST cuts-off the last element from list

```
(REMOVELAST (LAMBDA (LIST)
  (COND ((NULL LIST) NIL)
        ((NULL (CDR LIS)) NIL)
        (T (PROG ()
            (MAP LIST (FQUOTE (LAMBDA (SUBLIS)
              (COND ((NULL (CDDR SUBLIS)
                     (RPLACD SUBLIS NIL))
                    (T (SUBLIS)) )))
            (RETURN LIS)  )))  ))))
```

MAKESET creates a set from a list

```
(MAKESET (LAMBDA (LIS)
   (PROG (SETMADE)
         (SETQ SETMADE NIL)
         (MAP LIS (FQUOTE (LAMBDA LIS) (PROG ()
           (COND ((NULL LIS) (RETURN SETMADE))
                 ((MEMBER (CAR LIS) (CDR LIS)) SETMADE))
           (SETQ SETMADE (CONS (CAR LIS) SETMADE )) )))
         (RETURN SETMADE) )))
```

MAPCAR2.

```
(MAPCAR2 LIS1 LIS2 FNEXP)
NORMAL, EXPR, FUNCTIONAL.
```

This is an equivalent of `MAPCAR` which applies the two-argument functional expression `FNEXP` to the subsequent pairs of elements from lists `LIS1` and `LIS2`:

```
(DEF
  (MAPCAR2 (LIS1 LIS2 FNEXP)
    (COND ((NULL LIS1) NIL)
          (T (CONS (FNEXP (CAR LIS1) (CAR LIS2))
                   (MAPCAR2 (CDR LIS1) (CDR LIS2)
                                       FNEXP)))) ))
```

Example 6.9

```
 (MAPCAR2 '(1 2 3) '(4 5 6) 'PLUS) ==> (5 7 9)
 (MAPCAR2 '(1 2 3) '(4 5 6) 'TIMES) ==> (4 10 18)
```

GENC.
Another useful functional is `GENC`. Analysis of this function and finding applications for it is left to the reader.

```
(DEFINE '(
  (GENC (LAMBDA (LIS FUN)
      (PROG (FUNVALUE)
 LOOP (COND ((NULL LIS) (RETURN T)))
      (SETQ FUNVALUE (FUN (CAR LIS)))
      (COND ((NULL FUNVALUE) (RETURN NIL)))
      (SETQ LIS (CDR LIS))
      (GO LOOP) ))) ))
```

6.2 Property List of an Atom. Functions Processing the Property List

In the previous examples we assumed that only one memory cell corresponds to each literal atom. We must take into account, however, that to each atom correspond also some of its properties, and for the atoms being the names of the functions – the descriptions of these functions. This leads to the conclusion that the structure subordinated to each literal atom in the memory is more complex than a single cell. At first we will not describe here exactly this structure, which differs from a Lisp to Lisp implementation. We will describe here only what is common for all versions. The exact description of the property list will be given later on.

6.2.1 The Header of the Atom

To each symbolic atom subordinated is some computer cell, called the *header* of this atom. In all versions of Lisp the header of an atom includes the address of the structure being the value of this atom and the address of some structure organized in the form of the list, called the *property list of this atom*. The method of placing these two addresses in the atom's header depends on the particular Lisp implementation.

The reference to the atom's value is done by reading the address of this value in the respective part of the header. The functions that assign a value to the atom, place the respective address in this part of the header. For reading and changing elements of the property list special functions are used; the most important of which are `PUT` and `GET`.

6.2.2 The Property List

The property list of a symbolic atom is created when this atom is encountered for the first time in the program. For standard atoms (as for instance the names of the standard functions) the property lists are created while the system is defined (the Lisp interpreter is called).

The property list is the list of ordered pairs. The first element of each pair is the atom being the name of the given property - the so-called *indicator*. The second element of the pair is the value of this property (*name property* is also used). There exist several standard names of properties whose values have also the *standard form*. The most important of them are:

`PNAME` name for print. The value of this property is the code of alphanumeric characters which create the given atom. The property `PNAME` is created while organizing the property list of the atom. It exists in the property list of each atom which can be printed.

`SUBR` standard function. The value of this property is the machine code of the standard Lisp function `FSUBR` – standard special form. The value of this property is the machine code of the standard Lisp function being the special form. (See Chapter 3.)

The properties `SUBR` or `FSUBR` exist in the property list of each atom being the name of the standard function and is introduced there while the Lisp system is defined.

`EXPR` nonstandard function. The value of this property is the definition of the function introduced to the system with use of function `DEFINE`. The property `EXPR` is written into the property list of an atom being the name of the function `DEFINE`, while the `DEFINE` is evaluated.

Example 6.10

The property list of atom `AB` has the form:

```
((PNAME <code of characters AB> ))
```

the only element of this property list is the property `PNAME`.
The property list of atom `CAR` is the following:

```
(  (PNAME <code of characters CAR>)
   (SUBR  <code of function CAR>  )  )
```

The property list of atom `DEFINE` has the form:

```
( (PNAME <code of characters DEFINE>)
  (FSUBR <code of function DEFINE>  ) )
```

After evaluating of the function

```
(DEFINE '(
    (EXP (LAMBDA (A) .....)) ))
```

the property list of atom `EXP` will take the form:

```
((PPNAME <code of characters EXP>) (EXPR (LAMBDA (A)...)))
```

The user can put arbitrary properties on property lists of *any* atom. It is, however, advised to be very careful not to destroy the system by assigning the names of the standard properties. The indicators `INFO`, `PNAME`, `EXPR`, `SUBR`, textbfFEXPR, `FSUBR`, `SYM`, `CMACRO`, `SMACRO`, `CSUBR` and `CFSUBR` have special meaning to the Lisp system and should be avoided or used with care. Property lists do not have the same structure as ordinary lists and should be manipulated only by the following functions.

6.2.3 Functions Processing Property Lists

We will discuss the most important functions which serve for reading, modifying and extending the property lists.
GET.

```
(GET LITATOM1 LITATOM2)
NORMAL, SUBR.
```

GET is a standard two-argument function. It searches the property list of the atom whose name is the value of the first argument. If the property list includes the property being the name of the second argument then the value of this property is returned as the value of GET, else NIL is returned.

Example 6.11

If the property list of atom A has the form

```
( (PNAME <code of character A>)
  (ALPHA (A B C))
  (BETA X) )
```

then

```
(GET 'A 'ALPHA) ==> (A B C)
(GET 'A 'BETA)  ==> NIL
(GET 'A 'Z)     ==> NIL
```

The call (PRINT (GET 'A 'PNAME)) leads to an error while the values of properties PNAME, SUBR and FSUBR, as *machine codes*, cannot be printed with PRINT.

Function GET can be also applied for searching normal list structures. If the list is the value of the first argument then the value of the function is the element of this list following in it directly the element being the value of the second argument, or NIL is the list doesn't include such an element.

If the dotted pair is the value of the first argument or the list terminated with such a pair then if the element was not found in the list then GET searches the property list of the second element of the pair.

Example 6.12

Let the property list of atom A be as in the previous example:

```
(GET '(A B C) 'A) ==> B
(GET '(A B C) 'B) ==> C
(GET '(A B C) 'Z) ==> NIL
(GET '(A B C . A) 'BETA) ==> X
(GET '(B.A) 'ALFA) ==> (A B C)
(GET (GET (QUOTE A) (QUOTE ALFA)) (QUOTE B)) ==> C
```

The definition of GET is:

```
(DEFINE '(
  (GET (LAMBDA (LITATOM1 LITATOM2)
    (COND
      ((NULL LITATOM1) NIL)
      ((ATOM LITATOM1) (GET1 (CSR LITATOM1) LITATOM2))
      ((EQ (CAR LITATOM1) LITATOM2)(CADR LITATOM1))
       (T (GET (CDR LITATOM1) LITATOM2)) )))
          (GET1 (LAMBDA (X Y)(COND
            ((NULL X) NIL)
            ((EQ (CSR X) Y) (CAR X))
            (T (GET1 (CDR X) Y)) )))      ))
```

Note the use of function CSR. Function GET can be defined without using CSR in other Lisps.

DEFLIST.

(DEFLIST ((LITATOM1 S1)...(LITATOMN SN)) LITATOM)
PSEUDOFUNCTION, SUBR.

DEFLIST is a two-argument pseudofunction. The *value* of the first argument must have the following form:

```
((LITATOM1 S1)...(LITATOMN SN))
```

where LITATOM1...LITATOMN are literal atoms and S1 .... SN are arbitrary expressions. The value of the second argument must be a litaral atom. The operation of the function consists in placing into the property list of each atom LITATOM1 ,..., LITATOMN, under indicator being the value of the second argument of the function, the properties S1, ..., SN, respectively. If the properties of this indicator have already existed in the property list of one of these atoms, then the old properties are replaced with the new ones. As the value of DEFLIST, returned is the list of LITATOM[I]-s which appear in the first argument, that is, a list of all atoms which have obtained the new property by DEFLIST.

Example 6.13

Evaluating of the expression

```
(DEFLIST '(
    (A (A B C))
    (B X)
    (C (LAMBDA (X) NIL))) ALFA)
```

writes element

```
(ALFA (A B C))
```

into the property list of atom A, the element

```
(ALFA X)
```

into the property list of atom B, and the element

```
(ALFA (LAMBDA (X) NIL))
```

into the property list of atom C.
DEFLIST execution with indicator EXPR is equivalent to the execution of DEFINE. However, the value is different because the value of DEFINE is the list of names of the defined functions.

There are also other defining functions in Lisp which serve for quick defining but are equivalent to DEFINE or DEFLIST.

```
(DEFINE '(
          (AF (LAMBDA (X) (CAR X)))
          (BF (LAMBDA (X) (CADR X)))
         ))
```

is equivalent to

```
(DEF    (AF (X) (CAR X))
        (BF (X) (CADR X))  )
```

and to

```
(DEFLIST  '(
            (AF (LAMBDA (X) (CAR X)))
            (BF (LAMBDA (X) (CADR X)))
           ) EXPR)
```

DEFF is an equivalent for DEFLIST, using FEXPR as the indicator,

```
(DEFF (SETQ (LISTAR)
            (SET (CAR LISTAR)
                 (EVAL (CADR LISTAR)))))
```

is the equivalent of

```
(DEFLIST '(
    (SETQ (LAMBDA (LISTAR)
        (SET (CAR LISTAR)
             (EVAL (CADR LISTAR)))))
     ) FEXPR)
```

DEF and DEFF are FSUBR pseudofunctions.

REMPROP

```
(REMPROP LITATOM1 LITATOM2)
PSEUDOFUNCTION, SUBR.
```

REMPROP is a two-argument pseudofunction. If the value of its first argument is a symbolic atom, then the function searches the property list of this atom and deletes from it the property whose indicator is the value of the second argument.

Function REMPROP can also be used for deleting elements from the normal list structures. If such list is the value of the first argument then REMPROP will delete from this list each element directly following the element being the value of the second argument. The value of REMPROP is the value of the first argument.

Example 6.14

Let the property list of atom A be:

```
((PNAME <code of A>) (A (X Y Z)) (B X) (C Z))
```

After evaluating

```
(REMPROP 'A 'A)
```

the list will take the form:

```
(REMPROP 'A 'C)
```

Then the property list of atom A takes the form:

```
((PNAME <code A>) (B X))

     (SET 'A '(X B Z X A Z X V)) ==> (X B Z X A Z X V)
     (REMPROP A (QUOTE X)) ==> NIL
     (PRINT A) ==> (X Z X Z X)
     (REMPROP A (QUOTE Z)) ==> NIL
     (PRINT A) ==> (X Z Z)
     (REMPROP A (QUOTE X)) ==> NIL
     (PRINT A) ==> (X Z)
```

PUT.

```
(PUT LITATOM1 LITATOM2 S)
PSEUDOFUNCTION, SUBR.
```

If the value of S is other than NIL, then PUT searches the property list of the atom being the value of LITATOM1. If the property of indicator being the value of LITATOM2 exists there, then the old value of this property is replaced with the value of S. If there is no property of this value, then it is placed into the property list. If NIL is the value of S, then the property of the name being the value of LITATOM2 is deleted from the property list. The value of S is returned as the value of function PUT. The definition of PUT:

```
(DEFINE '(
        (PUT (LAMBDA (A B S)
             (COND ((NULL S) (REMPROP A B))
                   (T (PUTPROP A B S))) ))

        (PUTPROP (LAMBDA (A B S)
                   (PROG () (DEFLIST (LIST (LIST A S)) B)
                             (RETURN S))  ))
         ))
```

PUTPROP places the property of the name being the value of its second argument and the value, being the value of its third argument on the property list of the atom being the value of its first argument. The value of the property is also returned as the value of PUTPROP.

GET should not be used to obtain the PNAME property of an atom if the PNAME is going to be used with the character manipulating functions. Instead the function GETPN should be used.

```
(PUTD LITATOM S)
PSEUDOFUNCTION, SUBR.
```

PUTD replaces the definition of LITATOM (using either EXPR or FEXPR indicator) with new definition S using the same indicator as was used in LITATOM. If LITATOM has no EXPR or FEXPR indicator, PUTD does nothing. The value of LITATOM is returned as the value of PUTD.

```
(DEF
     (PUTD (LITATOM S)
PROG2 (OR (AND (GET LITATOM 'EXPR)
               (PUT LITATOM 'EXPR S))
          (AND (GET LITATOM 'FEXPR)
               (PUT LITATOM 'FEXPR) S))
LITATOM) ))
```

QKEDIT.

```
(QKEDIT LITATOM SD S2)
PSEUDOFUNCTION, SUBR.
```

QKEDIT substitutes S for S2 within the definition of any EXPR or FEXPR function LITATOM.

```
(DEF (QKEDIT (LITATOM S1 S2)
       (PUTD LITATOM
          (SUBST S1 S2 (GETD LITATOM))) ))
```

Example 6.15

We want to make selective *"trace"* of function AFUN, pretty printing its values, but only when AFUN is called inside function BFUN. We do this by:

```
(QKEDIT 'BFUN '(AFUN) '(PPRINT (AFUN)))
```

FLAG.

```
(FLAG LAT LITATOM)
PSEUDOFUNCTION, SUBR.
```

FLAG puts the property indicator LITATOM on the property list of each of the atoms in LAT with the value *T*. The value of FLAG is NIL.

```
(DEF
   (FLAG (LAT LITATOM)
      (MAP LAT
           (FQUOTE (LAMBDA (AT)
              (PUT AT LITATOM T)))) ))
```

REMFLAG.

```
(REMFLAG LAT LITATOM)
PSEUDOFUNCTION, SUBR.
```

REMFLAG removes the occurrences of the indicator LITATOM from the property list of each of the atoms in LAT. The value is NIL.

```
(DEF
   (REMFLAG (LAT LITATOM)
      (MAP LAT
           (FQUOTE (LAMBDA (AT)
              (REMPROP AT LITATOM)))) ))
```

GETD.

```
(GETD LITATOM)
PSEUDOFUNCTION, SUBR.
```

GETD returns the functional definition of LITATOM by getting the value associated with the EXPR or FEXPR indicator on LITATOM's property list. GETD will also force a retrieval of the function LITATOM from the disk, if LITATOM has previously been the argument of a call to DISKOUT (see the description of the Lisp virtual memory facility for functions). If LITATOM does not have an EXPR or FEXPR indicator on its property list, GETD will return NIL. Thus system functions with SUBR or FSUBR definitions are not retrieved by GETD.

PROP.

```
(PROP LITATOM1 LITATOM2 FNEXP)
NORMAL, SUBR.
```

PROP is similar to GET in action. The property list of LITATOM1 is searched for the indicator LITATOM2. If such a property is found, the entire property list of LITATOM1 beginning with property LITATOM2 is returned by PROP. If LITATOM2 is not an indicator on the property list of LITATOM1, then FNEXP which must be a function of no arguments is applied and the value returned by PROP is the value of FNEXP.

6.2.4 Problems

1. Describe the property lists of the atoms:

   a) ABC,

   b) CONS,

   c) PROG,

   d) READ,

   e) PRINT.

   if no additional properties were put there.

2. Let the property list of the atom X have the form:

   ((PNAME < *code X* >) (A (X Y Z)) (B X) (ALFA B))

   How would the property list of this atom look after evaluating the expression:

   a) (DEFLIST ((X (A B C))) 'BETA)

   b) (REMPROP 'X 'A)

   c)
   ```
   (DEFLIST '((GET (QUOTE X) (QUOTE B))
              (GET (QUOTE X) (QUOTE A)))
            'BETA )
   ```

   d) (DEFLIST '((X (CDR (GET (QUOTE X)(QUOTE A))))) A)

   e)
   ```
   (LIST '(SETQ A (GET (QUOTE X)(QUOTE A)))
         (SETQ B (GET (QUOTE X)(QUOTE B)))
         (DEFLIST (((QUOTE X) B)) (QUOTE A))
         (DEFLIST (((QUOTE X) (CADR A))) (QUOTE B)))
   ```

3. Define the function LISTPROP of arguments A and B. The value of A is the list of the form:

```
((NAME1 VAL1)
 (NAME2 VAL2)
      ...
 (NAMEN VALN))
```

   where NAME1 ... NAMEN are literal atoms. The task of the function is to introduce into the property list of this atom the properties of the names NAME1 ... NAMEN and the values respectively VAL1 ... VALN. The value of the function is NIL.

4. Define function LISTPROP1 which differs from the function LISTPROP from problem 3 only in that its value is the list of names of the properties being introduced into the property list of the atom being the value of B.

5. Define function PRINTPROP of arguments A and B. The value of A is a literal atom. The value of B is a list of literal atoms. The list of pairs shall be returned as the value. The first element in each pair is the successive element from the list being the value of B and the second element is the value of this name, taken from the property list of the atom being the value of A or NIL, if the value of this name does not exist in this property list.

6.2.5 Solutions to Problems

1

a) ((PNAME <code ABD>))

b) ((PNAME <code CONS>) (SUBR <code of function>))

c) ((PNAME <code PROG>) (FSUBR <code of function>))

d) ((PNAME <code READ>) (FSUBR <code of function>))

e) ((PNAME <code PRINT>) (FSUBR <code of function>))

2

a)
```
((BETA (A B C)) (PNAME <code X>) (A (X Y Z))
     (B X) (ALFA B))
```

b) ((PNAME <code X>) (B X) ALFA B)

c) ((BETA (X Y Z)) (PNAME <code X>) (A (X Y Z)) (B X) (ALFA B))

d) ((PNAME <code X>) (A (Y Z)) (B X) (ALFA B))

e) ((PNAME <code X>) (A X) (B Y) (ALFA B))

3

```
(DEFINE '( (LISTPROP (LAMBDA (A B) (PROG (X)
      (SETQ X A)
   AA (DEFLIST ((B (CADAR X))) (CAAR X))
      (COND((NOT(NULL(SETQ X (CDR X)))) (GO AA))) )))
    ))
```

4

```
(DEFINE '(
    (LISTPROP1 (LAMBDA (A B) (PROG NIL
      (COND ((NULL A) (RETURN NIL)))
      (DEFLIST ((B (CADAR A))) (CAAR A))
      (RETURN (CONS (CAAR A) (LISTPROP1 (CDR A) B))) )))
                ))
```

5

```
(DEFINE '(
      (PRINTPROP (LAMBDA (A B)
          (COND ((NULL B) NIL)
                (T (CONS(LIST(CAR B)(GET A(CAR B)))
                        (PRINTPROP A(CDR B))))) ))
      ))
```

We will define now a nonstandard function PRINTPROP which writes the actual state of the property list as the associative list. For SUBR's and FSUBR's the addresses of respective machine codes are printed and the printname is printed for PNAME. The list is printed in the reverse order.

```
(DEF
  (PRINTPROP (LITATOM)
    (COND
      ((ATOM LITATOM)(COND ((NUMBERP LITATOM)(PRINT LITATOM))
                           (T (PRINTPROP1 (CSR LITATOM)))))
      (T (ERROR (LIST
          "$$$ ERROR - ARGUMENT OF PRINTPROP$ LITATOM
          "$$$ IS NOT ATOM $))) ))

  (PRINTPROP1 (PROPLIST)
    (COND
       ((NULL PROPLIS) NIL)
       (T (CONS
            (CONS (CSR PROPLIS)
                  (SELECT (CSR PROPLIS)
                     ((QUOTE PNAME)               ;; where explained??
                      (INTERN (GETPN LITATOM)))
                     ((QUOTE SUBR)
                      (ADDR (CAR PROPLIS)))
                     ((QUOTE FSUBR)
                      (ADDR (CAR PROPLIS)))
                     (CAR PROPLIS)) )  ; CONS FOR PAIR
          (PRINTPROP1 (CDR PROPLIS)) )) )))
```

6.3 Description of Graphs

The GRASPE functions serve for creating, modifying and searching graphs [194].

Graphs and Hipergraphs.

An ordered graph consists of the set of nodes and arrows joining the nodes. The data structures describing the graphs must include: names of nodes, the multiple arrows among two nodes, occurrence of the same node in more than one graph. The nodes can be treated as objects and the arrows as some relations among objects. Graph describes then the set of relations. We assume that arrows are local in each graph but the same names of nodes can occur in many graphs.

A convenient tool for describing the sets of graphs are the hipergraphs. A *hipergraph* is the set of all graphs at the given point of program execution. With the use of hipergraphs, defining global nodes and the operations on graphs becomes simple.

Let us assume that initially the hipergraph is empty. While Lisp program using GRASPE functions is executed, the nodes, arrows and graphs are added or removed. Each operation on each graph is then considered to be an operation on the hipergraph.

Definition 6.1 Hipergraph *is an ordered 5-tuple* (G, N, A, s, f) *where:*

G *is the finite set of graphs,*

N *is the finite set of nodes,*

A *is the finite set of names of arrows,*

s $G \rightarrow 2^N$ *(*2^N *is the set of all subsets of set* N*,* s *defines the nodes in each of the graphs.*

f: $G \rightarrow 2^{N \times A \times N}$ *and for all* $l \in G, f(l) \subseteq s(l) \times A \times s(l)$

If $(n, a, m) \in f(l)$*, then this is an arrow from node* n *to node* m *with label* a *in graph* l*. Let us note that each single graph is completely defined by the value of* $s(l)$ *(specifying its nodes) and* $f(l)$ *(specifying its arrows).*

Basic Functions.

The basic GRASPE functions can be divided into three groups:

- operations searching graphs,
- destroying graphs,
- creating graphs.

The mnemonics:

```
s - set of                    n  - nodes
d - destroy                   g  - edges
c - create                    gr - graph
o - out-pointing              x  - given extra argument
i - in-pointing               e  - empty
a - adjacent                  is  - isolated
p - pair of pairs
```

Then, the operation 'SIN' is read as SET OF INPOINTING NODES. The data structure on which we operate is always the single hipergraph (G, N, A, s, f) of arbitrary complexity, $n, m \in N, l \in G, a \in A$.

1) **Searching Operations in Graphs.**

a) `sop (n, l)` = $\{ (m, a) \mid (n, a, m) \in f(l) \}$

b) `sog (n, l)` = $\{ a \mid \exists m \ni (n, a, m) \in f(l) \}$

c) `sogx (n, m, l)` = $\{ a \mid (n, a, m) \in f(l) \}$

d) `son (n, l)` = $\{ m \mid \exists a \ni (n, a, m) \in f(l) \}$

e) `sonx (n, a, l)` = $\{ m \mid (n, a, m) \in f(l) \}$

f) `sip (n, l)` = $\{ (m, a) \mid (m, a, n) \in f(l) \}$

g) `sig (n, l)` = $\{ a \mid \exists m \ni (m, a, n) \in f(l) \}$

h) `sigx (n, m, l) = sogx (m, n, l)`

i) `sin (n, l)` = $\{ m \ \exists a \ \ni \ (m, a, n) \in f(l) \}$

j) `sinx (n, a, l)` = $\{ m \mid (m, a, n) \in f(l) \}$

k) `sap (n, l) = sop (n, l)` $\cup$ `sip (n, l)`

l) `sag (n, l) = sog (n, l)` $\cup$ `sig (n, l)`

m) `sagx (n, m, l) = sogx (n, m, l)` $\cup$ `sigx (n, m, l)`

n) `san (n, l) = son (n, l)` $\cup$ `sin (n, l)`

o) `sanx (n, a, l) = sonx (n, a, l)` $\cup$ `sinx (n, a, l)`

p) `nodeset (l) = s(l)`

* functions with "a" (as `sap` or `san`) serve for description of non-oriented graphs.

2) **Operations Destroying Graphs and their Parts.**

a) `dop (n, a, m, l)` = *T* with the side effect f(l) := f(l) – {(n, a, m)}

b) `dog (n, l)` = *T* with the side effect f(l): = f(l) – $\{ (n, a, m) \mid \exists a, m \ni (n, a, m) \in f(l) \}$

c) `dogx (n, m, l)` = *T* with side effect f(l): = f(l) - $\{ (n, a, m) \mid \exists a \ni (n, a, m) \in f(l) \}$

d) `dip (n, a, m, l) = dop (m, a, n, l)`

e) `dig (n, l)` = *T* with side effect f(l) = f(l) - $\{ (m, a, n) \mid \exists m, a \ni (m, a, n) \in f(l) \}$

f) `digx (n, m, l) = dogx (m, n, l)`

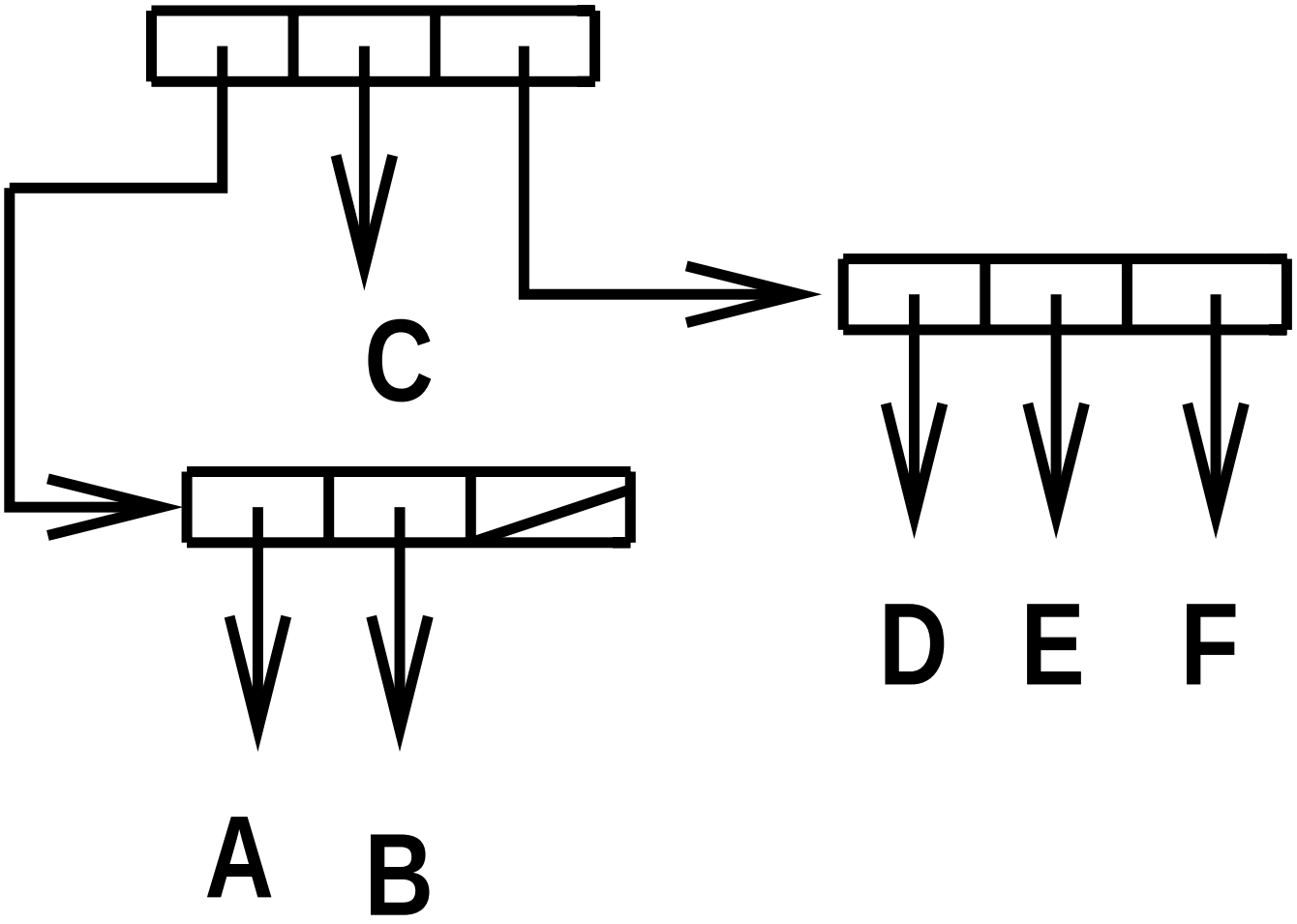


Figure 6.1: Structure to Problem 1

g) `dap (n, a, m, l) = dop (n, a, m, l)` $\wedge$ `dip (n, a, m, l)`

h) `dagx (n, m, l) = dogx (n, m, l)` $\wedge$ `digx (n, m, l)`

j) `disn (n, l)` = *T* with side effect s(l): = s(l) - {n} when `sag (n, l)` = {}

k) `degr (l)` = *T* with side effect G: = G - {l} when s(l) = {}

3) **Operations Creating Parts of a Graph.**

a) cegr (l) = *T* with side effect G: = G $\cup$ {l} , s(l) = {} and f(l) = {} when $l \notin G$

b) cisn (n, l) = *T* with side effect N = N $\cup$ {n} and s(l) = s (l) $\cup$ {n}

c) cop (n, a, m, l) = *T* with side effect N = N $\cup$ {n, m} , A = A $\cup$ {a}, s(l) = s(l) $\cup$ {n,m} and (f(l) = f(l) $\cup$ {(n, a, m)})

(BREAK IN SEQUENCE IN TEXT)

6.3.1 Problems

1. The following procedures `READ` and `PRINT` write programs for reading and printing packed Lisp structures. For instance, the structure from Figure 6.1 is written outside as

```
((A * B. NIL) * C . (D * E . F))
```

2. Write procedures `READG` and `PRINTG` for printing looped packed structures. For instance, structure `X` from Figure 6.2 has to be written on input as:

```
(1 = (G * 1 . (2 = (2 * A . (2 * B . C)) )))
```

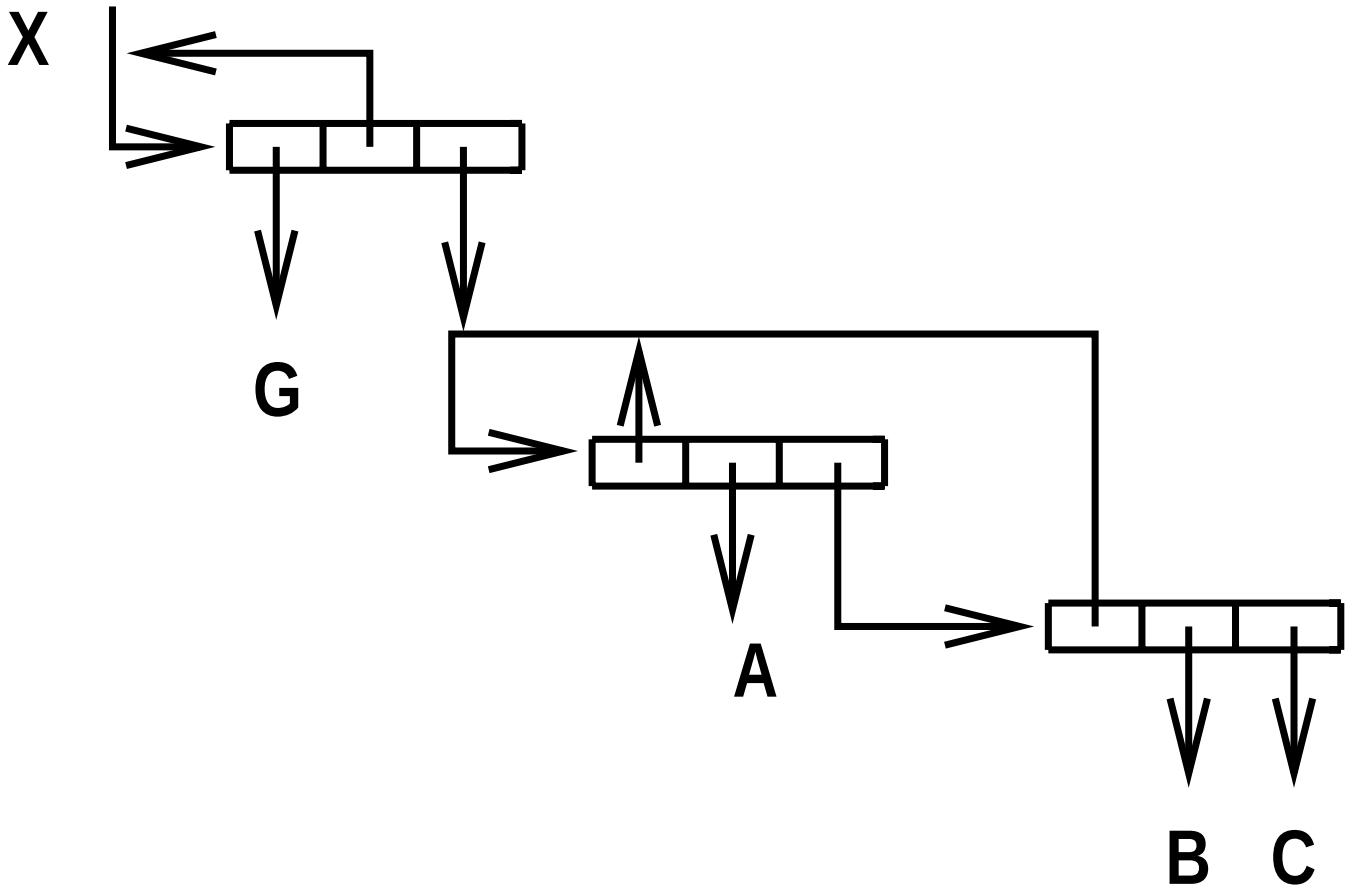


Figure 6.2: Structure to Problem 2

On output the numbers which are identifying the elements are replaced with addresses. Apply function `ADDR`.

3. Write function `(SORTFUN PRED LIST)` being the functional sorting the atoms on list `LIST` with respect to some predicate `PRED`. Test on functions `ALPHAP`, `GRADP`, `LESSP`, `GREATERP`.

4. Write the program which orders the set of subsets of set `A` into the lattice of subsets.

5. Write function `(NONDET INSTATE FINSTATE OPERATORS)` where `INSTATE` is the arbitrary S-expression describing the initial state of the solution space. `FINSTATE` is the arbitrary S-expression describing the final state and `OPERATORS` is the set of operators to apply.

6. Write the input procedure reading the input lines of the format

```
0 1 1 0 x . . . 1 0
```

   and transforming them to actual words, where 0 is encoded with 10, 1 with 01 and X with 11.

7. What is this program doing. Improve it.

```
(def test2  (lambda ()
;---------------------------------------
(prog()
  (setq inp (infile 'dane))
  (setq out (outfile 'result))
  loop
  (setq x (read inp))
  (setq sett nil)
  (flatten x)
  (setq yy (sort  sett nil ))
  (pp-form yy out)
  (pp-form yy )
  (go loop)
  )))

(def flatten (lambda (expr)
(cond
  ((atom expr)
   (cond ((and (litatom expr)
       (not (member expr sett)))
  (setq sett (cons expr sett)))
 (t nil)))
  (t (flatten (car expr))
     (flatten (cdr expr))))))
(setq sett nil)
```

Chapter 7

Advanced Lisp and Extensions to Lisp.

7.1 Definitions of Standard Special Forms of Lisp

As more examples, we will define standard special functions of Lisp: SETQ, CONC, SELECT, OR, PROGN and COND*.

```
(DEFF                          ;; is it DEFF or DEF
 (SETQ (X) (SET (CAR X) (EVAL (CADR X))))

 (CONC (X) (MAPCON X `(LAMBDA
                       (X) (EVAL (CAR X)))))

 (SELECT (X)
   (PROG (U V)
     (SETQ U (EVAL (CAR X)))
   A (COND ((NULL (CDDR X)) (RETURN (EVAL (CADR X) ))))
     (SETQ X (CDR X))
     (COND ((EQUAL (EVAL (CAAR X)) U)
            (RETURN (EVAL (CADAR X)) )))
     (GO A) ))

 (OR (X)
  (PROG (C)
  A (COND ((NULL X) (RETURN NIL))
          ((EVAL (CAR X)) (RETURN T)))
    (SETQ X (CDR X)) (GO A) ))
        )

(DEFLIST `(
 (PROGN (LAMBDA (X)
  (PROG ()
    A (COND ((NULL X) (RETURN T))
            ((NULL (CDR X)) (RETURN (EVAL (CAR X)) )))
      (EVAL (CAR X))
      (SETQ X (CDR X))
      (GO A) )))) `FEXPR)

(DEFLIST `(
 (COND* (LAMBDA (X)
   (PROG (CLAUSE PREDICATE)
   A (COND ((NULL X) (RETURN NIL)))
     (SETQ CLAUSE (CAR X))
     (PPRINT (LIST `CLAUSE CLAUSE))
     (PPRINT (LIST `PREDICATE
        (SETQ PREDICATE (CAR CLAUSE))))
     (PPRINT (LIST `VALUE-OF-PREDICATE
```

```
        (SETQ VALUE-OF-PREDICATE (EVAL PREDICATE))))
     (COND ((VALUE-OF-PREDICATE
            (RETURN (EVAL (CONS 'PROGN (CDR CLAUSE))) ))))
     (SETQ X (CDR X))
     (GO A) )))) 'FEXPR)
```

FOR ¿¿¿¿¿¿¿¿¿¿¿¿¿ BRAKUJE
The call of this function is

```
    (FOR <name of Loop's variable> FROM <initial value>
         BY <step> TO <final value> DO <sequence of actions>)
```

The elements of list `LIS` of arguments of `FOR` is then

```
    (CAR LIS) = <name of Loop's variable>
    (CADR LIS) = atom FROM
    (CADDR LIS) = non-evaluated initial value
    (CADDDR LIS) = atom BY
    (CADDDDR LIS) =  non-evaluated step
    (CADDDDDR LIS) = atom TO
    (CADDDDDDR LIS) = non-evaluated final value
    (CADDDDDDDR LIS) = atom DO
    (CDDDDDDDDR LIS) = list of statements to evaluate
```

```
(DEFLIST '(
 (FOR (LAMBDA (LIS)
  (PROG (FROMV BYV TOV CURV)
    (COND ((NOT (AND
             (EQ (CADR LIS) 'FROM)
             (EQ (CADDDR LIS) 'BY)
             (EQ (CADDDDDR LIS) 'TO)
             (EQ (CADDDDDDDR LIS) 'DO)))
           (ERROR '$$$BAD FORM OF STATEMENT 'FOR' $)))
    (SETQ FROMV (EVAL (CADDR LIS)))
           ;    FROMV - EVALUATED INITIAL VALUE
    (SETQ BYV (EVAL (CADDDR LIS)))
           ;    BYV - EVALUATED VALUE OF STEP
    (SETQ TOV (EVAL (CADDDDDDR LIS)))
           ;    TOV - EVALUATED FINAL VALUE
    (SET (CAR LIS) FROMV)
         ; INITIAL VALUE OF LOOP'S VARIABLE IS ASSIGNED.
         ; (CAR LIS) IS THE NAME OF THE VARIABLE
  A (SETQ CURV (EVAL (CAR LIS)))
         ; CURV IS THE CURRENT VALUE OF LOOP'S VARIABLE
    (EVAL (CONS 'PROGN (CDDDDDDDDR LIS)))
          ; EVALUATION OF SEQUENCE OF STATEMENTS
    (SET (CAR LIS) (PLUS CURV BYV))
    (COND ((EQUAL CURV TOV) (RETURN NIL)))
    (GO A) )))
        ) 'FEXPR)
```

7.2 Character Manipulating Functions

CLEARBUFF.

```
(CLEARBUFF)
PSEUDOFUNCTION, SUBR.
```

CLEARBUFF clears the internal buffer and resets the counter of its contents to zero. The value of CLEARBUFF is NIL. This function should be executed before any of the other functions which affect the buffer are used.

PACK

```
(PACK CHARACTER)
PSEUDOFUNCTION, SUBR.
```

PACK places the character being the value of its argument at the next available position in buffer, following all characters which have been previously placed there. The value is NIL.

MKNAM

```
(MKNAM)
PSEUDOFUNCTION, SUBR.
```

MKNAM returns as its value a list of full-words created from the contents of the internal buffer. The characters are taken from the buffer ten (implementation dependent) at a time and placed into full-words containing up to ten display character codes. The last full-word is filled out with zero bits if necessary. The resulting list is one whose CAR fields point directly to the full-words containing the character codes. This type of list structure is not a normal Lisp list. The internal buffer is cleared after this function is completed.

INTERN

```
(INTERN FWL)
```

or

```
(INTERN LITATOM)
PSEUDOFUNCTION, SUBR.
```

The argument of INTERN can be either FWL or a LITATOM. INTERN searches the OBLIST for an atom whose print image matches FWL or LITATOM. If such an atom is found, INTERN returns that atom. Otherwise INTERN puts LITATOM or an atom it creates from the FWL on the OBLIST and returns this atom as its value.

COMPRESS

```
(COMPRESS LAT BOOLEAN)
PSEUDOFUNCTION, SUBR.
```

LAT must be a list of single-character atoms. If BOOLEAN is FALSE or omitted, COMPRESS creates and returns an atom identical to the result of READ if the same characters were read from an input file. That is, if the characters are a legal numeric representation, a numeric atom is returned; otherwise a standard symbolic atom of the first 30 characters created according to the syntax of Lisp.
If BOOLEAN is TRUE, then special characters which would ordinarily terminate the reading of an atom from the input file are included in the first 30 characters of the atom returned. If BOOLEAN is TRUE, the atom is interned, otherwise it is not.

Example 7.1

```
(COMPRESS '(A B #. #, #.)) = AB
(COMPRESS '(A B #. #, #.) T) = AB.,.
```

NUMOB.

```
(NUMOB)
PSEUDOFUNCTION, SUBR.
```

NUMOB expects the internal buffer to contain a sequence of characters defining a number. The syntax of the numbers is the same as if the characters had been typed. The value of this function is the Lisp number which corresponds to that character representation. If the characters in the buffer do not form a legal Lisp number, then NUMOB returns as its value a literal atom whose print name contains those characters. The buffer is empty after this function is complete.

```
(DEF
    (COMPRESS1 (LAMBDA (LISTA)
       (PROGN
          (CLEARBUFF)
          (MAP LISTA (FQUOTE (PACK ((CAR LISTA)) ))
                     (INTERN (MKNAM)) ))) ))

    (COMPRESS1 '(A B C)) ==> ABC
```

UNPACK.

```
(UNPACK FW)
PSEUDOFUNCTION, SUBR.
```

The argument FW is a full-word, i.c. a memory word containing up to ten display code characters, filled with zero bits if less than ten characters are present. The value of UNPACK is a list of symbolic atoms, one atom for each character contained in the argument of UNPACK. The print names of the symbolic atoms in the list correspond to the character codes in the full-word.

IMAGEL.

```
(IMAGEL ATOM BOOLEAN)
PSEUDOFUNCTION, SUBR.
```

IMAGEL returns the integer length of the printed representation of ATOM. If BOOLEAN is false, the image used is the normal printer image. If BOOLEAN is true, the image used includes any delimiter or escape symbols necessary to reproduce a readable form of the ATOM. This function facilitates formatting printed output.

NUMTOATOM.

```
(NUMTOATOM NUMBER)
PSEUDOFUNCTION, SUBR.
```

NUMTOATOM creates and returns a symbolic atom whose print name is equivalent to the representation NUMBER would have if printed. The literal atom is not placed on OBLIST. The format used for floating-point numbers is under the control of NFORMAT.

EXPLODE.

```
(EXPLODE ATOM)
PSEUDOFUNCTION, SUBR.
```

This is a reverse function to COMPRESS, i.e.:

```
    (COMPRESS (EXPLODE ATOM)) ==> ATOM
```

ATOM can be either symbolic or a number

Example 7.2

```
    (GENSYM1 (LAMBDA (ATOM)
       (COND ((MEMBER ATOM ATOMS) (GENSYM2 ATOM))
             (T (SETQ ATOM (CONS ATOM ATOMS))
                (PUT ATOM 'VAL 0)
                (GENSYM2 ATOM))) ))

    (GENSYM2 (LAMBDA (ATOM)
       (COMPRESS (APPEND
          (EXPLODE ATOM)
          (EXPLODE NUMTOATOM
             (PUT ATOM 'VAL (ADD1 (GET ATOM 'VAL)))
                                      )) )))

    (GENSYM1 'MACHINE) ==> MACHINE1
```

```
(GENSYM1 'CAR) ==> CAR1

(GENSYM1 'MACHINE) ==> MACHINE2
```

The reader is asked to analyse functions GENSYM1 and GENSYM2 and explain their possible applications.

LITER.

```
(LITER S)
PREDICATE, SUBR.
```

LITER returns S as its value if S is a symbolic atom whose print name is a single alphabetic character A - Z. The value of LITER is false in all other circumstances.

DIGIT.

```
(DIGIT S)
PREDICATE, SUBR.
```

DIGIT returns S as its value if S is a symbolic atom whose printname is a single numeric character 0 - 9. The value of DIGIT is false under all other circumstances. Note that DIGIT is false for the single-digit numeric atoms 0 - 9.

OPCHAR

```
(OPCHAR S)
PREDICATE, SUBR.
```

As its value, OPCHAR returns S if S is a symbolic atom whose print name is a single character and is either +, -, / or *. OPCHAR returns false otherwise.

7.3 Few More Control Functions

EXIT.

```
(EXIT LITATOM EXP)
PSEUDOFUNCTION, SUBR.
```

EXIT causes Lisp to exit the most recent call to the EXPR or FEXPR function LITATOM. EXIT evaluates EXP a second time. This second evaluation takes place in the context of LITATOM rather than in the context of the call to EXIT. EXIT returns the resulting value as the value of the function LITATOM. EXIT differs from RETURN in that EXIT returns from LITATOM to the function which called LITATOM independent of whatever PROG expressions have been entered.

```
(DEFINE '(
   (EXIT (LAMBDA (LITATOM EXP)
            (RETFROM
               (NTHFNBK LITATOM 1) EXP) )) ))
```

EVAL.

```
(EVAL EXP)
NORMAL, SUBR.
```

EVAL evaluates its argument a second time. The value of this second evaluation of EXP is returned as a value of EVAL. Variables within the expression will be evaluated in the context in which the call to EVAL is made.

Example 7.3

```
(OPEN FILENAME)
(EVAL (INPUT FILENAME))
```

if FILENAME includes definition of functions, then after execution of the above statements, these functions are defined.

```
(EVAL 'A) ==> A
(EVAL (CONS 'PLUS (LIST 2 3))) ==> 5
(EVAL '(PLUS 2 3)) ==> 5
```

APPLY.

```
(APPLY FUNCTION LIST)
NORMAL, SUBR.
```

The LIST is a list of arguments for the FUNCTION which is the first argument of APPLY. The value of APPLY is the value obtained by applying FUNCTION to LIST. The first argument must be either a function name or a λ-expression. If it is a function name, the function must be of type EXPR or SUBR. If a FUNCTION of type FEXPR or FSUBR is given to APPLY, then an undefined function error will occur even though the function is in fact defined.

```
(APPLY 'CONS '(2 3)) ==> (2 . 3)
(APPLY '(LAMBDA (X Y) (CONS X Y)) '(2 3)) ==> (2 . 3)
(APPLY 'PLUS '(2 3)) ==> ERROR   ;   ALWAYS?     WHY?
```

EVALQUOTE.

```
(EVALQUOTE FUNCTION LIST)
NORMAL, SUBR.
```

The result of application of FUNCTION to LIST is the value of EVALQUOTE. The FUNCTION can be FEXPR, FSUBR, EXPR or SUBR.

```
(EVALQUOTE 'PLUS '(2 3)) ==> 5

(DEFINE '(
   (EVALQUOTE (LAMBDA (FUNCTION LIST)
        (EVAL (CONS FUNCTION LIST)) )) ))
```

Some Lisp implementations have EVALQUOTE on top level, some other have EVAL. Most current implementations of Lisp have EVAL.

EVLIS.

```
(EVLIS LIST)
NORMAL, SUBR.
```

EVLIS expects the value of its argument to be a list of expressions. It evaluates these expressions one at a time from left to right and returns a list whose elements are the respective values of the expressions.

COMMENT.

```
(COMMENT S1...SN)
PSEDUDOFUNCTION, FSUBR.
```

Pseudofunction COMMENT retains comments within a defined Lisp program. It does nothing and returns NIL.

7.4 General Application Functions that have Applications in Diagnostics

ADDR.

```
(ADDR S)
PSEUDOFUNCTION, SUBR.
```

ADDR takes the contents of register A1 (being the actual address of S in the memory) and returns the octal number being the address of this S-expression. ADDR enables the user to determine the actual machine's address of any given S-expression.

```
(ADDR NIL) ==> 21615Q

((LAMBDA (X Y) (ADDR (GET X Y))) 'GET 'SUBR) ==> 5462Q
```

RECLAIM.

```
(RECLAIM)
PSEUDOFUNCTION, SUBR.
```

Calling of `RECLAIM` causes execution of Garbage Collector. Value od `RECLAIM` is `NIL`. The user can determine the degree of occupation of memory testing atom `CROWDEDP`. If the value of property `APVAL` of this atom is `*T*`, then GC has detected that the list of free memory is short. `NIL` is returned otherwise. Testing this atom we can determine when it is good to send some big S-expression to the hard disk memory.

Part II

Combinational Problems and Multiple-Valued Logic

Chapter 8

Introduction to Search

8.1 Depth-First Search Strategy

Below given is a program that uses Depth-First Search strategy to create a so-called *morphological box.* Morphological box is nothing else than the Cartesian Product of sets being its argument. Please observe the order of enumeration of elements of this Cartesian Product.

```
(DEFINE
  '((INITIALIZE
     (LAMBDA (FILENAME)
       (PROG (X)
        AA (COND ((NOT (EQ (SETQ X (INPUT FILENAME)) $EOF$))
                  (EVAL X)
                  (GO AA)))
           (RETURN 'INITIALIZED))))))

(DEFINE '((MORFBOX (LAMBDA (MORFOL) (MORFOLOG1 (LIST (LIST1 NIL)))))))

(DEFINE
  '((MORFOLOG1
     (LAMBDA (OPEN)
     (PROG (NODE)
        AA (COND ((NULL OPEN) (RETURN)))
           (SETQ NODE (CAR OPEN))
           (SETQ OPEN (CDR OPEN))
           (SETQ OPERATORS (NTH MORFOL (SD NODE)))
           (COND ((EQN (ADD1 (LENGTH MORFOL)) (SD NODE))
                  (PRINT (REVERSE (QS NODE))))
                 (T
                  (SETQ OPEN (APPEND (SUCCESSORS NODE
                   OPERATORS) OPEN))
                                          ))
           (GO AA))))))

(DEFINE '((SD (LAMBDA (N) (CAR N))) (QS (LAMBDA (N) (CADR N)))))

(DEFINE
  '((SUCCESSORS
     (LAMBDA (NODE OPERATORS)
     (MAPCAR OPERATORS
             (FQUOTE
             (LAMBDA (OPERATOR)
             (LIST (ADD1 (SD NODE)) (CONS OPERATOR
                                     (QS NODE)))))))))))
```

```
(MOREBOX '(
  (BIG SMALL FAT THIN)
  (STUPID SMART)
  (BANDIT POLICEMAN MOVIE-GIRL)
        ))
```

As an exercise, please run this program and think how it can be modified to generate only some subsets of the Cartesian Product, defined by some additional constraints.

8.2 Breadth-First Strategy

Breadth-First Search Strategy is defined by the following pseudo-code.

1) Place description of initial state in list OPEN.
2) STATE = (CAR OPEN), OPEN = (CDR OPEN).
3) Create list OP = (op_1, ..., op_n) of operators applicable to STATE. Apply each of them to STATE and create list NEWSTATES of states obtained by applying operators.
4) OPEN = (APPEND OPEN NEWSTATES)
5) If there exist STATE $\in$ NEWSTATES that fulfills the end conditions, then print solution and terminate.
6) If OPEN = NIL, then print 'no solution to the problem'.
7) Go to 2.

As an exercise, write Lisp program that searches space from the previous problem using this strategy.

8.3 Strategy of Ordered Search

Strategy of Ordered Search is specified by the following pseudo-code.

1) Place nodes $n_{o_i} \in S_0$ in list OPEN in the increasing order of $\widehat{f}(n_{oi})$.
2) If OPEN = NIL, print *"no solution."*
3) Select node $n \in OPEN$ for which $\widehat{f}(n) = min_{y \in OPEN} \widehat{f}(y)$ Delete n from OPEN, add n to CLOSED.
4) If $n \in S_k$, then print the solution (state or path).
5) Find $successors(n) = \Gamma(n)$.
6) If $successors(n) = NIL$, then go to 2.
7) Take n_i from $successors(n)$, delete it from $successors(n)$.
8) Calculate $\widehat{f}(n_i)$.
9) If $n_i \in OPEN \cup CLOSED$, place n_i in OPEN together with its $\widehat{f}(n_i)$.
10) If $n_i \in OPEN$, then assign as a value of $\widehat{f}$ the smaller value from the stored one and the new value calculated in step 8.
11) If $n_i \in CLOSED$, then as a value of $\widehat{f}$ assign the smaller value of the stored one and the new value calculated in step 8.
12) Else go to 6.

$\widehat{f}_i(n_j)$ - value of $\widehat{f}(n_j)$ when n_i is closed.
As an exercise, write a Lisp program to solve the covering problem using this strategy.

8.4 Strategy of Equal Costs

Let us first denote:

$$(\forall i, n)[\widehat{f}_i(n_k) \ = \ \widehat{g}_i n_k)]$$

The following Theorem holds:

Theorem 8.1
Strategy of equal costs gives by closing node n the shortest path from S_0 to n. It holds also that $\widehat{g}_i(n_i) \ = \ g(n_i)$.

Extension of the strategy' $S_k \cup OPEN$ = NIL?
The A* Algorithm of Nilsson.

1.

$$h(n) \ \geq \ \widehat{h}(n) \ > \ 0 \tag{8.1}$$

2.

$$\widehat{h}(m) \ - \ \widehat{h}(n) \ < \ h(m,n) \tag{8.2}$$

where $h(m,n)$ is the cost of the shortest path from m to n, where $m, n \ \notin \ S_0 \ \cup \ S_k$.

The first relation means that function $\widehat{h}$ should evaluate the real distance of n from the terminal node from bottom. The second relation requires that function $\widehat{h}$ should also evaluate each part of the path in the same way.

Finding of the heuristic function which fulfills these conditions is usually difficult. It is however worth considering while the ordered search algorithm gets convergent.

Theorem 8.2
Strategy of ordered search which applies the function $\widehat{f} \ = \ \widehat{g} \ + \ \widehat{h}$ where $\widehat{h}$ fulfills relations 8.1 and ?? is convergent, i.e. finds the minimal solution. It holds also

$$\widehat{g}_i(n_i) \ = \ g_i(n_i) \tag{8.3}$$

It comes also from condition $\widehat{g}_i(n_i) = g_i(n_i)$ that each closed node has minimal value of $\widehat{g}$, then the steps 10 and 11 of the algorithm can be omitted. These steps cannot be omitted when 8.1 and 8.2 are not fulfilled, especially when the method of calculating $\widehat{h}$ changes in the process of extending tree.

The above theorem guarantees finding the optimal solution when the algorithm terminates in step 4. It doesn't however say anything of the search efficiency. It is then important to find conditions by which the algorithm searches the optimal path, generating as small amount of nodes as possible. The following theorem points out the importance of function $\widehat{h}$, whose selection essentially influence the quality of algorithm's behavior.

Let ST_1 and ST_2 be the A^* Nilsson strategies, $\widehat{h}_1$ and $\widehat{h}_2$ be their heuristic functions.

We will say that the ST_1 strategy is *not worse defined* than strategy ST_2 if for all nodes n it holds:

$$h(n) \ \geq \ \widehat{h}_1(n) \ \geq \ \widehat{h}_2(n) \ \geq \ 0 \tag{8.4}$$

i.e. that both strategies evaluate h from bottom but $\widehat{h}_1$ does this more exactly than $\widehat{h}_2$.

Theorem 8.3 *If ST_1 and ST_2 are A^* Nilsson strategies and ST_1 is not worse defined than ST_2 then for each solution space the set of nodes closed by ST_1 is included in the set of nodes closed by ST_2.*

This theorem concludes then in other words, that if we limit ourselves to the set of all A^* Nilsson strategies then there exists one of them, better than all the remaining, because it closes not more nodes than any other strategy. This is the strategy which most accurately evaluates h (satisfying of course conditions 8.2 and 8.3). (OLD B.3 and B.4???)

Theorem 8.4 *A^* Nilsson strategy with function $\widehat{h} = h$ generates only the nodes on the paths to minimal solutions.*

Finding good heuristic function $\widehat{h}$ is difficult. It applies specific information of the problem solved, which characterizes the global properties of the state space. On the other hand, the heuristic function should be applied in the case when this information is not available. Creating a complex function $\widehat{h}$, which considers different specific information available to us, we get closer to h. However, the time for tree generation increases, while calculating complex $\widehat{h}$ is time consuming. This controversy must be solved in each case by the programmer with means of his experience, results of experiments with the program and common sense. In practice, often the computer resources do not permit for finding the optimal solution in step 4 of the algorithm. The method discussed already for the algorithm of equal costs can be applied in such case. The determination of terminating moment of the algorithm with the quasioptimal solution can be done with the use of different heuristic methods, including some discussed below.

The strategy of the described properties can be also created in a somewhat different method.

Theorem 8.5 *If the following conditions are fulfilled*

1) $(\forall n)\ [\widehat{h}(n) \ = \ \widehat{f}(n) \ - \ \widehat{g}(n) \ \geq \ 0]$

2) *(*$\forall m$ *being the far successor of* n*)*
$[g(n) \ \leq \ g(m) \quad \wedge \ f(n) \ \leq \ f(m)]$

3) *(*$\forall m \ \in \ S_k$ *and being the far successor of* n*)*
$[\widehat{f}(n) \ \leq \ \widehat{g}(m) \ = \ g(m) \ = \ f(m)]$
then the search strategy defined with such functions is convergent and optimal, and it also holds

$$(\forall n)[\widehat{h}(n) \ \leq \ h(n)] \tag{8.5}$$

8.5 Bidirectional Search

The number of nodes created in the solution space grows very quickly with the depth (usually it grows exponentially. If then the final state as well as the initial state are directly specified, it is advised to use one of the methods for bidirectional search [402, 403, 96, 404]. These methods assume that the two trees which grow from two sides should generate a smaller total amount of nodes (if they will meet approximately half of the way) then one tree growing from the start or from the goal, of the twice higher depth. The solution in the space specified as above is always some path. The cost function should be specified for both directions forward (F) and backward (B). We will denote them by $\widehat{f}_F$ and $\widehat{f}_B$. Other notation:

- a_{min} - current minimum of the path cost.
- $OPEN\_BACKWARD$ - set of nodes open in the backward direction of the search.
- $CLOSED\_BACKWARD$ - set of closed nodes in direction BACKWARD.
- $\widehat{f}_F(n) \ = \ g_F(N) \ + \ \widehat{h}_F(n)$
- $\widehat{f}_B(n) \ = \ g_B(n) \ + \ \widehat{h}_B(n)$
  where $\widehat{h}_F$(n) and $\widehat{h}_B$(n) are respectively the costs of minimal paths n, ..., n_k and n_0, ..., n.

The **Pohl's bidirectional search strategy** is defined by the following pseudo-code.

1) Place n_0 in OPEN and calculate $\widehat{f}_F(n_0)$. Place n_k in OPEN_BACKWARD and calculate $\widehat{f}_B(n_k)$. Assume $\alpha_{min} \ := \ \infty$.
2) If OPEN $\cup$ OPEN_BACKWARD = NIL, then print "no solution" and stop. Select direction of generation F or B. If F, then go to 3, else go to 11.
3) Select such node from OPEN that $\widehat{f}_F(n) \ = \ min_{y \in OPEN} \ \widehat{f}_F(y)$. Remove n from OPEN and insert it to CLOSED.
4) Calculate $successors(n) := \Gamma(n)$.
5) If $successors(n)$ = NIL, go to 12.
6) Select x from $successors(n)$, delete it from $successors(n)$.

7) If $x \notin OPEN \cup CLOSED$, then insert x to OPEN.

8) If $x \in$ OPEN, then take for x the smaller of the new and the old value of $\widehat{f}_F(x)$.

9) If $x \in CLOSED$, then take for x the smaller of the new and the old values of $\widehat{f}_F(x)$. Delete x from CLOSED and insert x to OPEN.

10) Go to 5.

11) Select such node $n \in$ OPEN_BACKWARD that

$\widehat{f}_B(n) = min_{y \in OPEN\_BACKWARD}\ \widehat{f}_B(y)$

Delete n from OPEN_BACKWARD and insert it to CLOSED_BACKWARD. Execute steps 3 to 10 of the algorithm in which n_0, CLOSED, OPEN, and Γ are replaced with n_k, CLOSED_BACKWARD, OPEN_BACKWARD and $\Gamma^{-1}(n)$, respectively.

12) If $n \in CLOSED \cap CLOSED\_BACKWARD$, then assume

$$\alpha_{min} := min(\alpha_{min},\ g_F(n) + g_B(n)) \tag{8.6}$$

13) If $\alpha_{min} \leq max[min_{n \in OPEN}\ \widehat{f}_F(n), min_{n \in OPEN\_BACKWARD}\ \widehat{f}_B(n)]$ then α_{min} is the length of the minimal path, end.

14) Go to 2.

For determination of the direction of tree growing in step 2 of the algorith, one of the following methods can be applied, respectively on the application.

1) If the depth in direction $F \leq$ the depth in direction B, then go to 3 else go to 11 (we aim at having equal depths).

2) If $CARD(OPEN) \leq CARD(OPEN\_BACKWARD)$, then go to 3, else go to 11 (we aim at equal numbers of nodes in both trees). This is advised for the state spaces as for instance this in Fig. 8.1.

3) If set OPEN includes less nodes of the smallest cost then set OPEN_BACKWARD then go to 3 else go to 11.

Pohl's algorithm executes practically two independent searches: forward to the goal node n_k and backward to the initial node n_0. The disadvantage of such an approach is that in the space which has many paths from n_0 to n_k, the trees cannot meet in the middle and at the moment of meeting be nearly as large as the two trees for one directional search.

Selection of $\widehat{h}$ which accurately estimates h cannot only improve, but even worsen the situation. It is analogical to building a tunnel from two sides, with the correction to the escape points on the other side of the tunnel. The more thin the tunnels, the more probability that the heads will meet. The goal should be then to lead the tunnels in such a way that they will not pass, but meet approximately in the middle of the way. One of the possible solutions is to find a certain state in between (not necessarily completely specified) and search towards it from both directions, while the meeting can occur not necessarily in this state but somewhere nearby. Such an approach is however not always possible. Another approach applicable always is to modify continuously function $\widehat{h}$ so that it directs the search towards the head of the opposite tree. The goal node for direction 1 is then, for instance, a node generated as the last in direction 2. There are different possibilities of solving this problem [402, 96, 404].

For instance, Champequx and Sint ?? have designed the bidirectional search algorithm different from the Pohl's algorithm.

8.5.1 Search in Decomposition Spaces

Many problems can be decomposed into simpler problems, these into even simpler problems, etc., until all problems are decomposed into elementary problems for which trivial solutions are known.

If each decomposition has the form *"problem A has solution when problem B and problem C, and problem C ... and problem Z have solutions"* then such problem can be represented with *AND graph*, whose nodes correspond to the problem and its composite problems.

If all decompositions are of the type *"problem A has solution of problem B or problem C ... or problem Z has solution"* then we have to deal with the already known solution space which can be represented as OR graph. In general cases in the decomposition space we have both kinds of decomposition - such space is represented with the AND-OR graph.

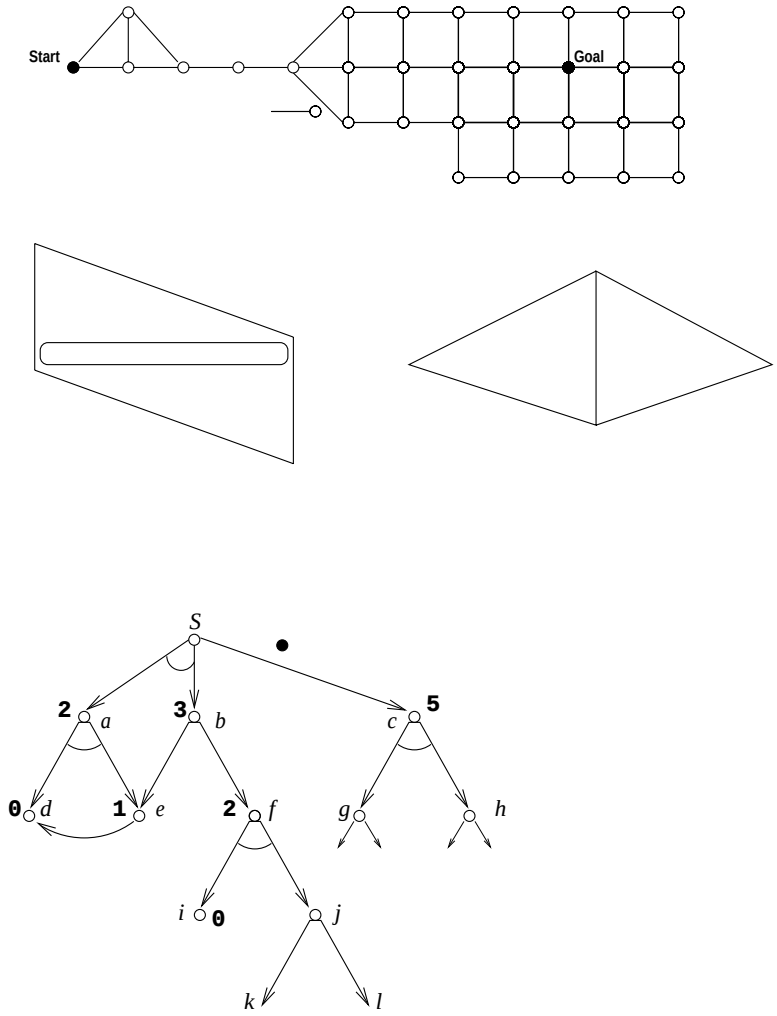


Figure 8.1: State space examples to Bidirectional Search

Figure 8.2: Example of Slagle's Symbolic Integrator

Example. Slagle's Symbolic Integrator

Example of AND/OR Tree is the symbolic integrator designed by Slagle 8.2.

Node OR has a solution when at least one of its successors has the solution. If node corresponds to the elementary problem, then without its extending, we can evaluate that it is true (has the solution) or is false. If the node appears to be false, then all its type AND predecessors are also false. If all successors of OR type node are false, then this node is also false. The solution to the problem consists in either finding such a subgraph of the AND-OR graph, whose kernel is the initial problem and whose each node has a solution, or proving that such subgraph does not exist. The cost of the subgraph is the sum of costs of all its arrows. The minimal solution is the subgraph of minimal cost. By the minimal transition graph from nodes s_1, s_2, ..., s_n to nodes t_1, t_2, ..., t_m we will call the graph from $\{s_i\}$ to $\{t_l\}$ of minimal cost.

It is infinitively clear that in OR nodes, the paths leading most quickly to the goals should be selected. The concept of using function $\widehat{f} = \widehat{g} + \widehat{h}$ as in OR graphs seems then natural. Starting out from such a concept, Chang and Slagle have designed the search strategy which can be reduced to the A^* Nilsson strategy - the convergence and optimality of Chang and Slagle's strategy results then from the properties considered previously.

Their basic concept consists in the simultaneous extension of the tree and its transformation using Boolean algebra in such a way that the closed part is the OR tree and the connections occur only in the leafs. To the node of the solution tree corresponds implicant $P = N_1, N_2, ..., N_r$ i.e. conjunction of AND-OR tree nodes for extension.

Functions $\widehat{g}$, $\widehat{h}$ and $\widehat{f}$ are specified on the implicants

$$\widehat{f}(p) = g(p) \; + \; \widehat{h}(p) \tag{8.7}$$

where

$\widehat{g(p)}$ - is the estimation of complexity of the transition graph from initial nodes to nodes n_1, n_2, ... n_r.

$\widehat{h(p)}$ - is the estimation of the complexity of the transition graph from nodes n_1, n_2, ... n_r to final nodes.

$\widehat{h}(P) \; = \; \sum_{n \in P} \widehat{h}(n) \; \leq \; h(P)$

The initial problem (or set of initial problems) is the root of the tree. The AND and OR type successors are created. The AND-OR tree can be described with the Boolean function. This function can be reduced to the normal form of snm of products - implicants. This corresponds to creating OR node, whose all successors are all AND nodes. At each step, the selection of the subsequent nodes n_1, n_2 ..., n_r is done from the corresponding to them implicant minimizing value $\widehat{f}$.

If the subproblem is elementary (can be solved at once), then it is replaced in the normal form by logic function T. The initial problem has the solution if one of the implicants is the logical value T.

Chang and Slagle's Strategy.

1) OPEN := $\{S_0\}$, CLOSED := NIL.

2) Calculate $\widehat{f}(Q)$ for each $Q \in OPEN$. Select such $P \in OPEN$ that $\widehat{f}(P) = min_{Q \in OPEN} \widehat{f}(Q)$. In the case of equality, the selection is arbitrary, but the problems corresponding to the nodes from the set of final nodes S_k and those which include such nodes are selected as first.

3) Let $P = N_1 \; N_2 \; ... \; N_r$. Let N_i are the problems connected to the nodes n_i, $i = 1, 2, ... \; r$. If all $n_i \in S_k$, then recreate the graph being the solution and terminate.

   Else

   a) delete from P all elementary problems

   b) calculate IMPLICANTS(P).

      b1) replace in P each problem N_i with the expression describing its component problems.

      b2) simplify the resulting expression $\bigcap P$ to the form of sum of products, taking into account the Boolean algebra laws.

4) $CLOSED := CLOSED \cup \{P\}$.
   $OPEN \; := \; (OPEN \; \cup \; IMPLICANTS) \; - - \; CLOSED$.

5) If OPEN = NIL, then print *"no solution"*. Else go to 2.

Figure 8.3: Potential decomposition space described using an AND-OR graph

Example 8.1

Let us assume that the potential decomposition space is described with the AND-OR graph of Fig.8.3.

We will define $\widehat{h}(P) = r - 1 + min(\widehat{h}(N_1), \widehat{h}(N_2), ..., \widehat{h}(N_r))$. The respective values of function $\widehat{h}$ are shown by the nodes. The costs of all arrows are 1. The subsequent stages of extension of AND-OR tree and the corresponding OR tree are presented in Fig. 8.4.

1) Node S is closed. OPEN = {AB, C}, CLOSED = {S}, IMPLICANTS = {AB, C}, $\widehat{h}(AB) = 2 - 1 + min(\widehat{h}(A), \widehat{h}(B)) = 2 - 1 + min(2,3) = 3$, $\widehat{f}(AB) = g(AB) + \widehat{h}(AB) = 2 + 3 = 5$. Analogically $\widehat{f}(C) = 6$. AB is selected as one with minimal value of $\widehat{f}$.

2) Nodes a and b are closed. $A \wedge B \rightarrow (D \wedge E) \wedge (E \vee F) = (D \wedge E \wedge E) \vee (D \wedge E \wedge F) = D \wedge E \vee D \wedge E \wedge F = D \wedge E \wedge (T \vee F) = D \wedge E$.

   DE is then selected. $\widehat{f}(DE) = \widehat{g}(DE) + \widehat{h}(DE) = 5 + 2 - 1 + min(\widehat{h}(D), \widehat{h}(E)) = 5 + 1 + min(0,1) = 6$.

   Now nodes DE and C have equal value of $\widehat{f}$. DE is selected while $d \in S_k$.

3) P = DE. D is removed, while $d \in S_k$. IMPLICANTS(E) = {D}, CLOSED = {S, AB, DE}, OPEN = {D, C}, $\widehat{h}(D) = 0$, $\widehat{f}(D) = 6 + 0 = 6, \widehat{f}(D) = \widehat{f}(C)$. D is the final node and is then selected.

The algorithm terminates its execution with the subgraph from Figure 8.4 as the minimal solution.
The properties of the Chang and Slagle strategy are analogical to the A^* Nilsson strategy.

Theorem 8.6 *If $\widehat{h}(Q) \leq h(Q)$ for all Q being implicants of S, then for all graphs in which the cost of the shortest path is not smaller than S, the Chang and Slagle strategy is convergent. (OLD is it S or sigma?, delta?)*
To prove the optimality of the strategy, some additional restrictions on the heuristic function are necessary.
For all P being implicants of S and including problems connected with final nodes

$$\widehat{h}(P) = 0 \tag{8.8}$$

For all Q being implicants of S and all P being implicants of Q

$$\widehat{h}(Q) - \widehat{h}(P) \leq k(Q,P) \tag{8.9}$$

where $k(Q,P)$ is the cost of the least expensive transition graph from $\{g_i\}$ to $\{p_j\}$

Theorem 8.7 *Let ST_1 and ST_2 be two convergent Chang and Slagle strategies, controlled with functions $\widehat{h}_1$ and $\widehat{h}_2$. If:*

1) For all P being implicants of S and including at least one problem connected with some node
$\widehat{h}_1(P) > \widehat{h}_2(P)$

2) For each graph fulfilling equations 8.8 and 8.6 and the previous theorem, and having the minimal subgraph being a solution.

then
Each implicant selected by ST_1 is also selected by ST_2 and each node closed by ST_1 is also closed by ST_2.

8.6 Problems in using Lisp for Search Theory

Some of these problems are more complex and can be used as final exam (take-home) or class projects.

1. . Explain in your own words and illustrate on your own examples.

   a) Definition of Lisp list structures.

   b) Elementary Lisp functions and their graphical interpretation.

   c) Uniform structure of programs and data in Lisp and its applications.

   d) Functions modifying structure (`RPLACA, RPLACD, NCONC`). Explain and give examples of applications.

   e) The principle of dynamic memory allocation and Garbage Collector.

Figure 8.4: Subsequent stages of extension of AND-OR tree and the corresponding OR tree

f) Defining functions in Lisp.

g) Lambda-expressions. Free and bounded variables.

h) Define functions `NULL, MEMBER, LIST, APPEND, PAIRLIS` and `SASSOC` using only functions `CAR, CDR, CONS, COND, EQ, ATOM, QUOTE, DEFLIST.`

i) Functionals, functional arguments. Examples and applications.

j) `FEXPR` functions and their applications.

2. Define statement `DO` from FORTRAN and function `PROGN` from Lisp as `FEXPR` functions.

3. Write the recursive program `DERIVE`, called as follows: `(DERIVE EXPRESSION VARIABLE)` which symbolically differentiates expression `EXPR` (in prefix Lisp notation), with respect to variable `VAR`, without algebraic simplification of results. Do not use `PROG`.

4. Write recursive function (without `PROG`) which simplifies the prefix notation predicates, applying the Boolean algebra laws:

```
(OR A...A) ==> A
(AND A..A) ==> A
(OR A..A0) ==> A
(AND A...0...) ==> 0
(OR A...1...) ==> 1
(AND A...1...1) ==> A
(OR A 0...0) ==> A
(AND A..0..) ==> 0
(NOT o) ==> 1 (NOT n) ==> 0
(NOT (AND A B...Z)) ==> (OR (NOT A)...(NOT 2))
(NOT (OR A B...Z)) ==> (AND (NOT A)...(NOT 2))
```

5. Give examples and explain application of property list to knowledge representation. Find an example related to hardware and logic design.

6. The knowledge representation problem. Proceduralists versus declarativists. Give arguments of both sides. Apply to any logic design problem.

7. Describe any of the above in term of implementing in Lisp.

   a) Semantic networks.

   b) Relational structures.

   c) Frames.

   d) Predicate calculus as representation of knowledge.

   e) Data bases in AI languages.

8. Draw the semantic network for the text. The exam from EE510VHDL is easy. It is easier than the exam from EE573. Everyone who wants to pass this exam can pass it with little effort.

   Assume primitives: OBJECT, EVENT, PROPERTY, etc. Present typical questions to data base.

9. Given are three towers (Hanoi Towers problem), Figure 8.5.

   The initial state of the world (for $N = 3$) is presented in Fig. 8.6. Write program `HANOI(N)` which finds a path from initial state to the final state when all discs are on Tower C. In any of the states the larger disc can be above the smaller one. One disc can be moved at the moment. Consider the following variants:

   A) Recursive program in Lisp

   B) Lisp iterative program searching the OR-tree.

   C) Lisp program searching the AND-OR tree. AND-OR tree is the value of variable `TREE`.

10. * Discuss models of a problem in Lisp. Concentrate on simulation, fault detection and localization, and Boolean logic.

Figure 8.5: Three towers (Hanoi Towers problem)

Figure 8.6: Initial state of the world (for N = 3)

11. Discuss search strategies: breadth first, depth-first, ordered search, k-successors, branch and bound, equal costs, A* Nilsson, switching strategies. Implement any of them in Lisp. Discuss the properties of basic search strategies: relations on operators, the cost functions versus the quality functions for states and operators.

12. The extensions of the tree search model. The role of heuristics. When is the bidirectional search applicable? Write general Lisp algorithm.

13. *** Nondeterministic programs in Lisp. Write a nondeterministic Lisp interpreter based on backtracking similar to Prolog.

14. The tree has in each node m successors and depth r. Each branch terminates with the solution on depth r. Write the expressions comparing the number of closed and open nodes for the basic strategies.

15. Given is list `PRECEDENCE` of precedence of two-argument operators (from the operators of highest precedence). For instance, for arithmetics it will be (* / + -). Write recursive program `TRANS` in Lisp (do not use `PROG`) which translates the prefix, parentheses notation into Lisp notation. For instance

```
              trans
   (A * B + C) ==> (PLUS (TIMES A B) C)
```

Advice.

Search the operators in the input list starting from the operators of the smallest precedence and apply the transformation

$$(TRANS\ `(E1\ ...\ E_n\ OPERATOR\ E_{n+1}\ ...\ E_{n+m}))$$

$$\Longrightarrow\ (OPERATOR\ (TRANS\ `(E1\ ...\ En))\ (TRANS\ `(E_{n+1}\ ...\ E_{n+m})))$$

16. Write the recursive or tree searching Lisp program which generates all subsets of a set. How would you write a generator which generates one subset by one call?

17. * Place 8 queens on the chessboard that each two are not beating. Write

    a) recursive Lisp program,

    b) backtracking program in Lisp that may be similar to Prolog.

18. Write the program which accepts the language described by the grammar

```
S --> A | B
A --> aAa | bAb| b
B --> cB | cA | c
```

19. The railroad net is described by the graph described with assertions:

```
CONNECTION (NEW YORK, BOSTON)
CONNECTION (PHILADELPHIA, NEW YORK), etc.
```

    Write Prolog-like program in Lisp, `PATH (*A, *b)` which finds all paths from `*A` to `*B`.

20. Find shortest path in railroad net from A to B assuming that the assertions are like:

```
CONNECTION (NEW YORK, BOSTON, 263)
```

    where the third argument is the length of the connection.

21. Given is the propositional calculus expression in the prefix form. Write a Lisp program which will check to see if this expression is the tautology of propositional calculus using the method of truth table (matrices).

22. * Given is a plan of big city. Write a Lisp program for planning a way from point A to point B. Consider the representation of knowledge. How would you introduce the data? What are the advantages and disadvantages for each of the presented formalisms?

23. * There are 2000 modules of electronic devices in the room of a railroad switching station, which includes 50 x 4 places for stands. Each stand can have not more than 20 modules. Disposing the multigraph of connections of modules (two modules can be connected with more than two connections) we have to find such placement of modules in stands and stands in the room that the sumaric length of connections is minimal. The among stands wires can go only on the floor. Assume that the modules are cubes and the i/o pins of the connections are on the middle of one of the side walls of the cubes (assume north). Write the Lisp program for an optimal solution and for an approximative solution. Consider different state spaces and methods of search in them. Estimate the computation complexity of the algorithms.

24. Write a Lisp program to solve the following problem: Three missionaries and three cannibals are going to pass the river. All of them can row a boat. Not more than two people can be in the boat at the same time. If there are more cannibals than missionaries at any place, then the last one could be in trouble. How should the missionaries organize the transfer?

25. * Generalize the previous program for the problem with n cannibals and n missionaries, m people in a boat maximally.

26. What is the program `x`, given below, doing?

```
(defun x (list)
  (cond ((null list) nil)
(t (append (x (cdr list)) (list (car list))))))
```

Chapter 9

Advanced Search Methods

9.1 Introduction to Advanced Search Methods

One of the most important components to create successful programs for many CAD applications is developing a good search strategy, that is based on the problem and its heuristics.

In principle, better search methods either use some kind of heuristics, or utilize some **systematic** search strategies that **guarantee**, at least local, optima. One convenient and popular way of describing such strategies is to use the concepts of **tree searching.**

The problem of creating complex heuristic strategies to deal with combinatorial problems is very similar to that of general problem-solving methods in Artificial Intelligence. There are five main principles of problem solving in AI:

- state-space approaches,
- problem decomposition,
- automatic theorem proving,
- rule-based systems,
- learning methods (neural nets, abductive nets, immunological, fuzzy logic, genetic algorithm, genetic programming, Artificial Life, etc.).

Since we will limit the discussion in this report to the description of the state-space principle, the approach used is based on the assumption that any combinatorial problem can be solved by searching some space of states. This seems to be the best approach because of its simplicity and generality. By using this approach, problems within this framework that we are concerned with are not greatly restricted

There are other reasons for choosing the state-space heuristic programming approach:

- The combinatorial problem can be often reduced to integer programming, dynamic programming, or graph-theoretic problems. The graph-theoretic approaches include in particular, the set covering, the maximum clique, and the graph coloring. The computer programs that would result from pure, classical formulations in these approaches would not sufficiently take into account the specific features and heuristics of the problems. Instead reducing to known models, we will create our own general model, and "personalize" it to our problems. Thus, instead of normal graph coloring, we will formulate the compatible graph coloring problem, and moreover, we will use heuristics based on our data to solve this problem efficiently. The problems are rather difficult to describe using these standard formulations. The transition from the problem formulation, in these cases, to the working version of the problem-solving program is usually not direct and cannot be automated well. It is difficult to experiment with strategy changes, heuristics, etc., which is our main goal here. We aim at the model's flexibility, and being able to easily tune it experimentally. In a sense, we are looking at a "fast prototyping" possibility.
- Some of these combinatorial problems (or similar problems) have been successfully solved using the state-space heuristic programming methods. The state-space methods include some methods that result from other AI approaches mentioned above. Some backtracking methods of integer programming, and graph-traversing programs used in graph partitioning and clustering methods, are for instance somewhat similar to the variable partitioning problem. They can be considered as special cases of the problem-solving strategies in the space of states.

Roughly speaking, several partial problems in logic CAD can be reduced to the following general model:

- The rules governing the generation of some set S, called *state-space,* are given.

- *Constraint conditions* exist which, if not met, would cause some set $S' \in S$ to be deleted from the set of potential solutions.
- The *solution* is an element of S that meets all the *problem conditions.*
- The *cost function* F is defined for all solutions.
- The solution (one, more than one, or all of them) should be found such that the value of the cost function is *optimal (quasi-optimal)* out of all the solutions.

A problem condition pc is a function with arguments in S and values $true$ and $false$. For instance, if set S is the set of natural numbers:

$$pc_1(x) = true - if\ x\ is\ a\ prime\ number;\ \ false - otherwise$$

In general, a *problem* can be defined as an ordered triple:

$$P = (S, PC, F), \tag{9.1}$$

where:

1. PC is a set of predicates on the elements of S, called *problem conditions,*
2. F is the cost function that evaluates numerically the solutions. Solution is an element of S that satisfies all the conditions in PC.

The *tree search method* includes:

- the problem P,
- the constraint conditions,
- *additional solution conditions* that are checked together with the problem conditions,
- the *generator of the tree,*
- the *tree-searching strategy.*

Additional solution conditions are defined to increase the search efficiency. For instance, assume that there exists an auxiliary condition that is always satisfied when the solution conditions are satisfied, but the auxiliary condition can be tested less expensively than the original solution conditions. In such case the search efficiency is increased by excluding the candidates for solutions that do not satisfy this auxiliary solution.

The additional solution conditions together with the problem conditions are called *solution conditions.* The method is *complete* if it searches the entire state-space and thus assures the optimality of the solutions. Otherwise, the entire space is not searched and the search method will be referred to as *incomplete methods.* Obviously, for practical examples most of our searches will use incomplete search methods.

To illustrate these ideas of the minimal covering problem, we use several applications. For instance:

1. the state-space S is a set that includes all of the subsets of the set of rows of the covering table (rows correspond to prime implicants contained in the function [257]).
2. The solution is an element of S that covers all the columns of the function.
3. A cost function assigns the cost to each solution. The cost of a solution is the number of selected rows. It may also be the total sum of the selected rows and their costs.
4. A solution (set of rows) should be found that is a solution and minimizes the cost function.
5. Additional quality functions are also defined that evaluate states and rows in the search process.
6. This process consists of successively selecting "good" rows (based on the value of the quality function), deleting other rows that cover fewer of the matrix columns (these are the *dominated rows*), and calculating the value of the cost function.
7. The cost value of each solution cover found can then be used to limit the search by *backtracking.*

8. This process can be viewed as a search for sets of rows in the state-space, and can be described as a generation of a tree *(solution tree)* using rows as *operators,* sets of rows as *nodes of the tree,* and solutions as *terminal nodes.*

A combinatorial problem of a set covering type can either be reduced to a covering table, or solved using its original data structures.

It has been shown in many former publications, that the following logic synthesis problems can be reduced to the Set Covering Problem.
These problems are:

(1) the PLA minimization problem,

(2) the finding of vacuous variables,

(3) the column minimization problem,

(4) the microcode optimization problem,

(5) the data path allocation problem,

(6) the Three Level AND/NOT Network with True Inputs (TANT) minimization problem,

(7) the factorization problem,

(8) the test minimization problem, and many other logic synthesis problems.

Therefore, the Set Covering Problem can be treated as a *generic logic synthesis subroutine.* Several efficient algorithms for this problem have been created [60, 61, 179, 222, 409, 530]. We can use the search ideas from this chapter to solve efficiently all these problems. Equivalently, we believe that some of the ideas originated in these papers can also be used to extend the search framework presented by us.

Moreover, various methods of reducing a problem to the Set Covering Problem exist. These methods would result in various sizes of the covering problem. By a smart approach, the problem may still be NP-hard, but of a smaller dimension. For a particular problem then, one reduction will make the problem practically manageable, while the other reduction will create a non-manageable problem. This is true, for instance, when the PLA minimization problem is reduced to the set covering of the signature cubes [Brayton] as columns of the covering table, instead of the minterms as the columns. Such reduction reduces significantly the size of the covering table. Similar properties exist for the Graph Coloring, Maximum Clique, and other combinatorial problems of our interest. Although the problems are still NP-hard as a class, good heuristics can solve a high percent of real life problems efficiently. This is because of the Occam's razor principle.

Many other partial problems in high-level synthesis, logic synthesis, and physical CAD can also be reduced to a class of NP-hard combinatorial problems that can be characterized as *constrained logic optimization problems.* They are described using multi-valued Boolean functions, graphs, or arrays of symbols. On this data, some constraints are formulated and some transformations are executed in order to optimize cost functions. These problems include Boolean satisfiability, tautology, complementation, set covering [222], clique partitioning [?], maximum clique [?], generalized clique partition, graph coloring, maximum independent set, set partitioning, matching, variable partitioning, linear and quadratic assignment, encoding, and others.

With respect to high importance of these problems, several different approaches have been proposed to solve them. These approaches include:

1. Mathematical analysis of the problems are done in order to find the most efficient algorithms (exact or approximate). If this cannot be achieved, the algorithms for particular sub-classes of these problems [?] are created. This can speed up solving problems on large classes of practical data, in spite of the fact that the problems are NP-hard so that no efficient (polynomial) algorithms exist for them. For instance, the proper graph coloring problem is NP-hard, but for a non-cyclic graph there exists a polynomial algorithm. How practical the polynomial algorithm is depends only on how often non-cyclic graphs are met in any given area of application where the graph coloring is used.

2. Special hardware accelerators are designed to speed-up the most executed or the slowest operations on standard types of data used in the algorithms.

3. General purpose parallel computers, like message-passing hypercubes, SIMD arrays, data flow computers and shared memory computers are used [94].

4. The ideas of Artificial Intelligence, computer learning, genetic algorithms, and neural networks are used, also mimicking humans that solve these problems [?].

In the future, we plan to use our FPGA-based Universal Hardware Accelerator from DEC, PERLE 1, to implement a special hardware architecture that will speed up the design process by speeding up the processes that are repeated the most, and contribute the most to the quality of results (perhaps checking the column compatibility, or the set covering). First, we have to develop some good respective models of our methods.

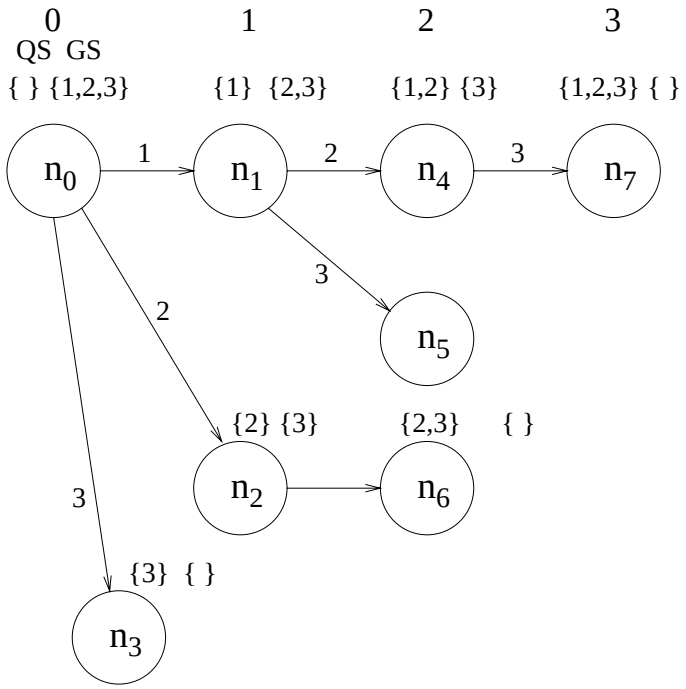


Figure 9.1: Example of T_1 type tree generator of a full tree

9.2 Multistrategical Combinatorial Problem Solving

9.2.1 Basic Ideas

The goal of this section is to explain how the general objectives outlined above can be realized in programs to solve combinatorial problems. Some of them have been already programmed, some other not yet. Our interest is in a uniform explanation and the creation of state-space tree search methods. Our first goal is *Fast Prototyping.* By fast prototyping, we want the program to be written in such a way that the developer will be able to easily change the program for each experiment. This includes changing the problem description variants and create various search strategies for different tree search methods to optimize the efficiency of the search.

The tree-search strategy was created by selecting respective classes and values of *strategy parameters.* The creation of multiple variants of a tree-searching program, that could require weeks of writing and debugging code would then be possible in a shorter period of time. Some efficiency during execution will be lost, but the gain of being able to test many variants of the algorithm will be much more substantial. The *behavior* of the variants of the tree search methods will then be compared and evaluated by the developer to create the more efficient algorithms.

Of course, these search programs will all use our *basic logic design subroutines* that include those that realize operations on *cubes* or *cube arrays,* like *sharp, intersection, or consensus* [**?**, **?**]. It also uses BDD subroutines, partition subroutines, and all other partial subroutines that already exist in our software.

9.2.2 Description of the Solution Tree

The search strategy realizes some design task by seeking to find a set of solutions that fulfill all problem conditions. It checks a large number of partial results and temporary solutions in the tree search process, until finally it determines the optimality or the quasi-optimality of the solutions.

1. The *state-space* S for a particular problem solved by the program is a set which includes all the solutions to the problem. The elements of S are referred to as *states.* New states are created from previous states by application of operators. During the realization of the search process in the state-space, a memory structure termed *solution tree, solution space,* is used.

2. The solution tree is defined as a graph:

$$D = [NO, RS]$$

1. A solution tree contains *nodes* from set NO, and *arrows* from the set of arrows RS. Nodes correspond to the stages of the solution process (see Figure 9.1.)

2. Each arrow is a pair of nodes (n_{i1}, n_{i2}). Arrows are also called *oriented edges*. They correspond to transitions from stages to stages of the solution process.

3. An *open node* is the node without created *children*, or immediate successors. A child of child is called *grandchild.* If s is a child of p then p is a *parent* of s. A *successor* is defined recursively as a child or a successor of a child. A *predecessor* is defined recursively as a parent or a predecessor of a parent.

4. A *semi-open node* is a node that has part of its children created, but not yet all of its children are implicitely formed.

5. A *closed node* is a node, where all of its children have already been created in the tree.

6. The set of all nodes corresponding to the solutions will be denoted by S_F.

3. In the solution space we can distinguish the following sub-spaces:

- **actual solution space** - the space which has a representation in the computer memory (both RAM and disk),
- **potential solution space** - the space that can be created from the actual space using operators and taking into account constraints,
- **closed space** - the space which has already been an actual space for some time, but has been removed from the memory (with exception of the solutions).

4. As the search process grows, the actual space is at the expense of the potential space. The closed space grows at the expense of the actual space. The actual space is permanently modified by adding new tree segments and removing other segments. Sometimes the closed space is saved in hard disk, and re-used only if necessary.
5. By "opening a node" we will mean creating successors of this node. The way to expand the space, called the *search strategy*, is determined by: (1) the way the open and semi-open nodes are selected, (2) the way the operators applied to them are selected, (3) the way the termination of search procedure is determined, (3) the conditions for which the new search process is started, and (4) the way the parts of the space are removed from the memory.
6. The arrows in the tree are labelled by the *descriptors of the operators.* Each node contains a description of a state-space state and some other search-related information. In particular, the state can include the data structure corresponding to the *descriptors of the operators* that can be applied to this node. Descriptors are some simple data items. For instance, the descriptors can be: numbers, names, atoms, symbols, pairs of elements, sets of elements. The choice of what the descriptors are, is often done by the programmer. Descriptors are always manipulated by the search program. (In some problems, they are also created dynamically by the search program.) Descriptors can be stored in nodes or removed from the descriptions of nodes. As an example of using descriptors, we will discuss the case where the partial solutions are the sets of integers. In this problem then, the descriptors can be the pairs of symbols (*aritmetic_operator*, *integer*). The application of an operator consists in taking a *number* from the partial solution and creating a new number. This is performed like this:

$$new\_number \;:=\; number\ aritmetic\_operator\ integer$$

The *number* is replaced in the partial solution of the successor node by the *new_number*.
6. In those cases that the descriptors are dynamically created, the programs that create them are called the *descriptor generators*. They generate descriptors for each node one-by-one, or all of them at once. The *operators* traverse the tree from a node to a node. Operator is a concept that corresponds to applying certain program to nodes of the solution tree. This program has the descriptor as its parameter. Creating new nodes of the tree is equivalent to searching among the states of S.
7. Each of the solution tree's nodes is a *vector of data structures.* For explanation purposes, this vector's *coordinates* will be denoted as follows:

- N - the node number,
- SD - the node depth,
- CF - the node cost,
- AS - description of the hereditary structure,
- QS - partial solution,
- GS - set of descriptors of available operators.

8. Additional coordinates can then be defined, of course, as they are required.
Other notations used:

- NN - the node number of the immediately succeeding node (a child),
- OP - the descriptor of the operator applied from N to NN,
- NAS - actual length of list AS,
- NQS - actual length of list QS,
- NGS - actual length of list GS.

9. The operator is denoted by OP_i, and it's corresponding descriptor by r_i. An application of operator OP_i with the descriptor r_i to node N of the tree is denoted by $O(r_i, N)$. A *macro-operator* is a sequence of operators that can be applied successively without retaining the temporarily created nodes.

A prerequisite to formulating the combinatorial problem in the search model is to ascertain the necessary coordinates for the specified problem in the *initial node* (the root of the tree). The way in which the coordinates of the subsequent nodes are created from the preceding nodes must be also found. This leads to the description of the *generator of the solution space (tree generator).* Solution conditions and/or cost functions should be formulated for most of the problems. There are, however, generation problems (such as generating all the cliques of a specific kind), where only the generator of the space is used to generate all the objects of a certain kind.

- QS is the partial solution: that portion of the solution that is incrementally grown along the branch of the tree until the final solution is arrived at. A set of all possible values of QS is a state-space of the problem. According to our thesis, some relation $RE \in S \times S$ of partial order exists usually in S. Therefore, the state s $\in$ S symbolically describes the set of all $s' \in S$ such that $s \; RE \; s'$. The solution tree usually starts with $QS(N_0)$ which is either the *minimal* or the *maximal element of S.* All kinds of relations in S should be detected, since they are very useful in creating efficient search strategies.
- The set $GS(N)$ of descriptors denotes the set of all operators that can be applied to node N.
- $AS(N)$ denotes the *hereditary structure.* By a hereditary structure we understand any data structure that describes some properties of the node N that it has inherited along the path of successor nodes from the root of the tree.
- The *solution* is a state of space that meets all the solution conditions.
- The *cost function* CF is a function that assigns the cost to each solution.
- The *quality function* QF can be defined as a function of integer or real values pertinent to each node, i.e., to evaluate its quality. It is convenient to define the cost and quality functions such that

$$QF(N) \leq CF(N) \text{ and if } QS(N) \text{ is the solution, then } QF(N) = CF(N) \tag{9.2}$$

- $TREE(N)$ denotes a subtree with node N as the root. Often function $QF(N)$ is defined as a sum of function $F(N)$ and function $\hat{h}(N)$:

$$QF(N) = CF(N) + \hat{h}(N) \tag{9.3}$$

- $\hat{h}(N)$ evaluates the distance $h(N)$ of node N from the best solution in $TREE(N)$. $F(N)$, in such a case, defines a partial cost of $QS(N)$, thus $\hat{h}(N)$ is called a *heuristic function.* We want to define $\hat{h}$ in such a way that it as close to h as possible (see [359] for general description).

A theoretical concept of function f is also useful to investigate strategies as well as cost and quality functions. This function is defined recursively on nodes of the extended tree, starting from the terminal nodes, as follows:

$$f(NN) = CF(NN) \qquad \text{when the terminal node NN is a solution from } \mathrm{S_F}, \tag{9.4}$$

$$f(NN) = \infty \qquad \text{when the terminal node NN is not a solution,} \tag{9.5}$$

$$f(N) = min(f(N_i)), \qquad \text{for all } N_i \text{ which are the children of node N.} \tag{9.6}$$

This function can be calculated for each node only if all its children have known values, which means practically that the whole tree has been expanded. $f(N)$ is the cost of the least expensive solution for the path which leads through node N. We assume that the function CF can be created for every node N (and not only for the nodes from the set S_F, of solutions), it holds that the following must also be true

$$CF(N) \leq f(N) \tag{9.7}$$

and

$$CF(NN) \geq CF(N) \qquad \text{for NN} \in \text{SUCCESSORS(N)} \tag{9.8}$$

The general idea of the **Branch and Bound Strategy** consists in having a CF that satisfies equations 9.4, 9.7, 9.8. Then, knowing a cost CF_{min} of any intermediate solution that is temporarily treated as the minimal solution, one can cutt-off all subtrees $TREE(N)$ for which $CF(N) > CF_{min}$ (or, $CF(N) \geq CF_{min}$ when we look for only one minimal solution).

In many problems it is advantageous to use a separate function QF, distinct from CF, such that CF guides the process of cutting-off subtrees, while QF guides the selection of nodes for expansion of the tree. In particular, the following functions are defined:

$g(N)$ the smallest from all the values of cost function calculated on all paths from N_0 to N.

$h(N)$ the smallest from all the values of increment of cost function calculated from N to some $N_k \in S_F$. This is the so-called **heuristic function**.

$f(N) = g(N) + h(N)$.

Since function h cannot be calculated in practice for node N during tree's expansion, and g is often difficult to find, some approximating functions are usually defined. Function CF approximates function g, and function $\hat{h}$ that approximates function h, such that

$$QF(N) = CF(N) + \hat{h}(N) \tag{9.9}$$

$$h(N) \geq \hat{h}(N) \geq 0 \tag{9.10}$$

$$h(M) - h(N) \leq h(M,N) \tag{9.11}$$

where $h(M,N)$ is the smallest of all increment values of cost function from M to N, when $M,\ N \notin S_F$.
It also holds that:

$$QF(N) = CF(N) \qquad \text{for N} \in \text{S}_\text{F} \tag{9.12}$$

$$h(N) = \hat{h}(N) = 0 \qquad \text{for N} \in \text{S}_\text{F} \tag{9.13}$$

Functions defined like this are useful in some search strategies, called *Nilsson A^* Search Strategies.* Sometimes while using branch-and-bound strategies it is not possible to entirely define the cost function $g(N)$ for $N \notin S_F$. However, in some cases one can define a function QF such that for each N

$$QF(N) \leq g(N) \tag{9.14}$$

For nodes $N \in S_F$ one calculates then $g(N) = CF(N)$, and then uses standard cut-off principles, defining for the remaining nodes N_i: $CF(N_i) = QF(N_i)$, and using function CF in a standard way for cutting-off. A second, standard role of QF is to control the selection of non-closed nodes. (By non-closed nodes we mean those that are either open or semi-open.) One should then try to create QF that plays both of these roles.

A *quasi-optimal* or *approximate solution* is one with no redundancy; i.e., if the solution is a set, all of its elements are needed. When the solution is a path in a certain graph, for example, it has no loops. An *optimal solution* is a solution $QS(N) = s \in S$ such that there does not exist $s' \in S$ where $F(s) > F(s')$. The problem can have more than one optimal solution. The set of all solutions will be denoted by SS. Additional *quality functions for operators* can also be used.

In many combinatorial problems, the set of all mathematical objects of some type are needed: sets, functions, relations, vectors, etc. For example, the following data are created:

- The set of all subsets of prime implicants in the minimization of a Boolean function.
- The set of all subsets of bound variables in the Variable Partitioning Problem.
- The set of all two-block partitions in the Encoding Problem.
- The set of maximal compatibles in the Column Minimization Problem.

These sets can be generated by the respective search routines created for them in a standard way, that use the generators of trees. This is useful in *direct problem descriptions.*

It is desirable to develop descriptor generators for several standard sets, several types of tree generators, many ordering possibilities for each generation procedure, and several tree extension strategies for each ordering. The *type of tree* is defined by two generators: the generator that creates the descriptors, and the generator that generates the tree. A *full tree* is a tree created by the generators only, ignoring *constraint conditions,* quality functions, dominations, etc. Full trees of the following types exist:

- T_1 - a tree of all subsets of a given set,
- T_2 - a tree of all permutations of a given set,
- T_3 - a tree of all one-to-one functions from a set A to set B.

and many others.

The type T_1 tree generator of the full tree of the set of all set's {1,2,3} subsets, as shown in Fig. 9.1, can be described as follows.

1. *Initial node* (root) is described as:

$$QS(N_0) = \emptyset; \tag{9.15}$$

$$GS(N_0) = \{1, 2, 3\}; \tag{9.16}$$

where $\emptyset$ is an empty set, and N_0 is the root of the tree.

2. In a recursive way, the *children* of any node N are described as follows:

$$(\forall\, r \in GS(N))\ [QS(NN) = QS(N) \cup \{r\};\quad GS(NN) = \{r_1 \in GS(N) \mid r_1 > r\}] \tag{9.17}$$

where NN is some child of node N, and r is the descriptor of the operator that creates the new nodes in this tree. Set GS is either stored in the node or its elements are generated one by one in accordance with the ordering relation $>$ while calculating the children nodes.

Figure 9.2 and Figure 9.3 present examples of full sets for many important combinatorial problems. They show the partial solutions in nodes and the descriptors in arrows. The trees in Figure 9.2 are: (a) the tree of all subsets of set, (b) the tree of all permutations of a set, (c) the tree of all binary vectors of length 3, (d) the tree of all two-block partitions of set {1,2,3,4}, (e) the tree of all partitions of set {1,2,3,4}, (f) the tree of all covers of set {1,2,3,4} with its subsets.

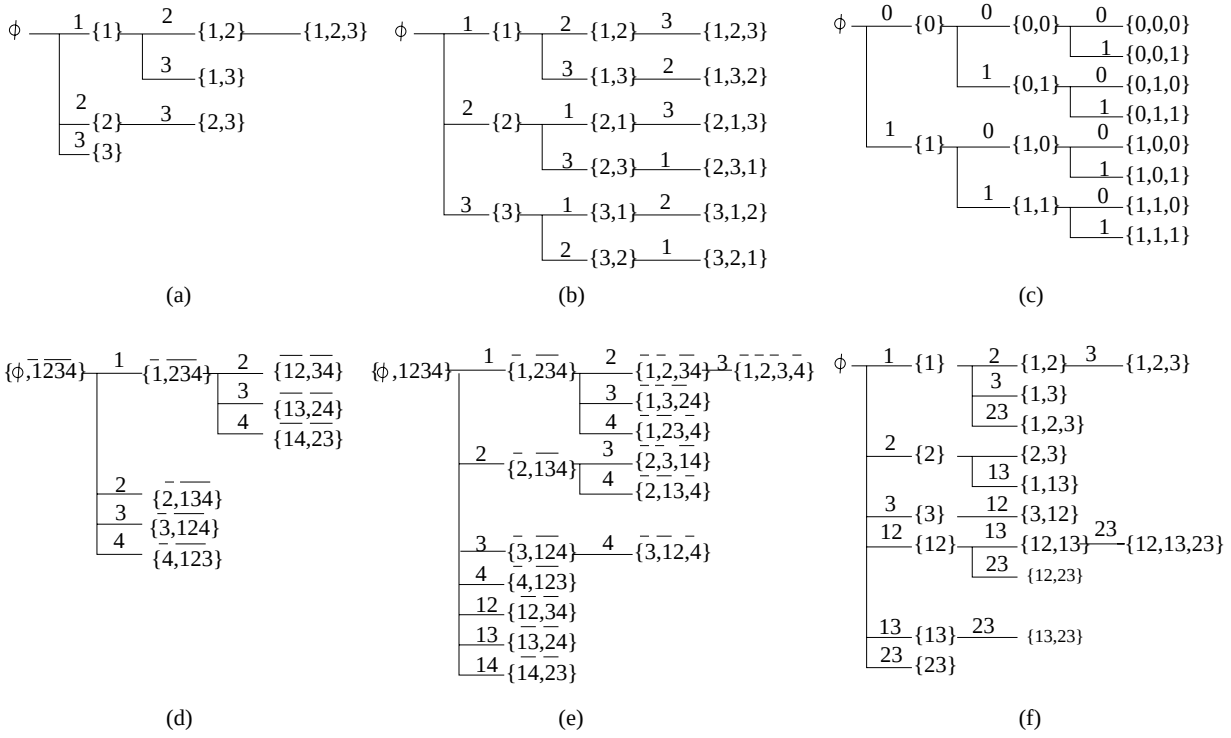


Figure 9.2: Examples of tree generators

The trees in Figure 9.3 are the following. Figure 9.3a presents the tree for all 3-element numerical vectors, such that they sum is a constant in every level. In the first level the sum is 3, in the second level the sum is 4, in the third level the sum is 5, in the fourth level the sum is 6. Figure 9.3b presents the tree of all subsets of set {1,2,3,4,5}. The tree generates levels of equal distance from the subset {1,2,3}, in the second level there are subsets that differ by one from {1,2,3}. All descriptors from set {1,2,3,4,5} are checked in the first level. If the descriptor is in the subset, it is subtracted, if the descriptor is not in the subset, it is added. In all next levels, the sets of descriptors for each node are created in exactly the same way as in the standard tree for all subsets, that have been explained in Figure 9.1. Explanation of the others is left as an exercise.

9.2.3 Creating Search Strategies

A number of *search strategies* can be specified for the tree search procedure along with the quality functions. Beginning with the initial node, the information needed to produce the solution tree can be divided into *global information,* that relates to the whole tree, and *local information*, that is concerned only with local subtrees. Local information in node N refers to subtree $TREE(N)$. The developer-specified search strategies are, therefore, also divided into a *global search* and a *local search*. The selection of the strategy by the user of the Universal Search Strategy from section 9.2.6 is based on a set of *strategy describing parameters.* By selecting certain values, the user can, for instance, affect the size of subsequent sets of bound variables or the types of codes in the encoding problem. We assume also that in the future we will create smart strategies that will allow to dynamically change the strategy parameters by the main program during the search process. Such strategies, that for instance search breadth-first and after finding a node with certain properties switch to depth-first search, have been used with successes in Artifficial Intelligence.

Let us distinguish the *complete search strategies* that guarantee finding all of the optimal solutions from the *incomplete strategies* that do not. Both the complete and the incomplete search strategies can be created for a complete tree search method. A tree searching strategy that is created for a complete tree search method and includes certain restricting conditions or cutting-off methods that can cause the loss of all optimal solutions is referred to as an *incomplete search strategy* for a complete search method. By removing such conditions a complete search strategy is restored, but it is less efficient.

This approach offers the following advantages:

- The *quasi-optimal solution* is quickly found and then, by backtracking, successive, better solutions are found until the *optimal solution* is produced. This procedure allows to investigate experimentally the trade-offs between the quality of the solution and speed of arriving at it.

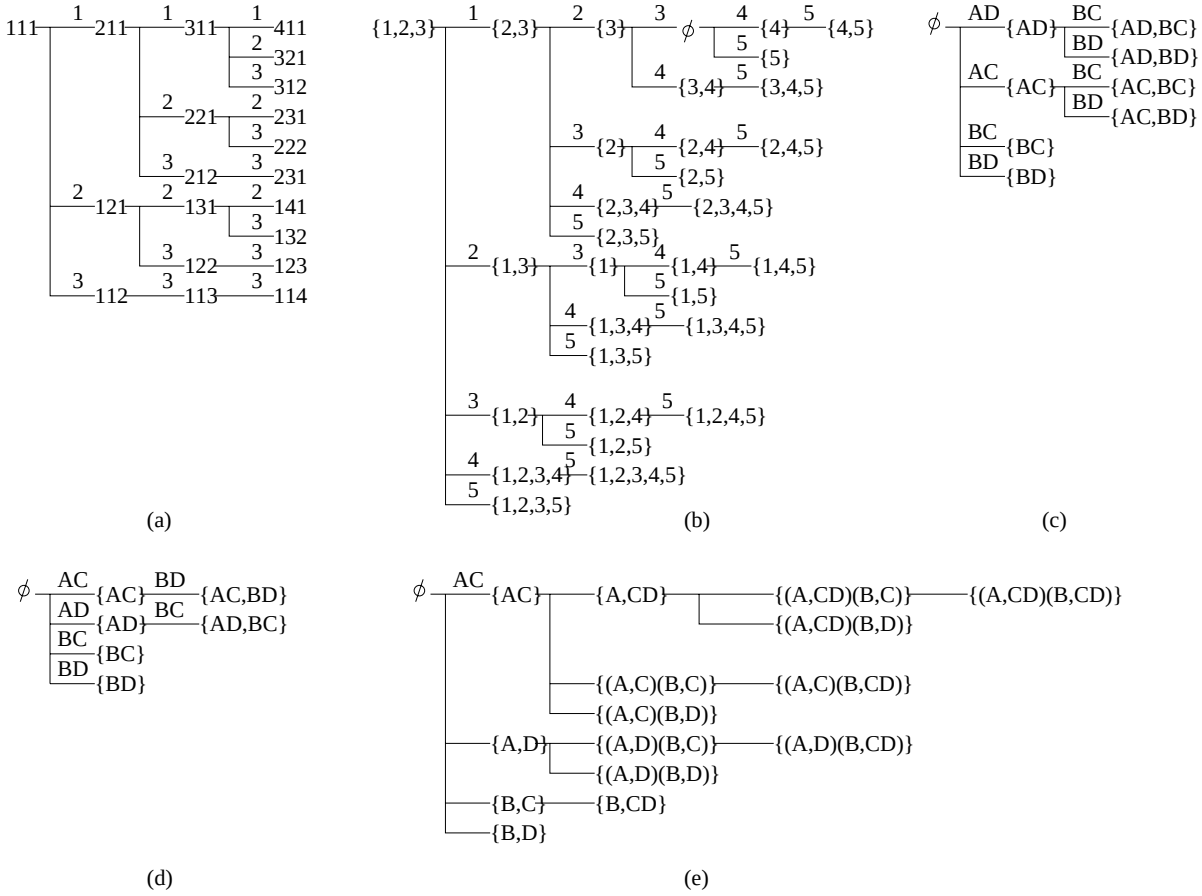


Figure 9.3: Further Examples of tree generators

- The search in the state-space can be limited by including as many *heuristics* as required. In general, a heuristic is any rule that directs the search. It will be more on the heuristics in the sequel.
- The application of various quality functions, cost functions, and constraints is possible.
- The problem can be described within several degrees of accuracy. The *direct description* is easy for the designer to formulate, even though it produces less efficient programs. It is created in the early prototype development stages, on the basis of the problem formulation only, and the heuristics are not yet taken into account. The only requirement is that the designer knows how to formulate the problem as a state-space problem using standard mathematical objects and relations. Only the standard node coordinates are used. The *detailed description of the tree search method,* on the other hand, provides the best program that is adequate for the specific problem but it requires a better understanding of the problem itself, knowledge about the program structure, and experimentation.
- By using macro-operators along with other properties, the main strategies require less memory than the comparable, well-known search strategies [265, 359, ?].

The search strategy is either selected from the *general strategies,* of which the following is a selection, or it is created by the developer's writing of the *sections codes*, and next the user assigning values to the *strategy describing parameters.*

- **Breadth-First.** With this strategy, each newly created node is appended to the end of the so called *open-list* which contains all the nodes remaining to be extended: *open nodes.* Each time the first node in the list is selected to be extended, it is removed from the list. After all the available operators for this node have been applied, the next node in the open-list to be extended is focused on.
- **Depth-First.** The most recently generated node is extended first by this strategy. When the specified depth limit SD_{max} has been reached or some other cut-off condition has been satisfied, the program backtracks to extend the deepest node from the open-list. This newly created node is then placed at the beginning of the open-list. The consequence is that the first node is also always the deepest.

- **Branch-and-Bound.** The temporary cost B is assigned which retains the lowest cost of the solution node already found. Whenever a new node NN is generated, its cost $CF(NN)$ is compared to the value of B. All the nodes whose cost exceed B will be cut off from the tree.
- **Ordering.** This strategy, as well as the next one, can be combined with the Branch-and-Bound strategy. A quality function $Q(r, N)$ is defined for this strategy to evaluate the cost of all the available descriptors of the node being extended. These descriptors are applied in the operators in an order according to their evaluated cost.
- **Random.** With this strategy, the operator or the open node can be selected randomly for expansion, according to the probability distribution specified.
- **Simulated annealing**. This strategy transforms nodes from open list using the respective algorithm.
- **Genetic algorithm**. This strategy uses open list as a genetic pool of parents' chromosomes.

All the strategy creating tools should be defined as C++ classes. The strategy describing subroutines and parameters are outlined below.

There are two types of conditions for each node of the tree: *by-pass condition* and *cut-off condition.* The cut-off condition is a predicate function defined on node N as an argument. If the cut-off condition is met in node N, the subtree $TREE(N)$ is prevented from being generated and backtracking results. The by-pass conditions do not cause backtracking and the tree will continue to extend from node N. The following cut-off conditions exist:

- **Bound Condition** . This condition is satisfied when it is found (possibly from information created in node N) that there exists node N_1 (perhaps not yet constructed) such that $CF(N_1) < CF(N)$ and $QS(N_1)$ is a solution.
- **Depth Limit Condition.** This condition is satisfied when $SD(N)$ is equal to the declared depth limit SD_{max}.
- **Dead Position Condition.** This condition is satisfied when no operators can be applied to N, i.e. $GS(N) = \emptyset$.
- **Restricting Conditions.** Each of these conditions is satisfied when it is proved that $QS(N)$ does not fulfill certain restrictions, i.e. no solution can be found in $TREE(N)$, for one or another reason.
- **Solution Conditions of the Cut-Off Type.** Any of these conditions is satisfied when the property of the problem is that if $QS(N)$ is a solution, then for each $M \in TREE(N)$, $CF(M) > CF(N)$ (or $CF(M) \geq CF(N)$). Therefore, node M may be not taken into account.
- **Branch Switch Conditions** and **Tree Switch Condition**. Satisfaction of Switch Condition causes modification of the actual search strategy to another strategy, resulting from the search context and previous conditional declaration of the user. This leads to the so-called *Switch Strategies* that dynamically change the search strategy during the process of search. For instance, the depth first strategy can be changed to breadth first if certain condition is met.
- **Other types of conditions.** They are formulated for some other type restrictions special to problems (selected by the user by setting flags in the main algorithm).

A value that interrupts the search when a solution node N is reached such that $CF(N) = CF_{min\ min}$ is denoted by $CF_{min\ min}$. This is a known minimum cost of the solution. This value can be arrived in many ways, usually it comes from a calculated guess, or is derived by some calculation or by mathematical deduction. It may also be a known optimal cost. In most cases the value is a "guessed value", that may be incorrect. Therefore, it will serve here only as one more control parameter.

When all the solution conditions are met in a certain node N, $QS(N)$ is a solution to the given problem. This is then added to the set of solutions and is eventually printed. The value of $CF(N)$ is retained. If one of the solutions is a "cut-off type solution", then the program backtracks. Otherwise, the branch is extended.

9.2.4 Relations on Operators and States

Determining some relations on operators (descriptors) is often very useful. Similarly, the developer may determine certain relations on states of the solution space, or on the nodes of the tree. Having such relations allows to cut-off nodes. It allows also to remove dispensable descriptors from the nodes. Specifically, in many problems it is good to check solution conditions immediately after creating a node, and next immediately reduce the set of descriptors that can be applied to this node.

The following relations between the operator descriptors (so called *relations on descriptors)* can be created by the program developer to limit the search process:

- *relation of domination,*
- *relation of global equivalence,*
- *relation of local equivalence.*

We will define *local* and *global* domination relations. Operator O_1 is locally dominated in node N by operator O_2 (or descriptor r_2 is locally dominated by r_1) when:

$$(O_1,\ O_2)\ \in\ DOML(N) \tag{9.18}$$

while relation $DOML$ satisfies the following conditions:

$$DOML\ is\ \text{transitive} \tag{9.19}$$

and

$$(O_1,\ O_2)\ \in\ DOML\ \Rightarrow\ f(O_1(N))\ \leq\ f(O_2(N)) \tag{9.20}$$

We will apply the notation:

$$(O_1,\ O_2)\ \in\ DOML\ \overset{define}{\Longleftrightarrow}\ O_1\ \underset{L}{\succ}\ O_2 \tag{9.21}$$

We will define operator O_2 as *locally subordinated* in node N with respect to operator O_1 (where $d_1,\ d_2\ \in\ GS(N)$), if

$$O_1\ \underset{L}{\succ}\ O_2\quad\wedge\quad O_2\ \underset{L}{\not\succ}\ O_1 \tag{9.22}$$

This will be denoted by

$$O_1\ \underset{L}{\succ}\ O_2 \tag{9.23}$$

If $O_1\ \underset{L}{\succ}\ O_2$ in node N, then tree $TREE(O_2(N))$ can be cut-off without sacrificing optimal solutions, since

$$f(O_1(N))\ <\ f(O_2(N)) \tag{9.24}$$

It is easy to check that relation $\underset{L}{\approx}$, defined as

$$O_1\ \underset{L}{\approx}\ O_2\quad\overset{define}{\Longleftrightarrow}\quad O_1\ \underset{L}{\succ}\ O_2\ \wedge\ O_2\ \underset{L}{\succ}\ O_1 \tag{9.25}$$

is an *equivalence relation*, which we will call the *Local Relation of Equivalence of Descriptors* in node N. Relation $\underset{L}{\approx}$ partitions set $GS(N)$ into classes of abstraction $[d_i]$. It is obvious from these definitions, that when one wants to obtain only a single optimal solution being a successor of N, then from each class of abstraction $[d_i]$ only one element should be selected. All remaining elements should be removed from $GS(N)$.

The relation of *global domination* gives better advantages than the local domination, in cases that such a relation of global domination can be defined. Operator O_2 is *globally dominated* in tree $TREE(N)$ by operator O_1 when

$$(O_1,\ O_2)\ \in\ DOMG(N)\ \subset\ O(TREE(N))\ \times\ O(TREE(N)) \tag{9.26}$$

By $O(TREE(N))$ we denote the set of operators to be applied in tree $TREE(N)$. Relation $DOMG$ satisfies the following conditions:

$$DOMG \text{ is transitive} \tag{9.27}$$

and

$$(O_1,\ O_2)\ \in\ DOMG(N)\ \ \Rightarrow\ \ (\forall M_1\ \in\ NO(TREE(N)))$$
$$[d_1 \in GS(M_1) \wedge d_2 \in GS(M_1) \wedge f(O_1(M_1)) \in f(O_2(M_1)) \vee d_2 \notin GS(M_1)] \tag{9.28}$$

Similar to local relations, one can define relation $\overset{\succ}{}_G$ of *global subordination in tree* $TREE(N)$, and relation $\overset{\approx}{}_G$ of *global equivalence* in tree $TREE(N)$.

Relation $\overset{\approx}{}_G$ partitions every set $GS(M_1)$ for each $M_1 \in NO(TREE(N))$ into classes of abstraction. If we have no intention to find all optimal solutions, then from each class of abstraction we take just one element, and the remaining operators are removed from $GS(M_1)$.

The following theorem can be proven.

Theorem. Let us denote by $[d_i]$ the global equivalence class of operator O_i in node N. If for each branch $N, N_1, N_2, ...N_k$ of tree $TREE(N)$ it holds $GS(N) \supseteq GS(N_2) \supseteq GS(N_k)$ then the descriptors from set $[d_i] \setminus \{d_i\}$ can be immediately removed from all sets GS in all nodes in $TREE(N)$.

When we want to use the relation of global equivalency in certain node N, and the property from this theorem does not hold, then it is necessary to calculate the descriptors, which should be not applied in node N (sometimes it can be easily done from an analogous set for the node being the parent of this node).

Node M is *dominated* by node N if $f(N)\ \leq\ f(M)$

$$(N,\ M)\ \in\ DOMS\ \ \Longleftrightarrow\ \ f(N)\ \leq\ f(M) \tag{9.29}$$

Similarly as before, we can introduce relations $\overset{\succ}{}_S$, $\overset{\succ}{}_S$, and $\overset{\approx}{}_S$.

If ST_1 and ST_2 are two strategies, which differ only in their domination relations D_1 and D_2 (these can be relations of domination of any of the presented types) and if $D_1 \supset D_2$ then ${k_i}^1\ \leq\ {k_i}^2$ for each of the introduced coefficients k_i.

Observe, that by incorporating the test for the relation of domination (or equivalence) to an arbitrary strategy that generates all optimal solutions, there exists the possibility of sacrificing only some optimal solutions (or all the optimal solutions but one). This decreases the number of generated nodes, which for many strategies is good both with respect to the reduced time, and reduced memory. On the other hand, if evaluating relations is very complex, the time of getting the solution can increase. The stronger is the domination relation, the more complicated is its evaluation, or the larger is its domain. Therefore, the time for testing domination would grow. In turn, the more gain from the decreased number of generated nodes. Often it is convenient to investigate relation of domination only in nodes of the same depth, or on operators of some group. Theoretical analysis is often difficult and experimenting is necessary.

Finally, observe that domination relations are not based on function f, because the values of f are not known a'priori, while creating the levels of the tree. The domination relations are also not based on costs, but on some additional problem-dependent information of the program, about the nodes of the tree. These relations come from certain specific problem-related information. In most cases, the implication symbol in equation 9.20 cannot be replaced by the equivalence symbol, since this would lead to optimal strategies with no search at all, and each branch would lead to optimal solutions.

9.2.5 Component Search Procedures

The **Main Universal Search** subroutine of a search program is in charge of the global search. It takes care of the selection of strategies, the arrangement of the open-list and the other lists as well as the decision making facilities related to the cut-off branch, and the configuration of the memory structures to store the tree. The lines of code that realize the strategies of breadth-first, depth-first, or branch-and-bound are built into the main search routine. Subroutines $RANDOM1$ and $RANDOM2$ are selectively linked for the random selection of the operator or the open node, respectively. The role of the subroutines linked to the Universal Search subroutine is as follows:

- $GENER$ is responsible for the local search that extends each node. $GENER$ cuts off the nodes which will not lead to the solution node when the description for the new node is created.
- GEN carries out the task of creating nodes.

Other subroutines, offered to create local search strategies, are the following:

- $MUSTAND$ and $MUSTOR$ are subroutines that serve to find two types of the so-called *indispensable operators.* (The indispensable operators are the operators that must be applied.) All operators found are indispensable in the $MUSTAND$ subroutine, and only one of operators is indispensable in case of the $MUSTOR$ subroutine. The set of indispensable operators is next substituted as the new value of coordinate $GS(N)$.
- subroutine $MUSTNT$ deletes *subordinate operators.* Subordinate operators are those that would lead to solutions of higher costs, or to no solutions at all. The set $MUSTNT(N)$ is subtracted from set $GS(N)$.

Domination and equivalence conditions for the tree nodes can also be declared as follows:

- $EQUIV$ cancels those nodes that are included in other nodes.
- $FILTER$ checks whether the newly created node meets the conditions.
- $SOLNOD$ checks the solution condition.
- $REAPNT$ is used to avoid the repeated applications of operators when the sequence of operator applications does not influence the solution.

These local strategies, as well as the global strategies listed above, can be selected by reading the parameter values as input data. $ORDER$ sorts the descriptors, QF calculates the quality function for the descriptors, and CF calculates the cost of the nodes.

9.2.6 Universal Search Strategy

In this section we will present the universal search strategy. First we will explain the meaning of all variables and parameters. Next the pseudo-code of the main strategy subroutine will be given, followed by the pseudo-code of its GENER subroutine.

Meaning of Variables and Parameters

CF_{min} - cost of the solution that is actually considered to be the minimal one. After a full search, this is the cost of the exact minimum solution.

SOL - set of solutions actually considered to be minimal. If parameter $METHOD = 1$, then this set has always one element. When a full search has been terminated, this set includes solutions of the exact minimal cost.

$OPERT$ - list of descriptors, which should be applied to the actual state of the tree.

$OPEN$ - list of open and semi-open nodes.

N - actual state of the space.

NN - next state of the space (this state is actually being constructed from node N).

$OUTPUT$ - a parameter that specifies the type of the currently created node;

when $OUTPUT = 0$, the created node NN is a branching node;

when $OUTPUT = 1$, the created node NN is an end of a branch;

when $OUTPUT = 2$, a *quasioptimal solution* was found, whereby by a quasioptimal solution we understand any solution that has the value of the cost function not greater than the user-declared parameter $CF_{min\ min}$.

$CF_{min\ min}$ - a parameter assumed by the user, determined heuristically or methodically, the value that satisfies him.

QF_{min} - the actually minimal value of the quality function.

OPT - a parameter. When $OPT = 1$, then any solution is sought, otherwise the minimal solution.

$PP9$ - a parameter. When $PP9 = 1$, then the subroutine *"Actions on the Selected Node"* is called.

EL - actual descriptor from which the process of macro-generation starts (this is the first element of list $OPERT$).

$ESCRIPTOR$ - actual descriptor during the macrogeneration process.

$MUST$ - list of descriptors of operators, which must be applied as part of the macrooperator.

$PG5$ - a parameter. If $PG5 = 1$, then it should be investigated, immediately aftear the creation of node NN, if there exists a possibility of cutting-off node NN.

$PG6$ - a parameter. If $PG6 = 0$, then it should be investigated if node NN can be cut-off with respect to the monotonically increasing cost function CF, and in respect to satisfaction of $CF_{min} = CF(NN)$.

$PG6D$ - a parameter. If $PG6D = 1$, then value CF_{min} should be calculated with respect to a subroutine of a user, otherwise CF_{min} is calculated in a standard way as $CF(NN)$.

$PG6E$ - a parameter. If $PG6E = 1$, then the learning subroutine is called.

$PG6F$ - a parameter. If $PG6F = 1$, then after finding a solution the actions declared by the user are executed.

$PG7$ - parameter; if $PG7 = 1$ then descriptors defined by other parameters are removed from $GS(NN)$.

The Main Search Strategy

1. Set the parameter variables to the values that will determine the search strategy.
2. $CF_{min} := \infty$, $SOL := \emptyset$, $OPERT := \emptyset$, $OPEN := \emptyset$.
3. Call the macrogeneration subroutine $GENER$ for the user-declared initial state N_0.
4. If the value of variable $OUTPUT$ (this value is set by subroutine $GENER$) is 1 or 2

   then, according to the declared parameters, return to the calling program for the problem, or select a strategy corresponding to the declared data.
5. State N_0 has been (possibly) transformed by subroutine $GENER$.

   Store the new state N_0' in the tree. $OPEN := \{N_0'\}$.
6. If $OPEN = \emptyset$ then either return to the calling program, or change the search strategy, according to the parameters and the strategy change parameters for trees (*Tree Switch*). (see section 9.2.8).
7. If the threshold values for the tree have been exceeded (size, time, etc) then return to the calling program, or change strategy, as in step 6.

   If the *Stopping Moment Learning Program* decides termination of the search, then this search process is terminated. Return to the calling program, that will decide what to do next (see section 10).
8. $N :=$ selected node from list $OPEN$. This step is executed on the basis of Strategy Selecting Parameters, including minimal values QF or CF.

   If parameters specify A^* Strategy of Nillsson and $QF(N) \geq QF_{min}$,

   then return to the calling program (since all minimal solutions have been already found).
9. If parameter $PP9 = 1$,

   then

   call subroutine "Actions on the selected node" (this subroutine can, for instance, declare such actions as: (1) cutting-off a node upon satisfying some condition, (2) sorting $GS(N)$, (3) assigning $GS(N) := \emptyset$, (4) deleting redundant or dominated operators).

   $OPERT := GS(N)$. item Remove from list $OPEN$ all closed nodes.
10. If $OPERT = \emptyset$ then go to 6.
11. $EL := OPERT[0]$, remove EL from list $OPERT$.

    ($OPERT[0]$ selects the first element of list $OPERT$)
12. Call subroutine $GENER$.
13. If a *Branch Switch Strategy* has been declared and a respective switch condition is satisfied

    then execute the Branch Switch type modification of the search strategy.

14. If $OUTPUT = 0$, then store the node NN (created earlier by subroutine $GENER$) in the tree (if a tree data structure is used in addition to list $OPEN$). Insert this node in certain position in list $OPEN$. This position depends on the selected strategy.

    If $OUTPUT = 2$, then (if parameter $OPT = 2$ then return to the calling program, else go to 11).

15. Go to 11.

Subroutine GENER

1. If $GENER$ is executed in step 13 of the main search strategy then $MUST := EL$ (value of EL has been previously set in the main search routine).
2. If $MUST = \emptyset$, then set $OUTPUT := 0$, return.
3. $DESCRIPTOR := MUST[0]$.
4. Call subroutine $OPERATOR$ written by User. We denote this by $O(N,\ DESCRIPTOR)$. This call generates the new state NN, for the $DESCRIPTOR$ selected in step 3.

   $GS(N) := GS(N) \setminus DESCRIPTOR$ (i.e. DESCRIPTOR is removed from GS(N)).
5. If parameter $PG5 = 1$

   and

   (node NN satisfies on of the Branch Cut-Off Conditions **or** NN is dominated by another node),

   then cut-off node NN. $OUTPUT := 1$. Return.

   (the above condition means that node NN is equal to another node, or node NN is dominated by another node based on one of the relations: Node Domination, Node Equivalence, Node Subordination)

   If $CF(NN) > CF_{min}$ (while looking for all minimal solutions)

   or

   If $CF(NN) \geq CF_{min}$ (while looking for a single minimal solution),

   then

   cut-off node NN. $OUTPUT := 1$, return.
6. If parameter $PG6 = 0$ then

   If $CF_{min} = CF(NN)$ and the parameter specifies that CF is monotonically increasing and node NN does not satisfy all the user-declared Solution Conditions,

   then cut-off node NN, $OUTPUT := 1$, return.

   If node NN satisfies all the user-declared Solution Conditions, then

   A If $CF(NN) \leq CF_{min}$ and the A^* Strategy of Nillsson is realized, then
   store $QF_{min} := QF(NN)$.

   B If $CF(NN) = CF_{min}$, then

   i. if all optimal solutions are sought,
   then append $QS(NN)$ to the list of solutions SOL else do nothing.)

   C If $CF(NN) < CF_{min}$ then set $SOL := \{\text{QS(NN)}\}$.

   D If parameter $PG6D = 1$, then calculate CF_{min} using the User Subroutine Calculating CF_{min}, else $CF_{min} := CF(NN)$.

   E If parameter $PG6E = 1$, then call the subroutine "Parametric Learning the Quality Function for Operators".

   F If parameter $PG6F = 1$, then call the subroutine "Actions after Finding a Solution". This is a subroutine used to specify the actions to be executed after the solution is found. These actions can be: printout, display, storage, etc.)

   G If $CF(NN) = CF_{min\ min}$, then $OUTPUT := 2$, return.

   H If $CF(NN) \neq CF_{min\ min}$, then $OUTPUT := 1$, return.

7. If $PG7 = 1$, then remove the indispensable descriptors from $GS(NN)$. Depending on the values of parameters, the following types of descriptors are being removed: (2) Inconsistent Descriptors, (3) Descriptors that result from: (3a) Local Subordination Relation, (3b) Local Domination Relation, (3c) Local Equivalence Relation, (3d) Local Equivalence Relation, (3e) Global Subordination Relation, (3f) Global Domination Relation, (3g) Global Equivalence Relation.

   Use subroutine $MUSTNT$.

   If a Condition of Node Expansion Termination is satisfied then set $GS(NN) := \emptyset$.

   If the set of Indispensable Operators of $MUSTOR$ type is declared

   and

   respective Condition of operators of $MUSTOR$ type is satisfied,

   then set $GS(NN) := MUSTOR(GS(NN))$.

8. $N := NN$.

9. If $MUST \neq \emptyset$, then go to 2.

   else $MUST :=$ set of Indispensable Descriptors of $MUSTAND$ type in $GS(N)$. Go to 2.

The first call of subroutine $GENER$ is intended to check if the indispensable operators of type $MUSTAND$ exist in the initial state given by the user. These operators are applied to the successively created states, until a solution is found, or a node is found, in which no longer exist any indispensable descriptors. When subroutine $GENER$ is returned from, the state N_0 may have been transformed. The condition to find the minimal solution is to terminate with empty list $OPEN$. In steps 8 and 9, with respect to the strategy determining parameters, the node for expansion is selected, together with the operators that will be applied to this node. This node can be the open or semi-open type. *Open* means all possible operators have been applied to it. *Semi-open*, means some operators (descriptors) were applied but other descriptors remain, ready to be applied in a future. Selected descriptors are successively applied to the node, until list $OPERT$ is cleared.

The value of parameter OPT is determined by the user. If $OPT = 1$, then the subroutine will return to the calling program after finding the first quasi-optimal solution.

Subroutine $GENER$ is used to find and apply macro-operators. Descriptor EL, selected in the main search program, is put to list $MUST$ of indispensable descriptors (except of the call in step 3). Such approach has been chosen in order to check if some indispensable descriptors exist in the initial state. It is known for all subsequent nodes that there are no indispensable descriptors, since if there were an indispensable descriptor in a node created by $GENER$, it would be immediately applied. Therefore, the result of subroutine $GENER$ is always a single child, that has no indispensable operators.

In a general case, pure branch-and-bound strategy (discussed below) will terminate in steps 4 and 6 of the main search strategy. The A^* strategy of Nilsson will terminate in step 8.

Of course, in lists $OPEN$, $OPERT$ and other lists, not objects are stored, but pointers to them.

9.2.7 Pure Search Strategies

In this section we present the so-called pure search strategies. They will not require strategy-switching. Many of these strategies are known from the literature. Pure strategies are the following.

1. Strategy ST_{QF} is defined as follows:

$$QF(SEL1_{QF}(OPEN)) \;=\; min_{N_i \in OPEN} QF(N_i), \quad SEL2(x) \;=\; x \tag{9.30}$$

   $SEL1$ is the node selection strategy and $SEL2$ is the descriptor selection strategy.

   In this strategy, all children of node N are generated at once. This corresponds to the "Ordered Search" strategy, as described in [359, 232].

   If, in addition to the above formula 9.30 function QF satisfies conditions 9.9, 9.10, 9.11, 9.12, 9.13, then it corresponds to the well-known A^* strategy of Nilsson.

2. Strategy ST_{CF} (strategy of equal costs), in which:

$$CF(SEL1_{CF}(OPEN)) \;=\; min_{N_i \in OPEN} CF(N_i), \quad SEL2(x) \;=\; x \tag{9.31}$$

   This is a special case of the strategy from point 1.

3. Depth-first Strategy

   $SEL1_d(OPEN)$ = the node that was recently opened, $SEL2(x) = x$

4. Breadth-first Strategy

   $SEL1_b(OPEN)$ = the first of the opened nodes, $SEL2(x) = x$

5. Strategy $ST_{d,s,k}$ (depth, with sorting and selection of k best operators)

   $SEL1_{d,s,k}(NON\text{-}CLOSED)$ = the node that was recently opened,

   $SEL2_{d,s,k}(GS(SEL1(NON\text{-}CLOSED)))$ = set that is created by selecting the first k elements in the set $GS(SEL1(NON\text{-}CLOSED))$ sorted in nondecreasing order according to function $q^N{}_i$. A particular case of this strategy is $ST_{d,s,1}$, called the Strategy of Best Operators (Best Search Strategy).

6. Strategy $ST_{d,s,s,k}$ (i.e. the depth-search strategy, with the selection of a node, sorting, and the selection of the k best operators).

   $SEL1_{d,s,s,k}(NON\text{-}CLOSED)$ = a node of minimum value of function QF among all nodes that are created as the extension of the recently expanded node (not necessarily of the recently opened node).

   $SEL2_{d,s,s,k}$ = similarly to $SEL2_{d,s,k}$.

   Similarly, one can define "k-children" strategies $ST_{QF,k}$, $ST_{CF,k}$, $ST_{d,k}$.

7. Strategy ST_{RS} of Random Search.

   $SEL1_{RS}(NON\text{-}CLOSED)$ = randomly selected node from NON-CLOSED.

   $SEL2_{RS}(GS(SEL1_{RS}(NON\text{-}CLOSED)))$ = randomly selected subset of descriptors.

8. Strategy $ST_{RS,d}$ of Random Search Depth.

   $SEL1_{RS,d}(NON\text{-}CLOSED)$ = recently opened node from $NON\text{-}CLOSED$.

   $SEL2_{RS}(GS(SEL1_{RS,d}(OPEN)))$ = randomly selected descriptor.

Similarly, one can specify many other strategies by combining functions $SEL1$ and SEL given above.
Let us now introduce few measures of quality of strategies.

k_1 = CARD(B_a), where B_a is the set of all closed and semi-open nodes that were created until all minimal solutions have been found.

k_2 = CARD(B_s), where B_s is the set of all closed and semi-open nodes that were created until one minimal solution has been found.

k_3 = CARD(V_a), where $V_a \supseteq B_a$ is the set of all closed, semi-open, and open nodes that were created until all minimal solutions have been found.

k_4 = CARD(V_s), where $V_s \supseteq B_a$ is the set of all closed, semi-open, and open nodes that were created until one minimal solution has been found.

k_5 = CARD(T_a), where $T_a \supseteq B_a$ is the set of nodes that were created until proving the minimality of solutions, it means the total number of nodes that have been created by a strategy that searches all the minimal solutions.

k_6 = CARD(T_s), similarly to k_5, but for a strategy that searches a single solution.

k_7 = max SD(N_i) - the length of the maximal path (branch) in the tree.

The advantage of the ordered search strategy is the relatively small total number of generated nodes (coefficients k_5 and k_6). The following theorem is true, similar to the theorem from [359].

Theorem 9.1 *If QF satisfies equations 9.9, 9.10, 9.11, 9.12, 9.13 and the ordered search strategy has been chosen (i.e. the strategy A* of Nilsson is being realized) and when some solution of cost QF' has been found, such that all nodes of costs smaller than QF' have been closed, then this solution is the exact minimal solution.*

It is important to find conditions, for which this algorithm finds the optimal solution, generating relatively few nodes. The theorem below points to the fundamental role of function $\hat{h}$. The way in which function $\hat{h}$ is calculated, can substantially influence the quality of solutions in approximate version, or efficiency of the algorithm in exact version.

Let ST_1 and ST_2 be two A^* Nilsson strategies, and $\hat{h}_1$ and $\hat{h}_2$ their heuristic functions. We will define that strategy ST_2 is *not worse specified* than strategy ST_2 when for all nodes N it holds:

$$h(N) \ \geq \ \hat{h}_1(N) \ \geq \ \hat{h}_2(N) \ \geq \ 0 \tag{9.32}$$

which means, both functions evaluate h from the bottom, but function $\hat{h}_1$ does it more precisely than $\hat{h}_2$.

Theorem 9.2 *If ST_1 and ST_2 are A^* Nilsson strategies, and ST_1 is not worse specified than ST_2, then, for each solution space, the set of nodes closed by ST_1 (before the minimal solution is found) is equal to the set of closed nodes of ST_2, or is included in it.*

This theorem says, in other words, that if we limit ourselves to A^* Nilsson strategies only, then there exists one strategy, not worse than all remaining strategies, since it closes not more nodes of the tree than any other strategy. This is the strategy that most precisely evaluates the function h, preserving of the equations 9.10, 9.11, 9.13.

For many classes of problems the ordered search strategy is very inefficient because it generates its first solution only when very many nodes have already been created. Then it proves its optimality relatively quickly. In cases, when the user wants to find quickly some good solution, but the **exactness** of the solution is only of secondary importance, it is better to use one of the variants of the branch-and-bound strategies that search in depth.

Many properties can be proven for the branch-and-bound strategy presented above. We assume that

$$for\ each\ N, NN, QF(N) \neq QF(NN)\ and\ QF(NN) > QF(N)\ for\ NN \in SUCCESSORS(N) \tag{9.33}$$

1) The branch-and-bound strategy is convergent, independent on function QF. Also the specific strategies included in it (such as "depth-first", "ordered search", etc,) are therefore convergent as well, if the user has not declared some additional cut-off conditions (that may cause the loss of the optimal solution). Some of these strategies do not require calculating function QF satisfying certain conditions. This property is an advantage of the given above universal search strategy, when compared with the A^* Nilsson Strategy.

2) If the user is able to define the quality function QF^* such that

$$(\forall N_i, N_j)[QF^*(N_i) < QF^*(N_j) \ \Rightarrow \ f(N_1) \ \leq \ f(N_j)], \tag{9.34}$$

then the strategy ST_{QF} is optimal in the sense of the number of opened nodes. Only the nodes that are on the paths leading to solutions are extended (other nodes are also opened).

3) If additionally the user succeeds to find a quality function for operators q^{*N} that is **consistent** with f,

$$(\forall N, O_1, O_2)[q^{*N}(O_1) > q^{*N}(O_2) \ \Rightarrow \ f(O_1(N)) \ \leq \ f(O_2(N))], \tag{9.35}$$

then the strategy is optimal in the sense of the number of generated nodes. Only those nodes are expanded, that lay on those paths that lead to minimal solutions. In addition, no other nodes are opened (this concerns the 1-child strategies).

4) It is possible to introduce the relation of partial ordering $\ll$ on the set of all possible strategies ST_{QF}. It can be proven that the strategies that are *adjacent* in the sense of this order have also similar behavior:

$$if\ ST_{QF_1} \ \ll \ ST_{QF_2}\ then\ {k_i}^1 \ \leq \ {k_i}^2,\ for\ i \ = \ 1, 2, ..., 6. \tag{9.36}$$

5) The best strategy with respect to relation $\ll$ (the minimal element of the lattice), is the strategy ST_{QF^*}. The "adjacent" strategies are defined. Next it can be proven, that if QF_0, QF_1, ..., QF_q is a sequence of such adjacent functions, then the corresponding strategies, ST_{QF_0}, ST_{QF_1}, ..., ST_{QF_q}, are adjacent in the lattice of strategies. Therefore, in the class of the ordered search strategies function ST is in a sense a continuous function of function QF: small changes of QF cause small changes of ST_{QF}. If $QF \approx QF^*$ then behavior of ST_{QF} is close to optimal. If the user is able to make choices among **all** functions QF, then by the way of successive experimental modifications he can approach the ST_{QF^*} strategy.

6) Since strategies "depth-first" are very sparsely located in the lattice (they have high distances from one another), small changes of QF can cause a "jump" from ST_{QF^*} to a lattice element that is located far from it. Similarly, small modification of QF in the direction of QF^* do not necessarily lead to the improvement of the algorithm's behavior.

7) It can be shown, that in the sense of some of the measures introduced above, the proposed algorithm is better than the branch-and-bound algorithms investigated by Ibaraki [232].

Usually, the user should always try to find function QF close to QF^*. With better functions QF, the program will find good solutions sooner, where by good solution we understand those with small values of CF_{min} (the decrease of coefficients k_1 - k_4). Therefore, the cut-off of subsequent branches will be done with a smaller value, which will in turn decrease the values of k_5 and k_6. When the depth-first strategy is selected, the changes in behavior can occur in jumps. In addition, with respect to 3), the user has to select function q. With respect to equations 9.19- 9.29, respectively, he has to define relations on descriptors and states.

When constructing the strategies, the user has also to keep in mind the following.

1. Generally, for those branch-and-bound strategies that search in depth, it is necessary to define that every branch of the tree terminates with a solution found. In addition the branch is determined with certain constructive conditions of cutting-off (for instance, the cutting-off occurs when certain depth of the tree was reached, or when there are no more operators to apply). The lack of these conditions may lead to the danger of infinite depth-search, or a very long depth-search. For instance, in case of strategy $ST_{d,1}$. This condition is not necessary for A^* Nilsson strategy, which is a special case of the strategy.

2. With respect to parameters k_1 - k_4, the strategies that combine properties of strategies $ST_{d,s,s,k}$, A^* Nilsson Strategy, and $ST_{d,s,k}$, have the best performance.

3. With respect to parameter k_7 the 1-child strategies are the best, and the $ST_{d,s,1}$ strategy in particular.

4. When the user looks for a solution with the minimal depth in the tree, the breadth-search strategy creates theoretically the exact solution as the first solution generated, which is sometimes good. However, the tree can grow often so rapidly, that the strategy cannot be used. Yet in another problems, it is good to use the disk memory. The strategy is useful when the problem is small, or when one can define powerful relations on descriptors or relations on states of the search space.

5. When the depth is limited or when good upper bounds can be found, the depth-first strategies allow to find the solutions faster. Depth-first strategies are good when there are many solutions. They are memory efficient. These strategies are not recommended when the cost function does not increase monotonically along the branches, allowing thus to use the cutting-off.

6. Strategies ST_{QF} and $ST_{d,s,k}$ often require the shortest times of calculations. The second strategy requires a smaller memory.

7. By constructing strategies that use quality functions one has to take into account that the evaluation of a more complex function allows to decrease the search. It takes, however, more time. Therefore, the trade-offs must be experimentally compared.

8. It is possible to combine all presented strategies, and also to add new problem-specific properties to the strategies. The user can, for instance, create from the depth-first strategy and breadth-first strategy a new strategy that will modify itself while searching the tree, and according to the intermediate solutions found. Another useful trick is to cut-off with some heuristic values, for instance some medium value of CF_{min} and $CF_{min\ min}$.

9. An advantage of random strategies is a dramatic limitation of required space and time. These strategies are good, when used to generate many good starting points for other strategies, and these other strategies find next the locally optimum solutions.

9.2.8 Switch Strategies

There are two types of Switch Strategies:

- switch strategies for branches,
- switch strategies for trees.

Below, we will present them both.

A Switch Strategy is defined by using the conditional expression:

$$[\ sc_1 \longmapsto (MM_1,\ TREE_1), ..., sc_n \longmapsto (MM_n,\ TREE_n)\ ] \tag{9.37}$$

where

1. $sc_1, ..., sc_n$ are *switch conditions*,
2. $MM_i = (M_i, ST^i)$ are methods to solve problems by Universal Strategy,
3. M_i are tree methods,
4. ST^i are pure strategies,
5. $TREE_i$ are initial trees of methods MM_i (trees after strategy switchings).

The meaning of formula 9.37 is the following. If condition sc_1 is satisfied, then use method MM_1 with initial tree $TREE_1$. Else, if condition sc_2 is satisfied, then use method MM_2 with initial tree $TREE_2$. And so on, until sc_1 is encountered. This is like the COND instruction of Lisp language.

In practice, M_i, ST^i and $TREE_i$ are defined by certain *changes* to the actual data. These can be some symbolic transformations, or numeric transformation. They can be also the selections of new data structures. Therefore one has to declare the initial data: M_0, ST^0 and $TREE_0$.

- In Switch Strategy for a Branch, the conditions $sc_i, i = 1\ , ...,\ n$ are verified when a new node is created. These conditions can be also verified in one of the following cases: (1) a new node being a solution is created, (2) a node is found, being a solution better than the previous solution. The type of the node is specified by the parameters.
- In the Tree Switch Strategy, the conditions are checked after a full tree search of some type has been completed.

In both types of strategies, the conditions of switching strategies can be defined on

- nodes NN,
- branches leading from N_0 to NN,
- expanded trees.

There can exist various *Mixed Strategies* STM, defined as follows

$$STM = (SST_T, SST_B) \tag{9.38}$$

where

$\mathbf{SST_T}$ - is a Tree Switch Strategy,

$\mathbf{SST_B}$ - is a Switch Strategy for a Branch.

For both the Switching Strategies for Tree, and Switching Strategies for Branches, there exist eight possible methods of selecting changes. These methods are specified by one of the subsets of the set $< M_i,\ ST^i,\ TREE_i >$ In a special case, by selecting an empty set, changes of M_i, ST_i or $TREE_i$ are not specified. This corresponds to a pure strategy ST_0 (which was declared as the first one). Pure strategies are therefore a special case of the switch strategies.

Similarly, complex methods, defined as $CM = (M_1, ..., M_r, STM)$ are generalizations of methods MM_i.

Changes of $TREE_i$, M_i, and ST_i will be now presented.

1. The following changes of $TREE_i$ has been considered.

- change of coordinates of nodes (locally, or in a branch, or in the whole tree),
- adding or removing some coordinates (locally, or in a branch, or in the whole tree),
- cut-off the tree.

2. Changes of M_i by use of a switch strategy can be executed by specifying new components of the solution space.

   The strategy for Graph Coloring from section 9.5 is an example of a switching strategy that changes both $TREE$ and M.

3. Strategy is modified by determining the Change of Strategy Parameters. For instance, the modification of the strategy consists in: (1) a permutation of list $OPEN$, (2) a selection of some its subset, (3) some modification to list $OPERT$. Since the entire information about the solution tree is stored in list $OPEN$, the new strategy can start working immediately after the Branch Switch. The Main Universal Search Subroutine is constructed in such a way, that even by applying the switch search strategy it is still possible to obtain the exact solution.

Examples of Switch Strategies

- **The Far-Jumps Strategy.** This strategy finds solutions with high mutual distances in the solution space. At first, the Breadth-First Strategy with macrooperators and dominance relations is used to develop a partial tree. Together with each node N of the tree also its level in the tree, $SD(N)$, is stored. A node from $OPEN$ that has the smallest level is selected. Next the "depth-first" strategy is used until the first solution is found. The program evaluates, using some additional method, whether this is a minimum solution, or a satisfactory solution. When program evaluates that this was not the minimum solution, the "strategy switch" is executed. The strategy switch is executed as follows. (1) the node with the lowest level in the actual list $OPEN$ is selected; (2) this node is added at the beginning of list $OPEN$. Starting from this node, the tree is expanded again using the depth-first strategy, until the next solution is found, etc. With each solution, the order of nodes in $OPEN$ can be modified.

- **The Distance Strategy.** An advantage of this switch strategy is that the successively generated solutions are placed far away one from another. This gives the possibility of "sampling" in many parts of the space, which can lead to quicker finding of good cut-off values (this happens thanks to the jumping-out of the local minima of the quality function). It may be useful, that the "sampling" property is the opposite to the "depth-first" or other pure search strategies.

- **The Strategy of Best Descriptors**. The principle of this strategy is that it stores, for some pure strategy (for instance the depth-first, or the ordered-search), all the descriptors that proved to be the most useful in finding the previous solution. Sometimes, only some of these descriptors are stored. For instance, the dominating descriptors, or the descriptors with the highest values of cost or quality functions are stored. After switch, these descriptors are placed at the beginning of list $OPERT$, and are therefore used as the first ones in the next tree expansion. The switch strategies of this type can be applied to find quickly good cut-off values in branch-and-bound strategies.

- **Strategy of Sequence of Trees.** This is an example of a strategy that switches trees. It expands some full tree, or a tree limited by some global parameters (time, number of nodes). Next, using some additional principles, it selects few nodes, SEL_NODES, of the expanded tree (for instance, the nodes with the minimum value of the cost function). Finally, the strategy expands new trees, each starting from one that start from SEL_NODES nodes. It usually uses a different set of components of the space, and/or pure strategy in these new trees. In particular, one of the strategies selects a new set of descriptors. Another strategy of this type, calculates the value of CF_{min} as some function of CF_{min} and other parameters, including the probabilistic evaluations of $CF_{min\ ,min}$ during the moment of switching. This strategy is not complete, but it can substantially limit the search by backtracking from smaller depth values.

9.2.9 Problems

1. What are the main concepts in tree search?
2. What is the difference between state space and search tree?
3. For problems from the previous section, specify each tree-search process in terms of states, operators, coordinates and strategies explained in this section.

4. Do any of the problems from the previous section require modifications to the main Universal Search Strategy? What problems?

5. Create Search Methods for the Maximum Clique Problem:

   a) searching starting from the smallest cliques,
   b) searching starting from the largest cliques,
   c) searching starting from cliques of certain size,
   d) searching starting from certain clique, and next from the cliques that differ only by few nodes from it.

6. Repeat Problem 5 above for the Problem of Finding the Bound Set of Variables.

7. Show full trees, for each of the following tree types:

   a) a tree of type T_1, of all subsets of a given set,
   b) a tree of type T_2, of all permutations of a given set,
   c) a tree of type T_3, of all one-to-one functions from a set A to set B.

8. Show full trees, for each of the following tree types: How the strategy programs given above can be modified to include a genetic algorithm?

9. How the strategy programs given above can be modified to include a simulated annealing algorithm?

10. Explain in more detail possible pure and mixed strategies, that have been only mentioned in the last two sections.

11. Using the framework of creating strategies shown, can one create a superior Universal Strategy? How?

12. Discuss the role of functions CF, QF, f and q. Compare the problems from this and previous sections. Illustrate with problem examples.

13. Given is the type T_1 tree generator for a full tree of the set of all subsets, discussed above. What are the possible applications of combinatorial problems that occur in logic and high-level CAD, especially all functional decomposition problems? List all of them.

14. Create a tree like T_1, but starting from an arbitrary subset of a set. Use a small adaptation of rules for T_1.

15. How a search program can be created, that will find all k-element subsets of a set with K elements?

16. How a search program can be created, that will find all k-element variations of a set with K elements?

17. How a search program can be created, that will find all k-element combinations of a set with K elements?

18. Give examples of descriptors in a practical problem. What are the solution spaces?

19. What are the types of descriptors? Give examples of applications.

20. Give examples of macro-operators.

21. For each tree from Figure 9.2 explain the following:

    a) what set is created in this tree?
    b) what are possible applications of this set?
    c) what are possible applications of this tree?

    Write the specification of the full tree generator. Draw an example how to expand the tree to a larger dimension using your generator. Explain what would be a possible application of this set or this tree.

22. For each tree from Figure 9.3 do the following:

    a) explain what set is created in this tree.
    b) explain what are few possible applications of this set.
    c) explain what are few possible applications of this tree.
    d) write the specification of the full tree generator.
    e) draw an example of the tree expanded to a larger dimension using your generator.
    f) explain what would be a possible application of this set or this tree.

9.3 Methodology to Create Tree Search Programs

In this section we will present the methodology to create tree search programs. First, the problem-solving model will be introduced, which will next be used to explain the experiments that are executed on it by both the developer and the user.

9.3.1 Problem Solving Model

The following is a model that describes how the developer, starting from the most general problem formulation, should proceed to create the completed code of search program.

Problem Formulation

The first task of the code developer is to formulate a given problem that is to be solved. The problem is understood according to the definition of the problem, given in section 9.2.

- *The types of given objects* (sets, functions, graphs, etc.) as well as the types of objects sought by the program, should be specified. If these objects are not the standard objects of the system, they should be defined or the problem needs to be re-formulated in such a way that it will be possible to use the system's standard objects.
- The developer should describe the problem conditions using these structures. He or she should also define the cost function, if it exists. He considers, how the problem conditions and the constraint conditions can be decomposed or re-formulated.
- It is determined, whether the conditions need to be checked separately or jointly.
- The developer has also to determine the order in which the conditions should be checked. Let us assume, for example, that a set of objects being generated by generator G that satisfy the problem conditions C_1 and C_2 need to be found.

    1. One of the methods would be to generate the set S_1 out of all the objects generated by G that meet the condition C_1 first, and then to test the S_1 objects for the satisfaction of condition C_2.
    2. The procedure in another method would be identical except that the conditions would be reversed. Subsequently, the objects could be generated one at a time using generator G and both conditions checked for each generated object. In this case, we also have to decide on the order of checking the conditions.
    3. Finally, another generator that would generate only those objects generated by G that pass the C_1 test, etc. can be constructed.

Many methods can be derived from the direct problem formulation based on the methodology provided here. The developer then determines which one is best, and in some cases creates a parametrized version to leave the final decisions to the user of the system for its testing on numerous benchmarks.

Creating the Method to Solve a Problem

The developer who already understands the problem structure well, can subsequently move on to the stage of formulating the *method to solve the problem.* In doing so, it is recommended that the *similarities of problems* and the *possibilities of reducing problems to other problems* be systematically and intentionally applied, such as:

- is this a covering problem?
- is this an ordering problem?
- is this an constraint-satisfaction problem?
- is this a path search in some graph?
- is this a partitioning problem?
- is this an encoding problem?

The similarity of problems and the reduction of combinatorial problems offer an experienced developer many opportunities, which include using the already existing subroutines of the system for new problems. A number of useful reductions is presented, for instance, in [785]. Other reductions for digital design problems are discussed in [?, ?, ?, 405] and [?].

When the developer considers the combinatorial problems that are useful for his design problem, he specifies whether the new problem is *isomorphic, similar,* or *reducible* to one of the familiar problems. If such a relation between problems is found, this relation can be used, for example, in one of the following:

- restricting the solution space,
- constructing the generator for this space,
- proving theorems related to cutting off or to equivalences in the tree,
- detecting and utilizing symmetries,
- creating the quality functions and cost functions,
- specifying relations on descriptors or nodes in the tree,
- specifying other method conditions.

If only one similarity exists in the problems, such as the cost function, the generator of the space, the conditions, the operators. Next, using this similarity, the user can made an attempt to re-formulate the problem in such a way that the number of similarities is increased. If there are no similarities and the problem cannot be reduced to some well-known problem, an attempt at decomposing the problem into well-known sub-problems can be made. In this way, part of the problems, or perhaps all of them, should be isomorphic, similar or reducible to one of the well-known problems.

Often, it is also useful to find a different problem that is simpler than the original problem, and for which one of the techniques mentioned above can be applied.

Finally, such problems can be transformed by modifying new problem conditions, or by adding new problem conditions, in order to conform with one of the previously mentioned cases. This can lead to another problem with respect to the specified earlier definition of the problem. This happens quite often, because actual problems are not precisely formulated.

The successful application of the above techniques can simplify the selection of the generators and the conditions of the solution space, and make it easier to work with the prototype development later on.

Until now, we used most of these techniques in one or another form in development, but we did not yet analyze them systematically.

Precise Definition of the algorithm

The next step is to formulate precisely, what is the *tree-search algorithm.* The existing specific "point" subroutines, such as Graph Coloring or Boolean Operations on BDDs, determine certain *space of methods.* The task of the application developer is to determine how to link them in the most efficient way. Therefore, certain subspace in the space of methods should be systematically investigated.

All of the steps taken are intended to define the method that is computationally efficient. Three basic directives result from the above general assumption:

1. The solution space needs to be limited by constructing a good tree generator.
2. The best possible way of extending the nodes in this space (order of generation) should be found.
3. The processing time and the memory used for each node should be decreased.

The developer complies with these directives by specifying the corresponding generators and conditions for the solution space, and by the selection of appropriate strategies. Therefore, six possible cases of improving the tree search exist for the developer, as shown in Table from Figure 9.4.

	Specification of Tree Generation and Conditions	Selection of Strategy
Limiting the state-space	M1	M4
Specifications of the best way of extending nodes	M2	M5
Decrease of the processing time and the memory used for each node	M3	M6

Figure 9.4: Six Possible Cases to Improve the Tree Search Methods

Discussion of Methods to Increase the Search Efficiency

Six methods to increase the search efficiency are outlined in detail below. Bear in mind that they are all mutually related.

CASE M1.
In each problem, the developer should focus first on maximally decreasing the solution space, by specifying the corresponding generators and conditions. To achieve this, the developer should consider the way in which the states need to be represented, so as to cut-off and check the conditions in these states in the most efficient manner.

- What should the initial node be?
- What the coordinates should be, the operators, the descriptors?
- What should the additive cost function be? The additive cost function should be formulated in such a way that it can be used for cutting off. The function can be defined in a more or less precize way, thus the final function creation may be deferred, not earlier than some experiments with the search are performed.

While maintaining a completeness of the strategy, the developer should focus on proving the theorems related to limiting the space.
These can be of the following types.

1. **The theorems to determine the solution space.** In some problems it can be proven that certain space S' exists, $S' \subset S$, such that $SS \subset S'$ (SS is a set of solutions). The theorems concerned with determining the solution space can be of two types. They either lead to a space-constructing mechanism that is built into a special tree generator, or they lead to new constraint conditions that are tested for nodes or descriptors.

2. **The theorems about the limit parameters that specify the solution tree.** In some problems, such parameters as SD_{max} or $CF_{min\ min}$ can be determined. Let us assume, for example, that a single, minimal coloring of a planar graph needs to be found, and it was shown that no solution of cost 3 exists, because a clique with 4 nodes exists in the graph. It is a known mathematical fact that a planar graph can be colored using 4 colors. Hence we set the control parameter $CF_{min\ min} = 4$ and when a solution with 4 colors is found, the algorithm terminates.

3. **Theorems about relations on descriptors.** Often, relations of subordinance, dominance, equivalence (local and global), inconsistency, symmetry, or implication, can be found and proven. The utilization of analogies of problems is useful (for example, [760] uses analogies to efficiently solve highly cyclic covering problems). It needs to be ascertained whether the relations on the descriptors are independent. Also, the developer should consider the possibility of checking only some of the subsets of those relations.

4. **Theorems about the relations between the tree nodes.** The relations of identity, isomorphisms, symmetry, and the similarity between the nodes in the solution tree should be considered, [760].

5. **Theorems about the construction of generators.** The possibility of moving the constraint conditions to the tree generator itself should be taken into consideration. Instead of first generating nodes and later removing them, fewer nodes would be generated in this approach. If, for example, the subset of a certain set is sought that does not fulfill a certain restricting condition W, then instead of generating the tree of type T_1 and checking the condition W in the nodes of the tree, it is more reasonable to construct a more efficient generator that would only generate the subsets of the set that do not meet condition W. However, this may be a more time consuming way. Therefore, trade-offs must be thoroughly considered by the developer.

CASE M2.
In case M2 from Figure 9.4 one contemplates which of the elements in the state space has to correspond to the initial node of the solution space. In the problems concerned with finding a path, for instance, the search can be performed from the final situation to the initial situation. The components discussed in the types 3,4 and 5 above also influence the search method as well as the ordering of the auxiliary sets (see example in [?]). In the Subset Selecting Problem (for instance a set of bound variables, or maximum clique) the developer can start from the largest, the middle, or the smallest elements of the lattice of subsets as the starting point.

CASE M3.
In case M3 from Figure 9.4 the processing time can also be decreased for the following reasons

a. **Selection of the corresponding order of checking the problem conditions and constraint conditions** while generating new nodes. Speed is also related to specifying the order in which the coordinates of the state vector are calculated. The problem conditions or constraint conditions can either be calculated jointly or the conditions can be decomposed to several conditions that are calculated in order. These are calculated intermittently using the values of some coordinates to allow for backtracking sooner. The various placement possibilities are considered along with the decomposition of conditions and the influence that they have on cases M1 and M3.

b. **Formulation of weaker relations on descriptors** (which is in opposition to the directive related to limiting the solution space). Let us consider the following example as an illustration. It could be possible to generate a small tree by applying strong relations on descriptors, but the search could take a longer time, because the relations would be checked slower. On the other hand, replacing this strategy with one that applies weaker relations that are checked rapidly produces a larger tree, the nodes in this tree are extended with in a reduced time. The total processing time of some problems can, therefore, be decreased if no memory limitation exists.

c. **Simplification of the quality functions** (which is opposed to the directives resulting from cases M4 and M5 from Figure 9.4, which will be presented below). The argumentation for this case is similar to that in "b" above.

d. **Specification of the type of nodes that are suitable to be checked in the relations on nodes** (it is sometimes sufficient, for example, to investigate only the nodes that have the same depth, or that are in the same branch of the tree as the current node N).

e. **Selection of the *additively calculated* cost function and quality function** in such a way that they are both mutually related. Additively calculated is the function that is non-decreasing from the root to leafs of the tree and that satisfies certain conditions. See discussion in [359, 760, ?].

f. **Selection of the corresponding data structures of the lowest levels and the auxiliary functions** influences both the processing time and the memory size. In some problems it is reasonable to represent sets as the lists or vectors, and in other problems to represent them as binary words in which each set element corresponds to one bit of the computer word.

CASE M4.
In case M4 from Figure 9.4 the solution space can be limited by the selection of the strategy. In case of complete strategies, the only possibility for such a selection is the application of the cutting-off principle that is based on the value of the cost function. The problem is this way reduced to Case M5 below, in order to find the quasi-optimal solution as quickly as possible. The cutting-off, however, can result from the selection of some local strategy parameter values for incomplete strategies.

CASE M5.
In case M5 from Figure 9.4, the developer needs to consider whether some of the basic strategies match the specifics of the problem. The values of each strategy parameter, different combinations of parameter values, and the quality functions for states and operators should be considered. The possibility and the rationality of focusing on matching the quality function to the cost function should be analyzed, so that they will satisfy the special properties mentioned previously. When the characteristic of the problem is known, it should be determined which of the strategies discussed is the best fit to the specifics of the problem. Applying the subroutines for ordering and selection should be considered by the developer, and next by the user, while selecting the corresponding strategy parameters.

Case M6.
If there is not enough memory for larger data, one should consider the following possibilities:

a. Changing the strategy (out of the complete strategies, the best is the Depth-First Strategy with One Child). If this is not sufficient, the Random search strategy should be applied, or the developer should use the disk memory for part of the tree.

b. Increasing the number and the power of the cutting-off rules. These rules are controlled by the strategy parameters. Adding more constraining rules, and making them more powerful can lead to losing the completeness of the method. Such and approach can also reduce the calculation time.

With respect to the cases shown above, the developer can expand the concept by experimenting with the program from the simplest to the more complex problem descriptions, and from the standard strategies to those especially created strategies for the problem. For instance, these non-standard strategies can be made by replacing the standard parameters and sections with the non-standard parameter values and code sections. Analysis of some of the above cases is important not only with respect to the efficiency, but is also very useful for the clarity of the results and the required interface to other programs. For instance, if the complete tree is being extended, the strategy does not affect the efficiency. However, the selection of the strategy is not irrelevant when a certain order of generating the objects is mandatory or expected.

The methodology outlined above has allowed us to create several classical logic design algorithms in the past [?, ?, ?, ?, ?, ?, ?]. These include state minimization or various PLA minimization algorithms created by formal transformations of space generators and conditions of the direct problem description. Several new algorithm variants for these problems have also been derived using the same methodology.

9.3.2 Experimenting with Directives by the Developer, and by the User

After having introduced in previous subsections the methodology of creating search programs, we will discuss now the manner in which the tree searching programs can be improved upon, by experimenting with orthogonal changes in various description sections. This has already been illustrated to some extent in section 9.3.1. These principles will be further illustrated using several examples of problem descriptions.

The advantage of the state-space methods is the natural manner in which they are described and their similarity to the human problem-solving techniques. This description allows for the introduction of various *heuristic rules* to the state-space generation process that are direct and easy to modify. The prototype programs created as recommended in previous sections are flexible and adaptable and can be modified by replacing the section codes in problem descriptions, problem and solution conditions, the search strategy parameters, and other sections. This can be done also by adding new classes derived from previous classes. The modification process is *orthogonal* and *incremental* in nature, which means that the changes in various segments of the code can be done separately, and one at a time. They exhaust the space of search methods.

Heuristics

A "heuristic" is any method, technique, or directive, that, in general, leads to a solution. We can only assume, with a certain degree of plausibility, that a given *heuristic directive* produces the desired results. In our case, the heuristics are expressed as subroutines, program segments, parameter segments, or computational mechanisms that expedite the generation of the optimal solution (or quasioptimal solution, if so desired) and even enables the developer to find a solution that would otherwise have been impossible.

If a heuristic directive is proven to lead to optimal solutions all of the time, it can be classified as a *methodic directive* and would no longer be referred to as a heuristic. However, it is usually necessary to have both types of directives because, without directives, the enormously large state-space would make finding solutions impossible. The methodic directives are based on the strict and formal analysis. They can relate to certain parameters that limit the solution space, the parameters that order the descriptors in the solution space nodes, or to the parameters that order such nodes themselves in the $OPEN$ list. The solution method is complete when both the tree generator and the strategy are complete. The search efficiency can be increased through independent attempts to ease the condition governing the method or the strategy completness. This independence is advantageous when experimenting with variants of the program and with various sizes of data for it.

The *heuristic directives* considered here are divided into three categories:

1. Those that correspond to the manner in which the problem is formulated. These directives determine the state-space. The introduction of additional problem conditions (or reasonably motivated constraints) can essentially reduce the size of the state-space.

2. Those that are related to the manner in which the problem is represented in the solution space.

3. Those that organize the search process.

When it is not possible or reasonable to specify a complete solution space, we look for the heuristic directives that attempt to achieve the same goals as the methodic directives that were described in section 9.3.1. Each of the cases M1 - M6 has its counterparts in heuristic directives.

The categorization of any particular heuristic is not unique. It is often useful, when seeking heuristic directives, to rely on some analogies of the problems, and to use the analogies of the methods applied in their solutions. The existence and type of the heuristic directives influences the speed and the quality of the solutions generated. The developer should look for the best trade-offs. An analysis of various variants of a program written for a single application reveals the presented already trade-offs between the enumeration and the knowledge-based reasoning. The initial, direct problem description uses only the problem conditions. However, the initial program needs not be the most efficient one, and usually it is not. It is sufficient if the initial program works correctly and as specified. Such program is obtained quickly, so that more experience is gained by the developer working with this program. A very large tree is generated and displayed. The developer can analyze the printouts of the large trees and suggest changes based on redundancies, symmetries and other information found in these trees. The obviously redundant or non-optimal sub-trees are noted. Usually, the analysis as to why they were generated, leads the developer to the introduction of new problem conditions. Often also to formulation new heuristic directives, and sometimes also methodic directives. Incremental changes lead to new variants. The behavior and speed of them are analyzed one by one by the developer, so that the optimal trade-offs between the speed and the quality of results will be found. The limitation of the search can sometimes go too far - solutions are lost or the search time increases because the manner in which the partial solutions are evaluated becomes too complex. In such cases, the developer should withdraw incrementally from some of his previous decisions. In some other cases, the developer initially "over-constrains" the method description. Therefore, he will have next to experiment with several sets of less constrained descriptions.

The Process of Fast Prototyping

Constructing the program is a complex process in which three phases are iterated:

- problem definition,
- construction of the algorithm,
- coding of the algorithm.

Coding itself is not the most time consuming task of programming. More time is spent on the problem formulation, testing of variants, debugging, trying various program segments and modifications in the data structure, testing various controls of programs, analyzing the usefulness of heuristic directives, preparing and modifying documentation, etc. Usually the algorithm is initially not known. It is the goal of this report to facilitate the task of finding the correct algorithm and implementing the program. Experimenting with the program towards this end should be simplified by planning and keeping each individual phase independent from other phases, as much as possible.

In order to solve a problem, when aided by the computer, the developer must have the knowledge required in the three phases outlined above. Since these phases of the design process, to varying degrees, are contained in the program experimentation process, the developer needs to keep in mind the aspects of the problem and method description already mentioned. The entire prototype development should consist of the sequence of the vertical transformations described in section 9.3.1 and the horizontal transformations from this section, intermingled with experimental variant evaluations. People learn heuristic directives with the experience of solving a great number of problems that are repetitive in nature. In fact, our methodology postulates that *such learning is facilitated and expedited by the human problem solver's use of the appropriate programmed computer, and not only through the use of pencil and paper.*

Observing human techniques of solving problems we see that people apply several intuitive hypotheses that are often informal analogies to the cutting-off methods of our more formal search model. These hypotheses are often formed based on induction from observing examples of how the program behaves. Sometimes, with new methods, the representation of the problem is also modified or even totally changed. These hypotheses are often not true, not always true, or not sufficiently founded. They play a very fundamental role, however, in the trial-and-error process of constructing the final version of the prototype program. The methodology proposed here has the aim of *improving the fundamental structure of human heuristics by providing better proofs, based on the model of solution space.*

We believe that such an experimental search environment helps to design algorithms that are experimental in nature and must, therefore, be done in an experimental manner. We observed several times the dogmatic tendencies with which the students, engineers, CAD tool developers and even researchers view algorithms out of textbooks and research papers. Creating an algorithm systematically and playing with many variants of the algorithm crushes these doctrinaire approaches of them, and makes them more critical and also more creative. The proposed methodology encourages to look for new ways of solving problems.

A reader may ask - "is it worthy to analyze all these problems that carefully, and with such a detail? How much improvement will it bring practically to the problem being solved? " In our opinion, this analysis may prove very useful, since it is a result of tests on many programs, that even small changes in strategies can lead to dramatic changes in the solution cost, the decomposition times, and their mutual trade-offs (see section ?? and ??).

9.3.3 Problems

1. Select one combinatorial problem that is not trivial and that you know really well, possibly you wrote a program for it, or at least created an algorithm. Follow the methodology outlined above to create several new ideas and methods to solve this problem.

2. Follow the methodology outlined above to create several new ideas and methods to solve the Set Covering Problem.

3. Follow the methodology outlined above to create several new ideas and methods to solve the Graph Coloring Problem.

4. Follow the methodology outlined above to create few new ideas and methods to solve the Column Minimization Problem.

5. Follow the methodology outlined above to create few new ideas and methods to solve the Parallel Decomposition Problem.

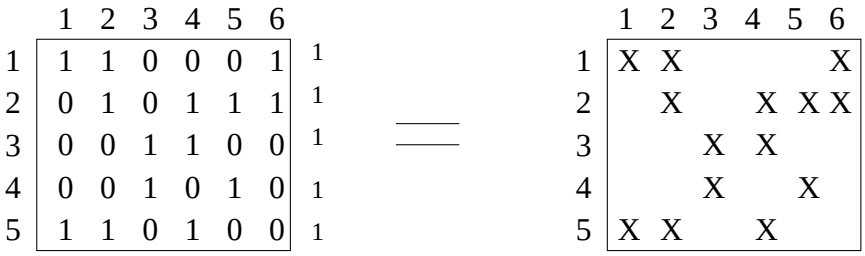


Figure 9.5: A Covering Table With Equal Costs of Rows

9.4 Example of Application: The Covering Problem

The following examples of some partial problems will illustrate the basic ideas involved in the state-space search. The examples will show also the methods that are used to formulate problems for multi-purpose search routines like those proposed in previous section.

9.4.1 The Formulation of the Set Covering Problem

This problem is used in Column Minimization. It is also widely encountered in logic design (among others, in PLA minimization, test minimization, multilevel design - see many recent examples in [405]). As an example, let us consider the covering table shown in Fig. 9.5.

Each row has its own cost indicated by the value to the right of it. In this example, they are all equal. An **X** at the intersection of row r_i and column c_j means that row r_i covers column c_j. This can be described as:

$$(r_i, c_j) \in COV \subset R \times C, \tag{9.39}$$

or briefly, by $COV(r_i,\ c_j)$.

A set of rows which together cover all the columns and have a minimal total cost should be found.

The direct problem formulation is as follows:

1. **Given:**
   a. the set $R = \{\ r_1\ ,\ r_2\ ,..., r_k\ \}$ (each r_i is a row in the table)
   b. the set $C = \{\ c_1\ ,\ c_2\ ,..., c_n\ \}$ (each c_j is a column in the table)
   c. the costs of rows $f1(r_j)\ ,\ j\ =\ 1, ..., k$
   d. the relation of covering columns by rows is $COV \subset R \times C$.
2. **Find**
   Set $SOL \subset R$
3. **Which fulfills the condition:**

$$(\forall\ c_j\ \in\ C)(\exists\ r_i\ \in\ SOL)[COV(r_i, c_j)] \tag{9.40}$$

4. **And minimizes the cost function**

$$f2\ =\ \sum_{r_i\ \in\ SOL} f1(r_i) \tag{9.41}$$

It results from the above formulation that the state-space $S = 2^R$. This means that $SOL\ \subset\ R$. Hence, it results from the problem formulation that all the subsets of a set are being sought. Then, according to the methodology, the standard generator, called T_1, that generates all the subsets of a set is selected. Operation of this generator can be illustrated by a tree.

The previously mentioned relation RE on the set $S \times S$ can be found for this problem and used to reduce searching for a respective search method. It can be defined as follows:

$$s_1 \; RE \; s_2 \iff s_2 \supset s_1 \tag{9.42}$$

Therefore, when a solution is found, a cut-off occurs in the respective branch.

There exists for each element $c_j \in C$ an element $r_i \in SOL$, such that their relation COV is met. In other words, r_i covers c_j which means that the predicate $COV(r_i, c_j)$ is satisfied. The cost function F assigns the cost to each solution. In this case, this means that $F = f2$ is the total sum of $f1(r_i)$; the costs of rows r_i that are included in set SOL. Thus, using the problem definition from section 9.2, the covering problem is formulated as the problem

$$P = (2^R, \{p_1\}, f2), \tag{9.43}$$

where

$$p_1(SOL) = (\forall \, c_j \in C) \; (\exists \, r_i \in SOL) \; [COV(r_i, c_j] \tag{9.44}$$

Tree Search Method # 1

The initial tree search method based on the direct problem formulation is then the following

1. The initial node N_0: $(QS, \; GS, \; F) := (\emptyset, \; R, \; 0)$.
2. The descriptors are rows r_i. The application of the operator is then specified by the subroutine

   $O(N, \; r_i) =$

   $[\quad GS(NN) := GS(N) \setminus \{r_i\}$

   $QS(NN) := QS(N) \cup \{r_i\}$

   $CF(NN) := CF(N) + f_1(r_i)$

   $]$
3. Solution Problem and Condition (cut-off type)

$$p_1(NN) = (\forall \, c_j \in C) \; (\exists \, r_i \in QS(NN)) \; [COV(r_i, \; c_j)] \tag{9.45}$$

Comments.

1. NN denotes a successor of node N.
2. $QS(N)$ is the set of rows selected as the subset of the solution in node N.
3. $F(N)$ is the cost function for node N. This is the total sum of costs of the selected rows from $QS(N)$.

As we can see in this problem, the formulation of the additive cost function is possible.

An example of the cover table is shown in Fig. 9.5. In this example, to simplify calculations, we assumed equal cost of rows. However, the method can be easily extended to arbitrary costs of rows. The solution tree obtained from such a formulation is shown in Fig. 9.6.

The nodes of the search tree are in the ovals. The arrows correspond to the applications of operators, and each descriptor of operator stands near the corresponding arrow. The solution nodes are shown in bold ovals. The costs of nodes are outside the ovals, to the right. The sets inside the ovals correspond to partial solutions in the nodes. Since the entire tree has been developed here, the sets GS for each node can be reconstructed as the sets of all descriptors from the outpointing arrows.

The cutting-off uses the fact that the cost function increases monotonically along the branches of the tree; this is the cut-off condition. The nodes that are the solutions are therefore not extended. If the cut-off conditions were not defined, for example, the nodes {1,2,3} and {1,2,4} would be extended. Otherwise, the tree is produced under the assumption that the cutting-off is not done for the solutions with cost function values worse than for those nodes previously calculated. The values of function f for nodes are shown to the right of these nodes.

Observe that some nodes of the tree are created (for example, node {3}) in a way that does not allow any solutions to be produced in their successor nodes. Because each column must be covered by at least one row in the node, the

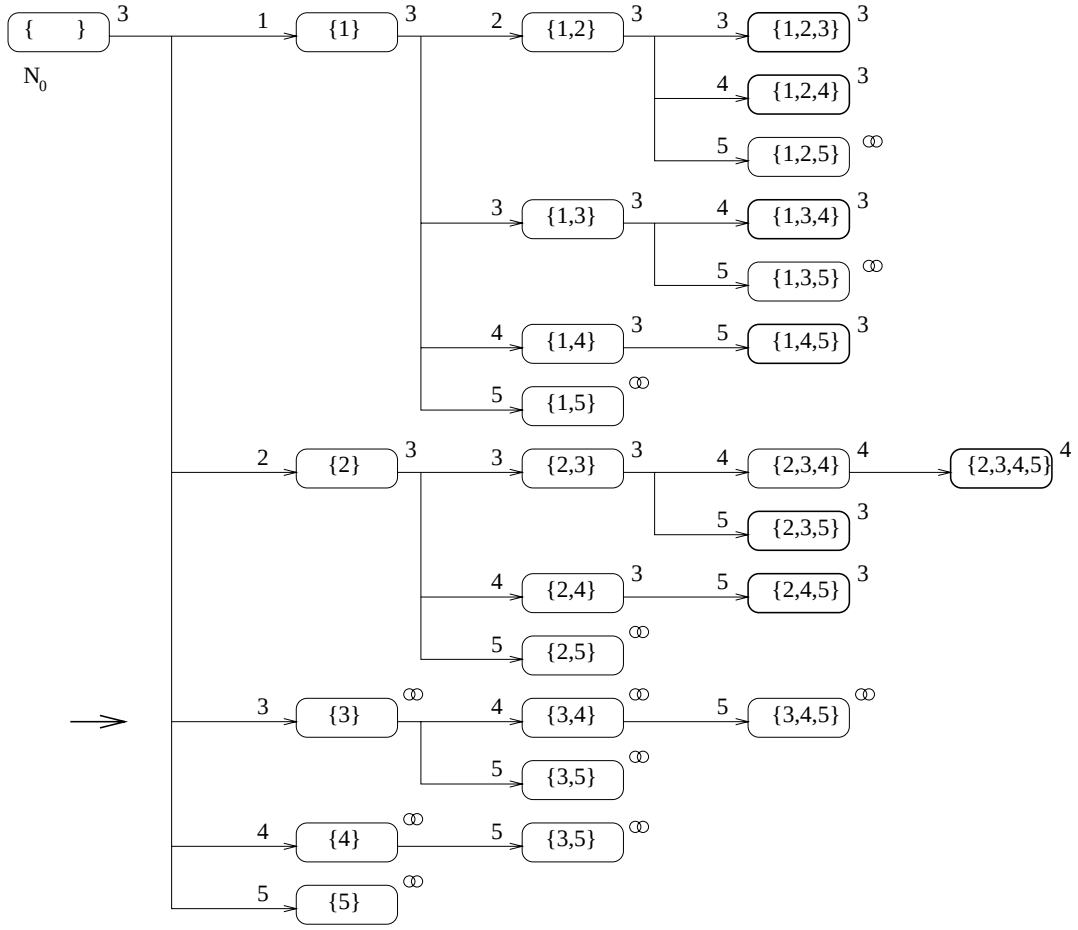


Figure 9.6: First Search Method for the Table from Figure 9.5

generation of such nodes can be avoided. This is done by storing the columns c_j that are not yet covered in set AS. The branching for all rows r_i that cover the respective column for each individual column is also generated. These are such rows r_i that $COV(r_i,\ c_j)$.

We can now formulate a new tree search method

Tree Search Method #2

1. Initial node N_0

$$(QS,\ GS,\ AS,\ CF)\ :=\ (\emptyset, \{r_k \in R\ \mid\ COV(r_k,\ c_1)\},\ C,\ 0) \tag{9.46}$$

   The first element of C is denoted by c_1 above.

2. Operator

   $O(N,\ r_i) = [$

   $QS(NN)\ :=\ QS(N)\ \cup\ \{r_i\}$

   $AS(NN)\ :=\ AS(N)\ \setminus\ \{c_j \in C \mid COV(r_i, c_j)\}$

   $c_j\ :=$ the first element of $AS(NN)$

   $GS(NN) := \{r_k \in R\ \mid\ COV(r_k,\ c_j)\ \}$

   $CF(NN) := CF(N)\ +\ f_1(r_i)$

   $]$

3. Solution condition (cut-off type)

$$p_1(NN)\ =\ (AS(NN)\ =\ \emptyset) \tag{9.47}$$

The corresponding tree is shown in Fig. 9.7.

Two disadvantages to this method become apparent from Fig. 9.7. The first disadvantege is creating the redundant descriptor 4 in $GS(N_5)$. This descriptor cannot be better than the descriptor 2. This disadvantage can easily be overcome by writing a new code for this section, that would define and use the domination relation on descriptors. The second disadvantage is due to the repeated generation of the solution {1,3,4}, the second time as {1,4,3}. If the optimal solution is desired, then there is no way to avoid the inefficiency introduced by the Tree Search Method #2.

Tree Search Method #3

Another method to avoid generating nodes for which $f = \infty$ is the application of the first method (the generation of the T_1 type of tree) and an additional filtering subroutine to check nodes to verify if the set of rows from $GS(N)$ covers all the columns from $AS(N)$. In addition, the following code of type *"Actions on the Selected Node"* is created:

If

$$AS(N)\ \not\subset\ \{c_j\ \in C\ \mid\ (\ \exists\ \ r_i \in GS(N))\ [COV(r_i,\ c_j)]\} \tag{9.48}$$

then

$$GS(N)\ :=\ \emptyset$$

This means, that the cut-off is done by clearing set $GS(N)$ when the set of all the columns covered by the available descriptors from $GS(N)$ does not include the set $AS(N)$ of columns to be covered. For example, at the moment of generation shown by the arrow in Fig. 9.6, the set of $GS(N_0) = \{3,4,5\}$ and it does not cover $AS(N_0) = C$. Therefore, it is assigned $GS(N_0) := \emptyset$, and the generation of the subtree terminates. This forms the Tree Search Method #3.

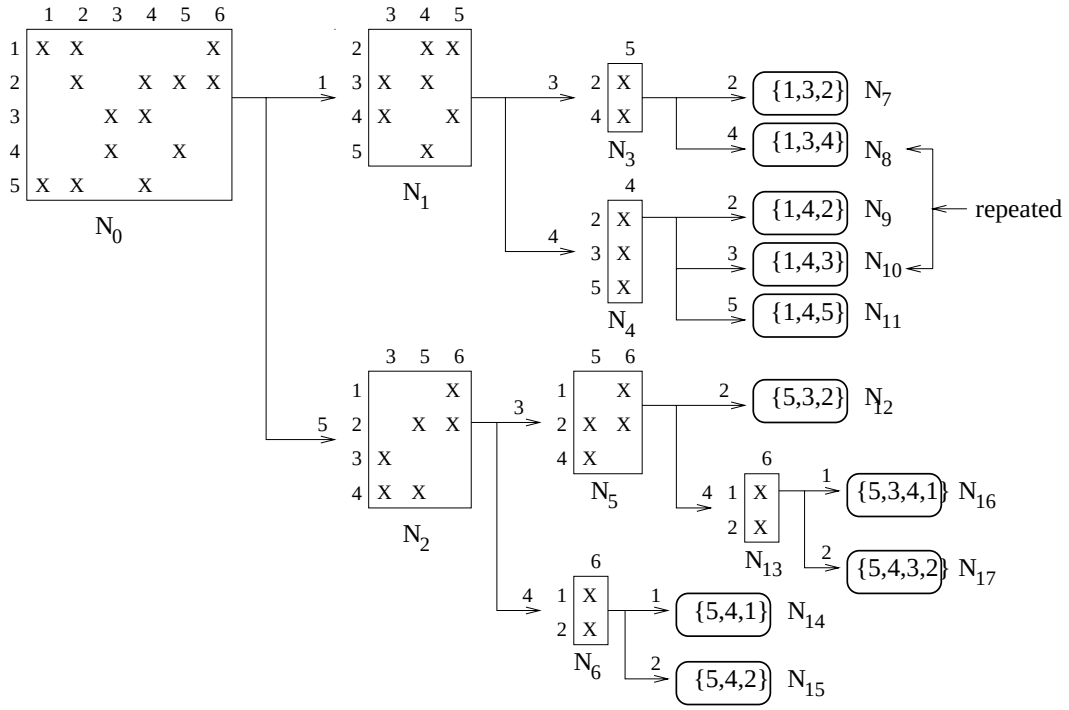


Figure 9.7: Second Search Method for the Table from Fig. reffig:Fig. 5.2

Tree Search Method #4
The generated tree can be decreased even further when the second method is used and it is declared in the operator that:

$$GS(NN) \ := \ \{r_k \in GS(N) \ \setminus \ \{r_i\} \ \mid \ COV(r_k, c_j)\} \tag{9.49}$$

Let us recall that symbol \ denotes operation of set difference.
However, this approach can cause losing the optimal solution. It is then a typical *heuristic directive* and not a *methodic directive* like those discussed previously.

In both trees, the cutting off condition based on the cost function has been not yet considered. If the solution {1,2,3} in the tree shown in Fig. 9.6 were first found, node {2,3,4} could be cut off, and the non-optimal solution {2,3,4,5} would not be generated. However, until now, only the methods of constructing the generator of complete and non-redundant trees have been presented. These are the trees calculated for the worst case of certain rules and heuristics that will be discussed in section 9.4.2.

9.4.2 Search Strategies

Various search strategies can be illustrated using this example, to give the reader an intuitive feeling for the concepts and statements introduced in the previous sections.

The node enumeration order from Fig. 9.7 corresponds to the Breadth-First strategy, and to the strategy of Equal Costs (with respect to the equal cost of rows applied in this example). Eight nodes were generated in node N_7 to find the optimal solution {1,3,2}. The optimality of the solution {5,4,2} was proven after creating node N_{15}, which means, after generating 16 nodes. Nodes N_7 to N_{15} were temporary. Cost-related backtracking occurs in node N_{13} and, therefore, nodes N_{16} and N_{17} are not generated.

The strategy Depth-First generates the nodes in the order N_0, N_1, N_2, N_5, N_6, N_{14}, N_{15}, N_{12}, N_{13}, N_3, N_4, N_9, N_{10}, N_{11}, N_7, N_8. After finding N_{14}, i.e., generating six nodes, the optimal solution {5,4,1} is found. As in the previous strategy, after generating 16 nodes, the optimality of the solution {1,3,4} is determined. We can state - *"it is proven"*, since the method is exhaustive, and we have generated all nodes.

The strategy Depth-First-With-One-Successor, generates the nodes in the order: N_0, N_1, N_3, N_7, N_8, N_4, N_9, N_{10}, N_{11}, N_2, N_5, N_{12}, N_{13}, N_6, N_{14}, N_{15}. The optimal solution {1,3,2} is found after creating four nodes. After generating 16 nodes, the optimality of the solution {5,4,2} has been proven. Because the selection of the descriptor depends on the row order among the rows covering the first column, the selection is arbitrary. Hence, in the worst case, the order of generation could be N_0, N_2, N_5, N_{13}, N_{16} (the temporary solution {5,4,3,1} of cost 4 has been found), N_{17}, N_{12},

N_6, N_{14}, N_{15}, N_1, N_3, N_7, N_8, N_4, N_9, N_{10}, N_{11}. A tree of 18 nodes would be generated to prove the optimality of {1,4,5}. This illustrates, that good heuristics are very important to limit the size of the solution tree.

Tree Search Method #5

Subsequent advantages will result from the introduction of the heuristic functions that control the order in which the tree is extended, with regard to the method #2 presented above. The introduction of such functions will not only lead to finding of the optimal solution sooner, but also to expediting the proof of its optimality. This is due to fuller use of the cutting-off property, which results in a search that is less extensive when the optimal solution is found earlier.

The quality function for the operators with regard to the selection of the best descriptors in the branching nodes, as well as the quality function for nodes with regard to the selection of the nodes to be extended is defined below.

Quality function for nodes:

$$QF(NN) = CF(NN) \; + \; \hat{h}(NN) \tag{9.50}$$

where

$$\hat{h}(NN) \; = \; CARD(AS(NN)) \; \cdot \; CARD(GS(NN)) \; \cdot \; K, \tag{9.51}$$

and

$$K \; = \; \frac{\sum\limits_{r_i \; \in \; GS(NN)} f_1(r_i) \; \cdot \; CARD \; \{c_j \; \in \; AS(NN) \; \mid \; COV(r_i, c_j)\}}{\left(\sum\limits_{r_i \; \in \; GS(NN)} CARD \; \{c_j \; \in \; AS(NN) \; \mid \; COV(r_i, c_j)\}\right)^2} \tag{9.52}$$

Such a defined function $\hat{h}$ is relatively easy to calculate. As proven in the experiments, it yields an accurate evaluation of the real distance h of node NN from the best solution. It is calculated as an additional coordinate of the node's vector. The function's form is an outcome of the developer trying to take into account the following factors:

1. The nodes N_i are extended for which the fewest columns need to be covered in the $AS(N_i)$. There is a higher probability that the solution is in the subtree $D(N_i)$ at the shallow depths for such nodes. Hence, the component $CARD(AS(NN))$.
2. The nodes, for which the fewest decisions need to be made, are extended. This is a general directive of tree searching. It is especially useful when there exist strong relations on descriptors, as happens in our problem. Hence, the component $CARD(GS(NN))$.
3. The coefficient K was selected in such a way that, with respect to the properties of the strategies discussed previously, the function $\hat{h}$ is as near to h as possible.

The quality function for operators is defined by the formula

$$q^{NN}(r_i) \; = \; c_1 f_1(r_i) \; + \; c_2 f_2(r_i) \; + \; c_3 f_3(r_i), \tag{9.53}$$

where c_1, c_2, c_3 are arbitrarily selected weight coefficients of *partial heuristic functions* f_1, f_2, *and* f_3 defined as follows

$$f_1 \; has \; previously \; been \; defined \; as \; the \; cost \; function \; of \; rows \tag{9.54}$$

$$f_2(r_i) \; = \; CARD\{c_j \; \in \; AS(NN) \; \mid \; COV(r_i, \; c_j)\} \tag{9.55}$$

$$f_3(r_i) = \frac{1}{f_2(r_i)} \sum_{j=1}^{n} \; CARD \; [r_e \mid c_j \in AS(NN) \; \wedge \; COV(r_i, c_j) \; \wedge \; r_e \in GS(NN) \; \wedge \; COV(r_e, c_j)] \tag{9.56}$$

where n is number of columns. Function $f_3\ (r_i)$ defines the "resultant usefulness factor of the row" r_i in node NN. Let us assume that there exist k rows covering some column in the set $GS(NN)$. The value of the usefulness factor of each of these rows with respect to this column equals k. When $k = 1$, the descriptor is *indispensable* (or with respect to Boolean minimization, the corresponding prime implicant is *essential*). The *resultant usefulness factor of the row* is the arithmetical average of the *column usefulness factors* with respect to all the columns covered by it. Then, one should add an instruction in the operator subroutine to sort the descriptors in $GS(NN)$ according to the non-increasing values of the quality function for descriptors q^{NN}.

The next way of decreasing the solution tree is by declaring new section code that checks the relations on descriptors.

If the descriptors r_i and r_j are in the *domination relation* in the node N (such relation is denoted by $r_i\ >\ r_j$), r_j can be removed from $GS(N)$ with the guarantee that at least one optimal solution will be generated.

If the descriptors r_i and r_j are in the *global equivalence relation* in node N, any one of them can be selected. The other descriptor is removed from $GS(N)$, as well as from $GS(M)$ where M is any node in the sub-tree $TREE(N)$. The *equivalence class* $[r]$ of some element r from $GS(N)$ is replaced in this coordinate by r itself. Descriptors declared as locally equivalent are treated similarly. The only difference is that the descriptors are then removed from $GS(N)$ only. Observe, that these relations are not based on costs, but on some additional problem-dependent information about the nodes of the tree that is available to the program. The covering problem may be a good example of this property.

The descriptors r_1 and r_2 are *globally equivalent* in node NN when they have the same cost and cover the same columns

$$r_1\ \equiv\ r_2\ \Longleftrightarrow f_1(r_1) = f_1(r_2)\ \wedge\ (\forall \in AS(NN))\ [COV(r_1, c)\ =\ COV(r_2, c)]. \tag{9.57}$$

Descriptors (rows) r_1 and r_2 are *locally equivalent* in node NN when, after removing one of them from the array, the number of columns covered by j rows is the same for each $j\ =\ 1, ..., CARD(GS(NN)) - 1$ as after removing the second one.

$$r_1\ \doteq\ r_2\ \Longleftrightarrow\ (\forall j = 1, ..., CARD(GS(NN)) - 1)\ [LK(j, r_1)\ =\ LK(j, r_2)] \tag{9.58}$$

where $LK(j, r)$ is the number of columns covered by j rows in the array that originates from $M(NN)$ after removing row r.

$$LK(j, r)\ =\ CARD\ \{c_k \in AS(NN)\ \mid\ CARD(X_k) = j\} \tag{9.59}$$

where X_k is the set of rows covering the column c_k

$$X_k\ =\ \{x\ \in\ GS(NN)\ \setminus \{r\}\ \mid\ COV(x, c_k)\} \tag{9.60}$$

Descriptor r_1 is *dominated* by descriptor r_2 when: (1) it has larger cost than r_2, and (2) r_1 covers at most the same columns as r_2, or when (3) r_1 has the same cost as r_2, and covers the subset of columns covered by r_2,

$$\begin{aligned} r_1 \leq r_2 \Longleftrightarrow f_1(r_1) > f_1(r_2) \wedge (\forall\ c_k \in AS(NN))[COV(r_1,\ c_k)\ =\ COV(r_2,\ c_k)] \\ \vee\ \ f_1(r_1) = f_1(r_2) \wedge \{c_k \in AS(NN)\mid COV(r_1,\ c_k)\} \subset \{c_k \in AS(NN)\mid COV(r_2,\ c_k)\} \end{aligned} \tag{9.61}$$

The developer can program all of the relations given above or only some of them. If all the relations have been programmed and parametrized, the user can still select any of their subsets for execution using parameters. The solution process is shown in Fig. 9.8.

Column 1 and rows 1 and 5 are selected at the beginning ($GS(N_0)\ =\ \{1, 5\}$). After the selection of row 1 to $QS(N_1)$, row 5 becomes dominated by 2 (or 3) and is removed. The domination of descriptor 5 by descriptor 2 is denoted in the Figure 9.8 by 2D5. Now descriptors 2,3,4 are locally equivalent, denoted as $LR(2, 3, 4)$. One of them, say 3, is selected. Descriptors 2 and 4 then become globally equivalent in node N_1' , denoted as $GR(2, 4)$. One of them, say 2, is selected. This leads to the solution $QS(N_1'')\ =\ \{1,3,2\}$. Now the backtracking to the initial node, N_0, occurs and descriptor 5 is selected. Next, descriptors 1 and 3 are removed since they are dominated and then descriptors 2 and 4 are selected as indispensable descriptors in node N_2' that is denoted as $IN(2, 4)$ in Fig. 9.8. This produces the solution $QS(N_2') = \{5,2,4\}$. After backtracking to the initial node $GS(N_0) = \emptyset$, node N_0 is removed from the open-list. The open-list $= \emptyset$ completing the search of the tree. The last solution of the minimal cost 3 is then proven to be the optimal solution.

Note the following facts

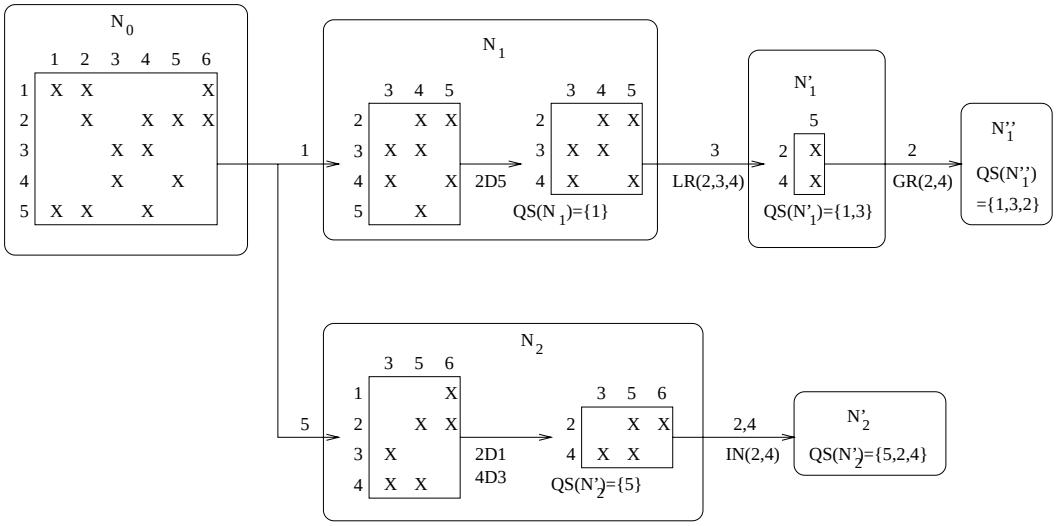


Figure 9.8: Final Search Method for the Table from Fig. 9.5

	1	2	3	4	5	6	7
A (4)			X	X			
B (3)			X		X		
C (3)							X
D (3)	X	X	X		X		
E (2)				X	X	X	
F (4)	X	X				X	

Figure 9.9: A Covering Table with Costs of Rows not Equal

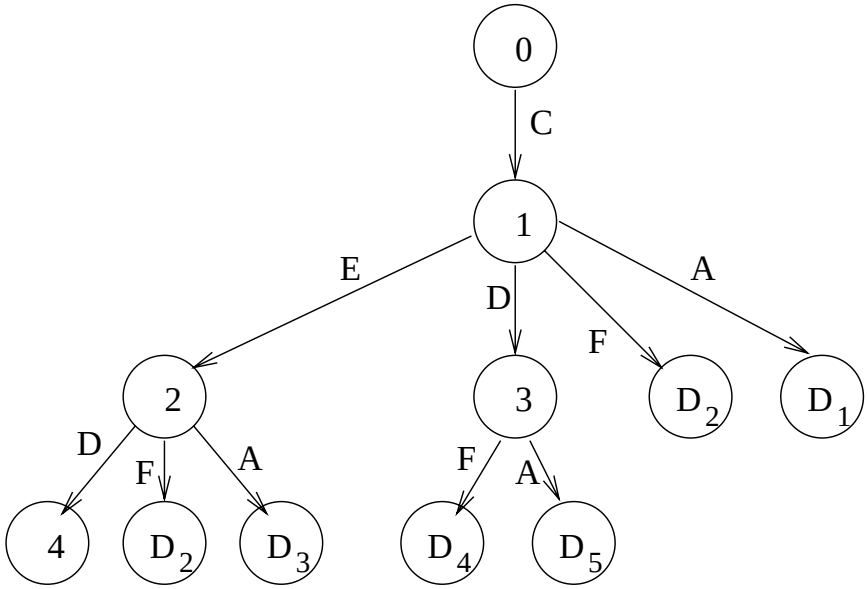


Figure 9.10: A Search Method for the Table from Figure ??

- not all of the minimal solutions were obtained but more than one was produced,
- only node N_0 is permanently stored in the tree,
- if the user declared parameter $F_{min\ min} = 3$, the program would terminate after finding the solution {1,3,2}. In some problems, guessing or evaluating the cost of the function is not difficult.

If all of the above relations, except the most expensively tested local descriptor equivalence, were declared, the complete tree consisting of 8 rows and 3 solutions would be obtained.

Yet another approach would be to select Branch-and-Bound and Ordering as global strategies and $MUST0$, $EQUIV$, $REAPNT$ to define the local strategy. $EQUIV$ only checks for the global equivalence of descriptors. The covering table shown in Fig. 9.9 will be solved in this example. The cost of each row is entered next to its respective descriptor. The costs of the rows are now not equal. The tree structured state-space for this problem is shown in Fig. 9.10. The details concerning the node descriptions for this tree are also illustrated in Table from Figure ??. (By *pred(N)* we denote the predecessor node of node N).

The search starts from node 0 where no column is covered, so set AS consists of all the columns. All rows are available as descriptors. Initial QS is an empty set, since no descriptor has been yet applied. After being processed by $EQUIV$, it is found that descriptor B is dominated by the another descriptor D. Therefore, descriptor B is deleted from the descriptor list. $MUST0$ finds that descriptor C is indispensable (with respect to column 7), and it is then immediately applied by GEN to create the new node 1. The descriptor list is then ordered by $ORDER$ using the quality function mentioned above. Assuming the coefficients $c_1 = 0.5$, $c_2 = 0$, and $c_3 = 1$, the costs of descriptors are

$$Q(A) \ = \ 2 + 4/2 = 4.0, \qquad Q(D) \ = \ 1.5 + 4/4 = 2.5, \tag{9.62}$$

$$Q(E) \ = \ 1 + 3/3 = 2.0, \qquad Q(F) \ = \ 2 + 4/3 = 3.3. \tag{9.63}$$

The descriptor list is arranged according to the descriptor costs as {E, D, F, A}. The descriptors are applied according to this sequence.

There is no difference between the application of the descriptors in the sequence of E, D or D, E for the solution in this problem. Therefore, if descriptors E and D have already been applied, it is not necessary to apply them again in another sequence. This is why descriptor E is cancelled for node 3; E and D for node D_0 ; as well as E, D and F for node D_1. This cancellation is done by $REAPNT$. The above procedure prevents node D_0 from finding the descriptor to cover column 5. Therefore, this node is not in the path to the solution and should be cut off by $GENER$. This phenomena also happen for nodes D_1, D_3, and D_5. The cost of node D_2, which is 13, exceeds the temporary cost B which is the cost of solution node 4 that has already been found. It was, therefore, cut off by the $Branch - and - Bound$ strategy. So was node D_4. The whole search procedure in this example deals with 11 nodes but only stores the descriptions of 5 nodes in the memory structure. A total of two optimal solutions were found in the search.

The methodology presented above illustrates the characteristic trade-off relationship between the knowledge-based reasoning and the exclusively intrinsic search already mentioned in section 9.3. The direct description of the problem allows us to find a solution based strictly on the generation of all possible cases that are not worse than the solutions generated previously. The successive addition of the information in the form of new heuristic directives and methodic directives that are based on the analysis of the problem and the solution process (e.g. quality functions, domination relations, equivalences, $F_{min\ min}$, etc.) allows for the search to be decreased.

Until now, we have not focused on how the relation COV is *represented.* This could be an array, a list of pairs (r_i, c_j), a list of lists of columns covered by rows, a list of lists of rows covering the columns, etc. The selection of the representation is independent from the selection of the method and from the strategy, but various combinations of these can have different effects. At some stage in creating the program, the user decides on the selection of, for example, the binary array and writes the corresponding functions. The user can then work on the representation of the array next: using words, or using bits. The arrays $M(N)$ also do not necessarily need to be stored in nodes as separate data structures, they can be re-created from $AS(N)$ and $GS(N)$. The local strategy parameters should also be matched to the representation. This is related to such factors as the total memory available for the program as well as the average times needed to select the node, to generate the node, to extend the node, to select the descriptor, and to check the solution condition.

From similar problems in the past, we found that the application of each of the equivalence conditions, dominance conditions, or indispensable conditions in the covering problem reduce the search space by about 2 to 3 times. The joint application of all the conditions brought about a reduction of approximately 50 to 200 times the generated memory.

	0	1	2	3	4
N	0	1	2	3	4
SD	0	0	1	1	2
pred(NN)	-	0	1	1	2
OP	-	C	E	D	D
F	0	3	5	6	8
NAS	7	6	3	2	0
NQS	0	1	2	2	3
NGS	6	4	3	2	0
AS	1~7	1~6	1,2,3	4,6	-
QS	-	C	C,E	C,D	C,E,D
GS	A~F	E,D,F,A	D,F,A	F,A	-

Figure 9.11: Node Descriptions for the Tree from Fig. 9.10

9.4.3 Problems

1. Create methods and strategies for Petrick Function Minimization Problem, that will make use of the similarity of this problem to the Set Covering Problem explained above.

2. Repeat Problem 1 for the Problem of Finding all Sets of Vacuous Variables for a binary single-output strongly unspecified function.

3. Repeat question 1 for the Problem of Finding all Sets of Vacuous Variables for a multiple-valued single-output strongly unspecified function.

4. Create methods and strategies for the POS Satisfiability Minimization Problem, that will make use of the similarity of this problem to the Set Covering Problem explained above. Remember, that contrary to the Petrick Function Problem, the POS Satisfiability can have no solution. If solutions exist, you want to find one that has the minimal number of positive (non-negated) variables.

5. Use the Set Covering Problem from this section as an example to create various methods and strategies for the Generalized Clique Partitioning Problem. Assume that each node of the graph has a cost, and the search is for the clique that maximizes the cost. What are the heuristics here? How can you use the heuristics?

9.5 Example of Application. The Graph Coloring Problem

The tree search algorithm for the Graph Coloring Problem given below colors successively nodes with actually available color of a smallest number, remembering for each node all remaining possibilities of coloring (it is assumed that initially the set of colors has as many elements as the set of nodes).

The chromatic number of the graph is equal to the number of colors in the exact solution. We apply the strategy: "depth-first with one child". After finding a solution that is better than a previous one (in a new branch) the algorithm has a new, improved evaluation of the chromatic number, so it can remove from sets $GS(N)$ all the operators that correspond to colors which were not included in the solution. This is a very simple example of a Branch Switch Strategy. The process of tree creation is continued using the cut-off principle based on cost function.

Algorithm for Proper Graph Coloring

This algorithm uses the Universal Strategy.
$NODES$ is the set of nodes.
$EDGES \subset NODES \times NODES$ is the set of edges.
$G = < NODES, EDGES >$ is the graph.

1. **Creating the Initial State**:

1. $NODE_1 := NODES[0]$, /* take first node from $NODES$ */
   create initial QS := $\{NODE_1$. 1)}.
   $NODE_2 := NODES[1]$, /* take second node from $NODES$ */
   if ($(NODE_1, NODE_2) \in EDGES$) or ($(NODE_2, NODE_1) \in EDGES$)
   then append ($NODE_2$, 2) to QS; $COLORS\_USED$:= {1,2}
   else append ($NODE_2$, 1) to QS; $COLORS\_USED$:= {1}.
   CF := CARD($COLORS\_USED$).
2. SET_OF_COLORS := {1,2,...,CARD(NODES) }.
3. $(QS, GS, REMAINING\_NODES, COLORS\_USED, CF) :=$
   ({ QS, $REMAINING\_NODES$, $COLORS\_USED$, CF)

2. **Operator** O(N, $COLOR$) = [

   a) SN := first element from $REMAINING\_NODES(N)$, /* SN = selected node */
   b) $REMAINING\_NODES(NN) := REMAININB\_NODES(N) \setminus SN$,
   c) $QS(NN) := QS(N) \cup \{ (SN, COLOR) \}$
   d) if $COLOR \in COLORS\_USED(N)$ then $CF(NN) := CF(N)$
      /* color exists, no change in cost */
      else
      $(CF(NN) := CF(N) + 1, COLORS\_USED(NN) := COLORS\_USED(N) \cup \{COLOR\})$,
   e) if $CF(NN) \geq CF_{min} \wedge REMAINING_NODES(NN) \neq \emptyset$
      then (cut-off, backtrack),
   f) $GS(NN) := SET\_OF\_COLORS \setminus \{FK(SN) \mid ((SN_i, SN) \in EDGES$ or $(SN, SN_i) \in EDGES) \}$
      ]

3 **Selected Strategy:** Depth-First With One Child.

4 **Switch Condition for Branch:**

If $REMAINING\_NODES(NN) = \emptyset$ then
for each N_i in branch from N_0 to NN do
$GS(N_i) := GS(N_i) \cap COLORS\_USED(NN)$

COMMENTS

1. Coordinate $QS(N)$ includes a partial coloring of the graph, it means the set of pairs (j, $FK(j)$) where $j \in NODES$. In the initial state two inconsistent nodes are colored with different colors 1 and 2, and two nodes n_1 and n_2 such that (n_i, n_2) $\notin EDGES$ with the same color 1.

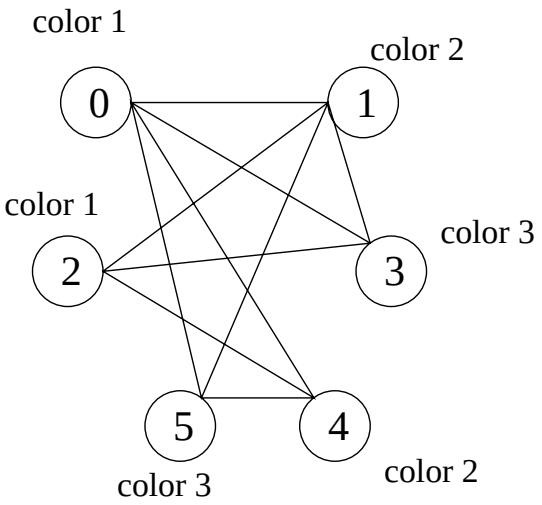


Figure 9.12: Graph for Coloring to Example ??

2. $GS(N)$ is the set of colors, which could be used to color the given node SN. $REMAINING\_NODES(N)$ are the set of non-colored nodes, and $COLORS\_USED(N)$ are the set of used colors.

3. In the operator, to increase the efficiency, the step e) of cutting-off before calculating $GS(NN)$ has been applied. As the candidates to color the selected node SN one can select the colors which are different from colors used for nodes that are linked with SN by edges (step f) above).

4. In the moment of finding solution $QS(NN)$ in node NN it is known that the minimial solution needs at most as many colors as in $COLORS\_USED(NN)$. Therefore one can use only a subset of $COLORS\_USED(NN)$. In the Switch Condition for Branch one can thus remove from all coordinates $GS(N_i)$ (for all nodes N_i in the branch from N_0 to the final node NN) the colors that do not belong to $COLORS\_USED(NN)$, since we are not interested in all possible colorings, but only in the colorings with the accuracy to an isomorphism (i.e the minimum chromatic number colorings).

5. Let us note that we have to deal here with two kinds of nodes: nodes of the graph, and nodes of the search tree.

Example 9.1

The tree for the graph from Figure 9.12 is shown in Figure 9.13. (For clarity, only some coordinates of the state vector are shown).

The search process is executed as follows.

1. The initial pair of graph nodes is 0 and 1, which obtain colors 1 and 2, respectively.

2. Next graph node 2 can obtain one of colors {1,3,4,5,6}. Selected is 1.

3. Now graph node 3 can obtain one of colors {3,4,5,6}. 3 is selected.

4. This way, the first branch of the tree from Figure 9.13 is created.

   After finding the first solution:

   color 1 = {0, 2},

   color 2 = {1, 4},

   color 3 = {3, 5},

   we know that three colors are enough, and these colors are 1,2, and 3.

5. All non-used colors are then removed from sets $GS(3)$, $GS(2)$, $GS(1)$ and $GS(0)$. Now backtrack to node 3 of the search tree occurs.

6. $GS(3) = \emptyset$, so next backtrack is to node 2 of the search tree. Further, $GS(2) = \emptyset$, so next backtrack is to node 1 of the tree is done.

7. Now, $GS(1) = \emptyset$, so next backtrack is to node 0 of the search tree is done.

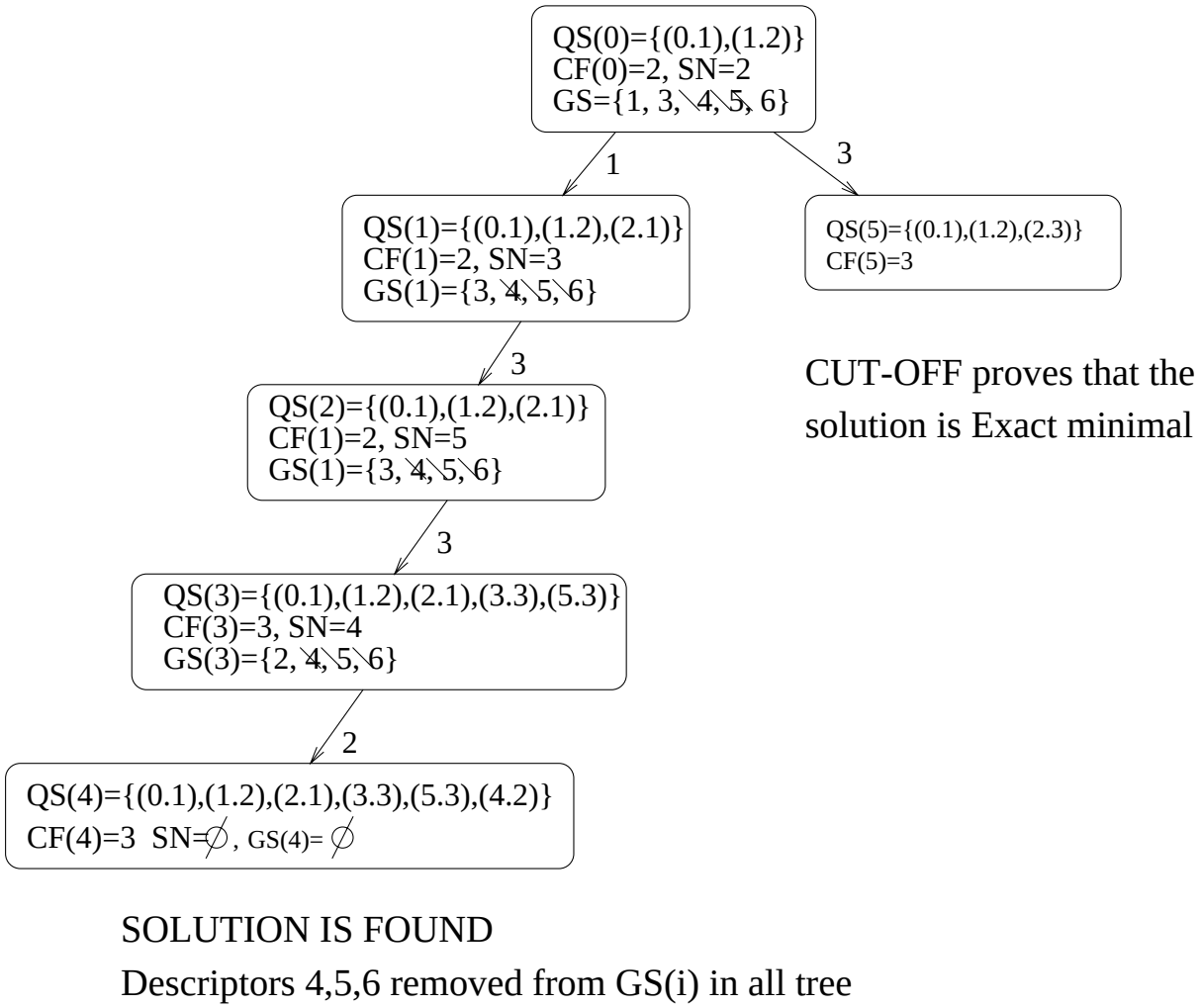


Figure 9.13: Tree Search for the Exact Graph Coloring Algorithm to Example ??

8. Now, $GS(0) = \{3\}$ so as a result of execution of operator O(0,3) node 5 of the tree is created. In this node, $CF(5) = 3$ and $NODES(5) = \{3, 5, 4\} \neq \emptyset$, so cut-off is executed and backtrack to node 0 of the tree.

9. Now $GS(0) = \emptyset$ and the program terminates its operation with the coloring found in node 4 of the tree. However, now we have also a proof that this is the exact minimal coloring.

9.5.1 Problems

1. How to modify the above algorithm to make smarter choices of successive nodes for coloring? Can you use the node density or other local graph node-related parameters? Can you use the cycle measures or other local graph cycle-related parameters? Write Lisp program.

2. How to modify the above algorithm to make an additional, secondary, requirement that the numbers of nodes colored with the same color will be approximately equal? Write Lisp program.

3. How to modify the above algorithm in such a way that nodes not connected to other nodes remain not colored at all? What would be the use of this property in Column Minimization Problem. Write Lisp program.

4. How to modify the above algorithm to make it a Multi-Coloring Graph Coloring? By Multi-Coloring we understand an algorithm in which a node can be colored with as many colors as possible, but there is never a conflict between any two nodes, it means, sets of colors assigned to any two adjacent nodes are disjoint. What would be the use of this property in Column Minimization Problem? Write a Lisp program similar to one above.

5. How to modify this algorithm that it will start from coloring large cliques and using them to calculate the lower bound on the chromatic number? Write a Lisp program.

6. How to use the algorithm to generate a set of relatively large, but not necessarily maximum, cliques, and not all of them? This problem would have applications to the Encoding Problem in Functional Decomposition. Write a Lisp program.

Chapter 10

Introduction to Automatic Learning in Search Strategies

The ideas of Artificial Intelligence, being ther learning and neural networks, as well as methods mimicking humans while solving problems [761, ?] can be used to program an efficient problem solver.

This chapter proposes such an approach. Since the problems of the constrained logic optimization class, such as NP-hard ones, will be always difficult to solve, one tries to find good heuristics that take into account the peculiarities of real life data in order to maximize the efficiency of execution. This can be done even at the cost of sacrificing the human's understanding why this or other technique works well. When a problem of developing a new algorithm is encountered, the logic theorist/program developer has, based on his experience and a large collection of similar problems, to find an appropriate tree-searching algorithm and the corresponding heuristics. A software system plus the outlined methodology should help the designer in this task.

The program should help the user to choose values for several parameters that all together define the searching strategy These parameters are: depth-first, ordered search, branch-and-bound, and other search heuristic routines.

The developer learns strategies and heuristics in the process, but he cannot test by hand too many examples. He cannot also perform very many experiments using the computer program, because making changes in source code and recompiling takes time. Therefore, we want to automate at least partially this computer-based experimentation process. This way, the program designer will be able to quickly evaluate the usefulness of his various ideas; and particular, how much each of them contributes to the success of the tree search.

Therefore, some of the parameters will also be used to select a combination of the *learning methods.* By the *learning methods* we will mean the methods that change some strategy parameters in order to improve the future behavior of the program, as the result of previous experience of it runs on categories of data.

Two learning methods will be proposed in this chapter. The first method learns criteria of selecting good operators by using a weighted evaluation function and learning its weight coefficients. The same approach is used for selecting nodes as well, where the coefficients of the evaluation function for solution tree nodes are learned. Another variant of this method observes the fact that a search strategy in a program can be described by a set of subroutine flags that are used to connect or disconnect the respective subroutines from the main tree-searching program. The subroutine with a flag taking values $0 \leq p \leq 1$ is connected with a probability of p. The probability coefficients of the flags can be learned in a similar way that the coefficients of the evaluation function are learned. Analogous approaches have been successfully applied in game playing programs and pattern recognizers [37, 59, 84, 197, 286, 334, 358, 424, 473, 474, 476, 534] but to the best of our knowledge, they have never been used to solve the class of combinatorial design problems considered by us here.

The second approach is an original one. It is used in Branch-And-Bound strategies and determines the Stopping Moment - such a moment of search in which backtracking can be terminated, because a better solution is very unlikely to be obtained in future. In several problems a strategy can be constructed that quickly leads to a minimal solution, however, it takes a very long time for the computer to prove that the solution is really the minimal one. Therefore, it is important to know the moment when the further backtracking can be terminated. The method uses a normalized shape diagram to determine the stopping method. The diagram plots the cost function values to the number of subsequently generated solutions. It is assumed, that these shapes are the same for combinatorial data of the same type.

The predictions of the stopping moment are calculated for some estimated probability of not loosing the optimal solution.

Section 10.1 presents the evaluation function learning method and illustrates with the *set covering problem* from section 9.4. Section 10.2 presents the backtracking stopping learning method, and section 10.3 illustrates this approach with a *linear assignment problem.* It has several applications in: VLSI layout, logic synthesis, operations

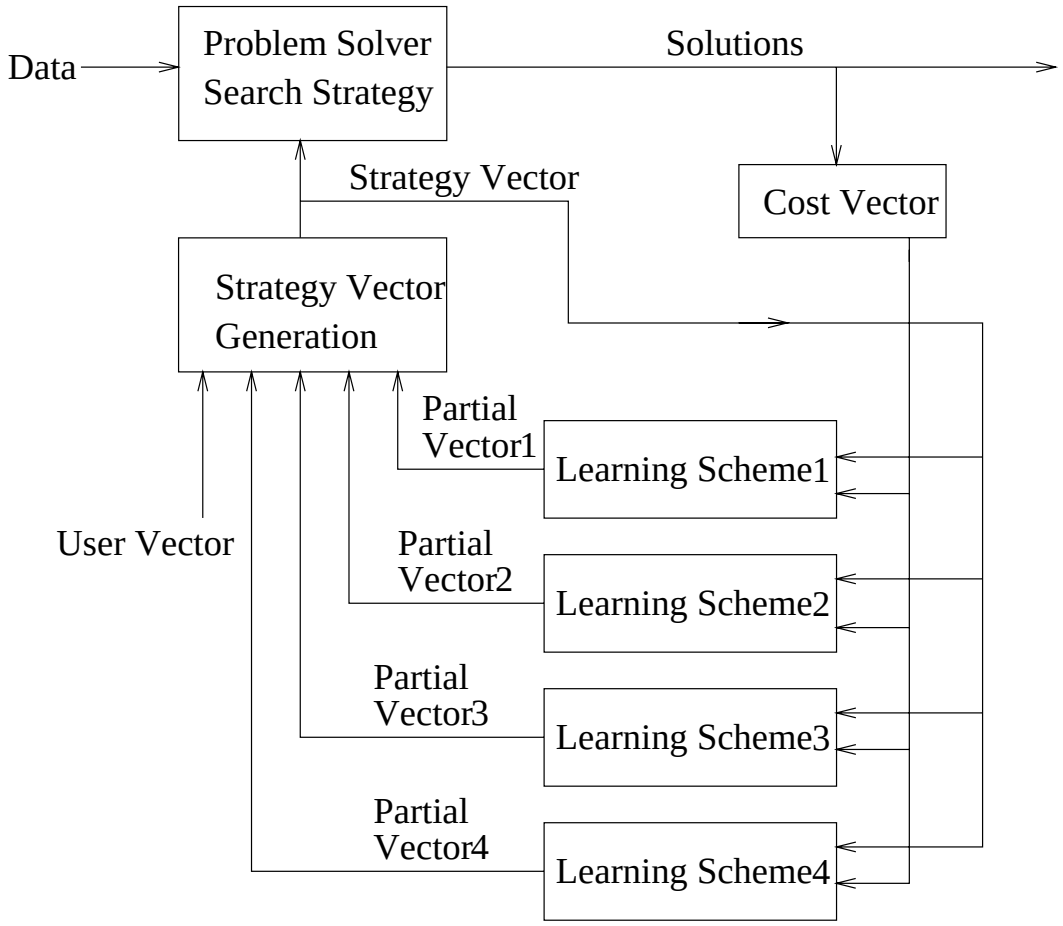


Figure 10.1: Collaboration of the Learning Methods with the Problem Solver

research, production scheduling, marriage counseling, Encoding and Assignment Problems in state machine design and decomposition, economics, communication networks, personnel administration, clustering, psychometrics, and statistical inference [15, 94, 170, 179, 226, 227, 236, 265, 269, 335, 408, 491, ?]. It is closely related to the traveling salesman problem [280]. Interestingly, the same "linear assignment" problem finds also applications to create general-purpose efficient heuristic learning schemes.

The conditions, relations, sorting functions, selecting functions, and strategy parameters, all together describe some "personalized" solution tree searching method and respective strategy. The possibility of dynamic modification of flags and coefficients used in them is *the basic principle of learning in program.* Fig. 10.1 presents schematic diagram of our approach. The Problem Solver in the top left corner is the tree searching one described in the previous sections. Its strategy is specified by a Strategy Vector - an ordered vector of numerical coefficients called **Strategy Parameters** (in general, real numbers, often zeros and ones or numbers from [0, 1]). Each value of the Vector describes one search strategy that can be realized by the Problem-Solver. Assuming r coordinates of the vector, and each of them with w values, one can realize w^r strategies. The sub-vectors of the System Vector (subsets of Strategy Parameters) are created by four Learning Schemes:

1. Learning Scheme # 1 - learning the Operators Evaluation Function.
2. Learning Scheme # 2 - learning the Tree Nodes Evaluation Function.
3. Learning Scheme # 3 - learning the moment when to stop the searching.
4. Learning Scheme # 4 - learning the probabilities of calling various subroutines and using parameters.

Strategy Defining Subroutines.

The user can set the initial values of all Strategy Parameters and some of them cannot be modified by learning. The Strategy Vector Generator checks the consistency of parameters. Certain subroutine flags are *conditionally disjoint.* For instance, there can be three subroutines, $SS1$, $SS2$, and $SS3$, all used to perform the sorting of operators. They use three different methods with different comparison criteria to be used in a node of the tree. Since only one of them

can be used at given time, selecting $SS1$ (flag for $SS1$ enabled) will disable flags for $SS2$ and $SS3$. Let us observe, that the sum of probabilities for parameters $SS1$, $SS2$ and $SS3$ does not have to be equal one, since in any case, selection of one of the subroutine flags will disable the other ones from the group. In general, the *Exclusion Conditions* can be of the form: $Si \wedge Sj$ or $Sk \wedge Sl \wedge Sf \rightarrow \neg Su \wedge \neg Sv$ which means that if flags Si and Sj or flags Sk, Sl and Sf have been selected then the flags Su and Sv must be disabled. The four learning schemes have their local memories that store sets of pairs [*partial vector sample, its cost*], or other similar data. The costs are provided by the calculations of some solution parameters, such as, the values of the cost function, the total solution times, the solution cost improvement times, the sizes of the used memory. All these methods are "orthogonal" (independent) and can be applied separately or together. For strategies #1, 2, and 4, the improper learning, or the lack of the convergence cannot result in non-minimal solutions, but the lack of convergence will cause the program to take more time to obtain the optimal solution.

10.1 The First Method: Learning the Evaluation Function

Several studies lay the stress on the importance of selecting an appropriate Evaluation Functions (Quality Functions) while searching a tree [476, 359]. This section considers only Branch-And-Bound strategies, or mixed strategies, that combine the Branch-And-Bound and Ordered-Search Strategies. This principle is used in Learning Schemes # 1, 2, and 4.

Let the measure of the intelligence of the system be the number of nodes which have to be generated to find the optimal solution. It is easy to observe that this number depends on how quickly the program will encounter the appropriate branch (it means one that terminates with the optimal solution) and this will in turn depend on the good selection of evaluation functions for operators and nodes. For instance, in program the quality function for operators can be defined in a form

$$\sum_{i=0}^{M} c_i \, f_i \;=\; C^T \;\cdot\; F \tag{10.1}$$

where

$$F^T \;=\; (f_1,\; f_2,\; ..., f_n) \tag{10.2}$$

is a vector of *partial quality functions* and

$$C^T \;=\; (c_1,\; c_2,\; ...., c_n) \tag{10.3}$$

is a *vector of weight coefficients.* For each new problem, or even particular set of data for this problem, there is a question of how to select the vector C^T. The program can automatically learn C^T while solving the given problem or a set of examples. The learning problem can be formulated as follows. On the basis of the already searched part of the solution tree, and the information gathered from the previously solved examples of the same set, select such C^T that if a subsequent searches of the same tree were executed, the program would directly extend the branch leading to the best of the solutions selected until now, opening only those nodes, which are on the branch from the initial state of the tree.

Let us denote by N any node on this path, except the solution, and by NN one of its successors on this path. Let $O(N)$ be the operator transforming N to NN on the best branch from all the previously searched branches, and let $O_i(N)$ stands for all other operators in N.

To solve the formulated above problem we need to select such vector C^T that the following set of inequalities be satisfied:

$$(\forall \;\; N)\; [C^T \;\cdot\; F(O(N)) \;>\; C^T \;\cdot\; F(O_i(N))] \tag{10.4}$$

or, after transformation

$$(\exists \;\; N)\; [C^T (F(O(N)) \;-\; F(O_i(N))) \;>\; 0] \;. \tag{10.5}$$

Such set is usually inconsistent, so the problem of learning is reformulated to the problem of selecting C^T which satisfies the greatest possible number of inequalities from the above set.

For illustration, let us consider a two-dimensional case (Fig. 10.2a).

Let us assume that in some node there are 7 operators, for which the quality vectors can be represented as in Fig. 10.2a. Let us also assume that after finding the next solution the vector 4 is best. Using 10.5 we create then the difference vectors $(F(O) \;-\; F(O_i))$. We can observe that no vector C exists in Fig. 3.1b that would satisfy the set of equations 10.5.

Let us, therefore, reformulate the learning problem in the following manner: Select such a vector C for which as many as possible of the inequalities from the set of equations 10.5 are satisfied. As it can be noticed, this problem is equivalent to the problem of calculating such a *hyperplane* running across the coordinates system center, that as many as possible of the differences $(F(O(N)) \;-\; F(O_i(N))$ are located on one side of this hyperplane, and as few as possible on the other side. In the case of n-dimensional space this problem is time consuming to solve, so we have implemented an approximate solution according to [476]. A hyperplane will be considered optimal, when

$$\sum_{i=1}^{n} C^T \;\cdot\; D_i = max, \tag{10.6}$$

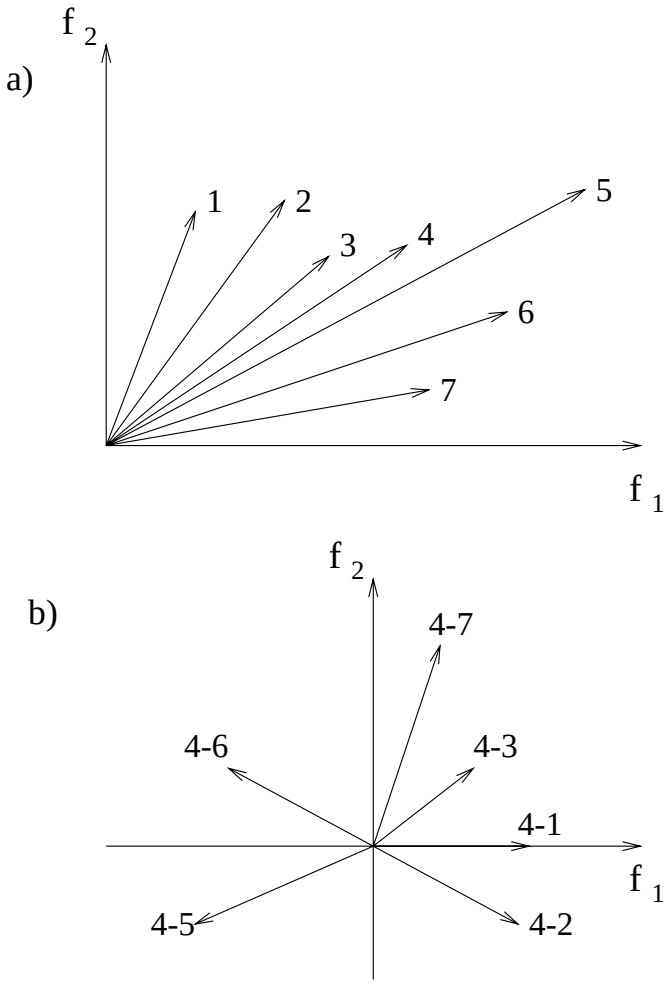


Figure 10.2: Two-Dimensional Case to Illustrate the Learning Method

where $\mid C \mid = 1$ and

$$D_i \;=\; \frac{F(O) \;-\; F(O_i)}{\mid F(O)) \;-\; F(O_i) \mid} \tag{10.7}$$

is a *normalized vector of differences.*
The normalized vector

$$C \;=\; \frac{\sum_{i=1}^{n} D_i}{\mid \sum_{i=1}^{n} D_i \mid} \tag{10.8}$$

is taken as the solution. Whenever new solution is found or when a backtrack occurs, the new inequalities are added to the learning procedures databases and the Learning Schemes #1, 2, and/or 3 are called in order to update the values of respective vectors C, using the above formulas.
An algorithm using the above principles is very fast and gives satisfactory results. Application of this type of learning increases the time (in the learning phase) by about 20% - 30%, but reduces the tree size by about 15%- 20%.

10.1.1 Experimental Results of the Set Covering Problem

This problem was discussed previously. Now we will only discuss the learning aspect of it. It has been found by us in past on another applications, that the application of each of the equivalence conditions, dominance conditions, or indispensable conditions in the covering problem reduces the search space by about 2 to 3 times. The joint application of all the conditions brings about a reduction of approximately 50 to 200 times of the generated space. The influence of learning method on solution efficiency has been investigated. The computation time was increased in the learning phase by 20% to 30%. The vector of coefficients obtained in this learning was: $C = [$ -0.85, 0.25, -0.35]. This vector brings next 15% - 20% decrease in the generated solution space size, as compared with the initial vector: $C = [$ -1, 0, 0].

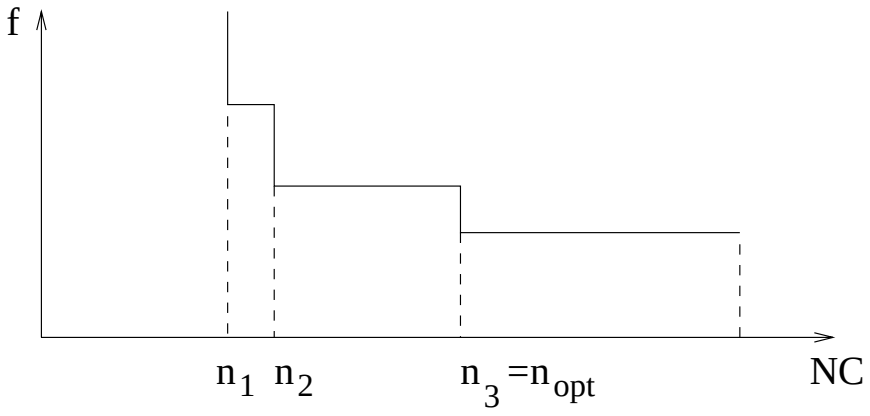


Figure 10.3: Improvement Curve for the Second Learning Method. Dependency of the cost function on the number of expanded nodes

10.2 The Second Method: Learning the Stopping Moment

The stopping process can be explained using an *"improvement curve"* in a diagram where the x-axis is a solution time expressed as a number of expanded nodes and the y-axis is a value of the cost function of the best of the solutions found until then.

An example of an improvement curve is shown in Figure 10.3.
The solution process can be in this case terminated after expanding n_3 nodes, while the further expansion will not bring any cost function improvements. A question can be thus raised *"can one construct a system that would learn to predict the effects (or, specifically, lack of effects) of the further solution space expansion ?"*. A positive answer to this question would additionally improve the solution space method efficiency.

A new approach to this problem will be proposed below. It takes the following assumptions:

1. There exist combinatorial problems for which for all examples from some set of data the improvement curve shapes are similar.

2. There exists a *statistical dependency* (proportionality) between the number of nodes which have to be expanded in order to find successive, better solutions, i.e., $n_1,\ n_2,\ n_3,\ \ldots$. (See Fig. 10.3.).

In order to establish this proportionality, a sufficiently large number of examples must be tested. Also, it is necessary to introduce certain "normalization" that would make possible comparison of various improvement curves.

Let us introduce the concept of the *dimension of the problem with the respect to the i-th improvement,* or *i-th dimension* for short, as the number of nodes that must be expanded in order to obtain the i-th improvement of the cost function value. It will be denoted by n_i. By n_{ij} we will denote the number of nodes to obtain the i-th improvement in the j-th problem's sample. In Fig. 10.3 the dimension with respect to the first improvement is n_1, with respect to the second is n_2, and so on. For the given search strategy, the dimension sequence determines a unique *improvement curve* for this problem. Various problems can have sequences of dimensions of various lengths.

Solution of each example by a computer creates some number of pairs

$$(n_i,\ n_{opt}) \tag{10.9}$$

In the discussed example the pairs $(n_1,\ n_3)$, and $(n_2,\ n_3)$ are created. Each pair is reduced to a standard dimension N, by multiplying both its coordinates by a factor of N/n_i. The following pairs are obtained

$$(N,\ \frac{n_{opt}\ \cdot\ N}{n_i}), \tag{10.10}$$

where N is a fixed coefficient.
The second coordinate of each pair determines the standard number of nodes N_{oi} that results from the i-th dimension and which also has to be expanded in order to find the optimal solution. By disposing several improvement curves for many examples, one can calculate average standard numbers of nodes $\overline{N_{oi}}$ as weighted averages, while the weight of each result N_{oi} grows with the value of the i-th dimension n_i. It is based on the assumption that with an increase of example dimension the results obtained from the example are more reliable, i.e. they are characterized by a smaller expected value of standard deviation. Hence:

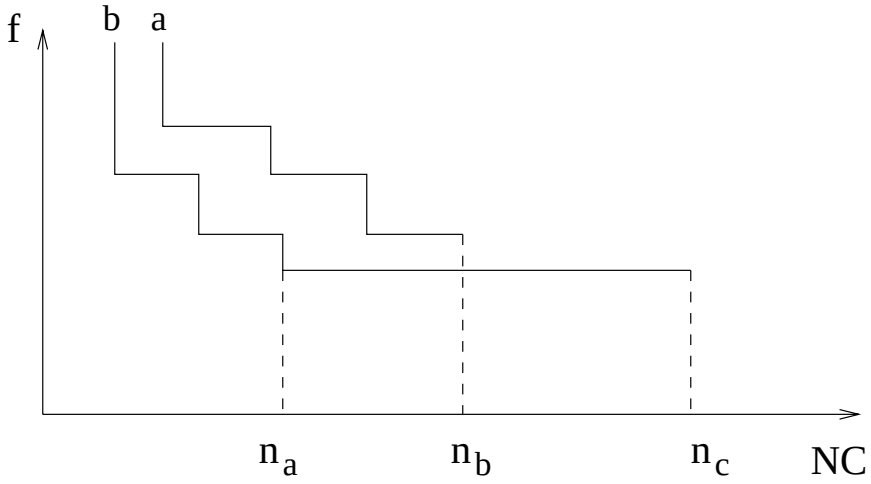


Figure 10.4: The Effect of Concurrent Application of Both Learning Methods, (a) without learning the quality function coefficients, (b) with learning the quality function coefficients.

$$\overline{N_{oi}} \;=\; \frac{\sum\limits_{j=1}^{k_i} N_{oij} \;\cdot\; n_{ij}}{\sum\limits_{j=1}^{k_i} n_{ij}} \;=\; N \;\cdot\; \frac{\sum\limits_{j=1}^{k_i} N_{jopt}}{\sum\limits_{j=1}^{k_i} n_{ij}} \tag{10.11}$$

where k_i is a number of already solved examples that have the i-th dimension. The standard deviations of the obtained results are as follows:

$$\sigma_i \;=\; \sqrt{\frac{\sum\limits_{j=1}^{k_i} n_{ij} \cdot \left(N_{oij} \;-\; \overline{N_{oi}}\right)^2}{\sum\limits_{j=1}^{k_i} n_{ij}}} \tag{10.12}$$

In order to assure a sufficiently high probability that the space expansion would be not terminated before obtaining the optimal solution, a width of the half-interval of confidence with some coefficient is added to the calculated values of $\overline{N_{oi}}$:

$$N_{i\ max} \;=\; \overline{N_{oi}} \;+\; m \;\cdot\; \sigma_i \tag{10.13}$$

Coefficient $m = 3$ assures the 99,7% probability of assuming the normal probability density of the results. Therefore, $N_{i\ max}$ determines the maximal standard number of nodes - calculated with respect to the i-th dimension - which should be expanded in order to find the optimum solution.

The calculated values are applied as follows. Having found the first solution by expanding n_1 nodes, the maximum number of nodes $n_{1_{max}}$ (already not the standard one) is calculated,

$$n_{1max} \;=\; \frac{N_{1max} \;\cdot\; n_1}{N} \tag{10.14}$$

If a better solutions is not found after expansion of n_{1max} nodes, the system stops. However, if after expansion of $n_2 \;<\; n_{1max}$ nodes a better solution is found, then the value of n_{2max} is calculated according to the formula:

$$n_{2\,max} \;=\; \frac{(k_1 \;\cdot\; n_{1\,max} \;+\; k_2 \;\cdot\; \frac{N_{2max} \;\cdot\; n_2}{N})}{(k_1 \;+\; k_2)} \tag{10.15}$$

where k_1 and k_2 determine number of examples which have been used to calculate the values of N_{1max} and N_{2max}. The system behavior after generating next solutions is based on the same principles.

The method described above bypasses several difficulties related to the normalization of several individual properties of the problems, such as different numbers of improvement curve steps, or various values of quality function decrements. It can be observed that this method collaborates well with the one to learn quality functions coefficients, and gives better results when the other method produces better results.

Fig. 10.4 presents schematically the effect that can be obtained by concurrent application of the both learning methods. Without learning, n_c nodes should be expanded. By applying the learning method from this section one would need n_b nodes while applying additionally the method from section 10.3 one would need only n_a nodes.

10.2.1 Example of Application of the Second Method: The Linear Assignment Problem

This problem, similarly to the previous one, has *several* CAD applications, most importantly here, for encoding.

Problem Formulation.

Given is n machines and n workers and the productivity of the worker i on machine j is denoted by w_{ij}. The assignment of machines to workers is sought that maximizes the total productivity of all workers. The assignment matrix $W_{n\times n} = [w_{ij}]$ is given from which n elements must be selected in such a way that any two of them are taken from a different row and column and the sum of the elements is maximum. Since program looks for a minimum, the problem is reformulated as below.

Each object has n properties. The value of property x_i determines the column number from which the element from row i has been selected. In order to simplify the program the elements of the matrix are selected in the following order: first an element from row one is chosen, next an element from row two, and so on. In this case the operator is a number specifying only the second coordinate of the selected element w_{ij}, and the first coordinate is specified by the depth of the node in the tree. Secondly, the list $AS(N)$ of nodes specifying the state of the object in node N becomes in this case unnecessary, while it would contain the set of non-selected rows, and this set is already specified by the depth of the node in the tree. For instance, in such description, the solution

$$< 3, 1, 2 >$$

for a 3×3 problem means that elements $w_{13},\ w_{21},\ w_{32}$ have been selected from matrix W. The description of the **initial state** is as follows:

$QS(0) = \emptyset$,

$GS(0) =$ set of numbers of all columns.

Operation of **operator** $O(N,\ r_i)$ can be described as follows:

$$\begin{bmatrix} QS(NN) = QS(N) \ \cup \ \{\ r_i\ \}, \\ GS(NN) = GS(N) \ \setminus \ \{\ r_i\ \}\ . \end{bmatrix}$$

The **solution condition** is $GS(NN) = \emptyset$.

The specification of the **quality function** values for operators is in this case obvious - they are the negated values of the respective elements w_{ij} of the matrix. The value of cost function for states is equal to the sum of negated costs of operators on the path from the initial state to the given state. **A heuristic quality function for nodes,** that specifies the minimal cost of the path from the given node to the solution is also useful. It is defined as follows:

$$\hat{h} \ = \ \sum_i w_{i,\ min}$$

where summing is extended over the non-selected rows, and $w_{i,\ min}$ denotes the least element of the i-th row among the elements from the non-selected columns.

Experimental Results

The influence of the search strategy on the solution efficiency has been investigated. Fig. 10.5 presents the statistical dependency of the solution time and the number of nodes expanded with respect to the problem's dimension, for various strategies. The Ordered-Search strategy has been the most efficient one for this problem. It is due to the usage of a very accurate heuristic function which permitted for cutting branches at small depth of the tree.

This problem is one for which the Stopping Learning Method gives good results, because it is characterized by a improvement curve with sufficiently large number of steps. (For instance about 4-10 steps for Branch-And-Bound strategy and $n < 14$). Additionally, it was verified that about 30% of the solution tree was expanded "redundantly" after finding the first optimal solution. Using the above method permitted to decrease this part of tree by about 40%. For instance, when the entire space generated by the system included, on average, 600 nodes and the optimal solution was found after 420 nodes, by using this method it was sufficient to expand on 530 nodes.

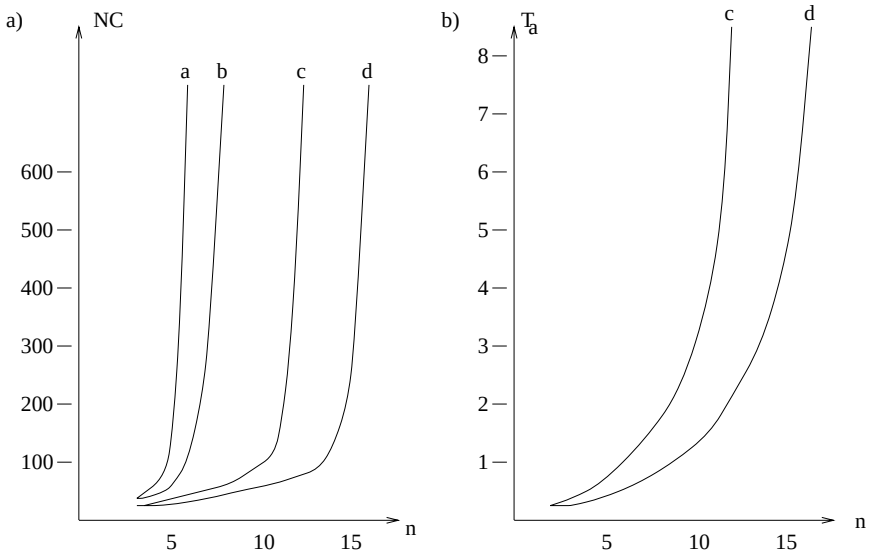


Figure 10.5: The dependence of (a) number of nodes and (b) the processing time on the problem size for strategies: a - Breadth First, b - Depth First, c - Branch-and-Bound, d - Ordered Search

Part III

Universal Spectra and Decision Diagrams with Applications

Chapter 11

Spectral Decision Diagrams

Chapter 12

Linearly Independent Logic

Chapter 13

Universal Spectral Decision Diagrams for Multi-valued Linearly Independent Logic

Chapter 14

Fuzzy and Arithmetic Decision Diagrams

Chapter 15

Word Level Spectral Decision Diagrams

Chapter 16

Haar Transforms and Generalized Wavelet Decision Diagrams

Chapter 17

Applications in Machine Learning

Chapter 18

Applications in Data Mining

Chapter 19

Applications in Decomposition

Chapter 20

Applications in Testing

Chapter 21

Applications in Robotics

Part IV

Anatomy of Inexpensive Robots

% PART 3

Chapter 22

Movement control in Lisp and in C++

Chapter 23

Path Planning and Obstacle Avoidance

23.1 Overview of Path Planning

23.1.1 The visibility graph method

23.1.2 Cell decomposition

23.1.3 Potential Fields

23.1.4 Voronoi Diagrams

23.1.5 Symbolic Representations

23.2 New approach to Path Planning, by Mike Burns

Chapter 24

Mobile Robot Localization

Part V

Robot Case Studies

% PART 4

Chapter 25

Robotics for Handicapped

Chapter 26

PSUBOT 2: An Intelligent Wheelchair with Fuzzy Logic

5.2.2. Wall Tracking.

26.10.4 Coordinating the Behaviors (The Pilot)

26.11 Localization for the PSUBOT

26.11.1 Odometry

26.11.2 Compass

26.11.3 Sonar Matcher

World Model

Sonar System

Matcher Algorithm

9.3.1 Mathematical Fundamentals of Sonar Matcher.

26.12 Custom Demonstration Software in C++ to Show Off the PSUBOT

Chapter 27

KALU: An Animated Ape that reasons in Multiple-Valued Logic

Chapter 28

Learning Reactive State Machines

Chapter 29

Animated Puppet that reasons in Quantum Logic

Glossary

Bibliography

[1] A.V. Aho, and J.D. Ullman, "The Theory of Parsing, Translation and Compiling - vol 1, Parsing," *Prentice Hall Inc.*, Englewood Cliffs NJ 1972.

[2] D.R. Armstrong, "A Programmed Algorithm for Assigning Internal Codes to Sequential Machines," *IRE Transactions Electr. Comp.,* Vol. EC-11, pp. 466-472, August 1962.

[3] D.R. Armstrong, "On the Efficient Assignment of Internal Codes to Sequential Machines," *IRE Transactions on Electronic Computers,* Vol. EC-11, pp. 611-622, October 1962.

[4] M. Abadir, "A Knowledge Based System for Designing Testable VLSI Circuits," *Technical Report* CRI-86-11, 1985, Computer Research Institute, University of Southern California.

[5] H. Afsarmanesh, D. Mcleod, D. Knapp, and A.C. Parker, "An Extensible Object-Oriented Approach to Databases for VLSI/CAD," *Technical Report CRI-85-09,* 1985, Computer Research Institute, University of Southern California.

[6] R. Ahad, and D. Mcleod, "An Approach to Semi-Automatic Physical Database Design and Evolution for Personal Information Systems," *Technical Report CRI-85-11,* 1985, Computer Research Institute, University of Southern California.

[7] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, and J.G. Linders, "Allocation of Multi-Port Memories in Data Path Synthesis," *CH2469-5/87*, pp. 266-269.

[8] M.S. Abadir, M.A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips", IEEE Design& Test, August 1985, pp.56-68.

[9] A.E.A. Almaini, M.A. Moosa, N.M. Aziz, "Computer Aided Design of Groups of Exclusive Logic Functions", Digital Processes, Vol.6, pp. 227-243, 1980.

[10] R.L. Ashenhurst,"The Decomposition of Switching Functions", Proceedings. Int. Symp. on the Theory of Switching, April 1957, pp. 74-116.

[11] H. Abelson, G. Sussman, and J. Sussman, "Structure and Interpretation of Computer Programs," *MIT Press.* (**A textbook used in MIT where Lisp (Scheme) is taught as the first programming language. Many excellent examples.**)

[12] P.W. Abrahams, "Application of LISP to Sequence Prediction," *ACM Symposium on Symbolic and Algebraic Manipulations*, March 1966. (**You can use these ideas to design a sequence predicting machine based on Lisp program**).

[13] G.M. Adelson-Velskiy, V.L. Arlasarov, and M.V. Donskoy, "On the Structure of an Important Class of Exhaustive Problems and on Ways of Search Reduction for Them," Advance papers of the Fourth International Joint Conference on Artificial Intelligence. Section 5, Search Techniques, Tbilisi. Georgia, USSR, September 3-8, 1975. (**This paper presented the well-known AVD search strategy**).

[14] A.V. Aho, and J.D. Ullman, "The Theory of Parsing, Translation and Compiling," Vol.1, Parsing, *Prentice Hall Inc.*, Englewood Cliffs, N.J., 1972.

[15] S.B. Akers, "On the use of linear assignment algorithm in module placement," *25 Years of Electronic Design Automation,* 1988, pp. 218-223.

[16] J. Allen, "Anatomy of Lisp," *Mc.Graw-Hill,* 1978. **This book is very detailed and shows you how to design Lisp interpreter and compiler. For Lisp system programmers and developers.**)

[17] A. Angyal, "The Structure of Wholes," *Philosophy of Science*, 1939, pp. 25-37.

[18] W.R. Ashby, "An Introduction to Cybernetics," *Methuen & Co., London*, 1964.

[19] R. L. Ashenhurst, "The decomposition of switching functions," *Proc. Int. Symp. Theory of Switching, Part I*, Ann. Comput. Lab. Harvard Univ., pp. 74–116, 1959.

[20] Boyer, J., "The Potential of Artificial Intelligence in Aids for the Disabled," *Proc. "Discovery '84: Technology for Disabled Persons,* 1984.

[21] R.E. Bryant,"Symbolic Manipulation of Boolean Functions Using a Graphical Representation", Proc. 22nd Design Automation Conference, pp.688-694, IEEE 1985.

[22] D. Barthel,"A Remark Concerning Minimization of Expressions of Boolean Algebra in the Case of DON'T-CARE Conditions", Elektron. Informationsverarb. Kybern. , Germany, Vol. 19., No.7-8, pp.393-396.

[23] C.R. Baugh,"Generation of Representative Functions of the NPN Equivalence Classes of Unate Boolean Functions", IEEE Trans. on Comp., Vol. C-21, No.12., pp. 1373-1379, December 1972.

[24] J.B. Bendas,"Design through Transformation", Proc. 20th Design Automation Conference, June 1983, pp. 253-256.

[25] P.W. Besslich, "Efficient Computer Method for EXOR Logic Design", IEE Proc. , Vol. 130, Pt.E, No.6, November 1983.

[26] J. Blank, J. Fox, T. Blackman, M. Ciesielski, L. Markov, "The Silc Silicon Compiler", GTE LABORATORIES PROFILE, September 1984.

[27] Y. Breitbart, K. Vairavan,"The Computational Complexity of a Class of Minimization Algorithms for Switching Functions", IEEE Trans. on Comp., Vol. C-20, No.12, December 1979, pp. 941-943.

[28] M.A. Breuer,"Generation of Optimal Code for Expressions via Factorization", Communications of ACM, Vol. 12, No. 6, June 1969.

[29] R.K. Brayton, C.T. McMullen,"The Decomposition and Factorization of Boolean Expressions", Proc. of 1982 ISCAS Symp., Rome, pp.49-54, May 1982.

[30] R.K. Brayton, G.D. Hachtel, L.A. Hemachandra, A.R. Newton, A.L.M. Sangiovanni-Vincentelli, "A Comparison of Logic Minimization Strategies Using ESPRESSO: An APL Program Package for Partitioned Logic Minimization", Proc. of 1982 ISCAS Symp., Rome, pp.42-48, May 1982.

[31] R.K. Brayton, J.D. Cohen, G.D. Hachtel, B.M. Trager, D.Y.Y.Yun, "Fast Recursive Boolean Function Manipulation", Proc. of 1982 ISCAS Symp., Rome, pp.58-62, May 1982.

[32] R.K. Brayton, et al, "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", Proc. 1984 IEEE Cust. Int. Circ. Conf. Rochester, NY., May 1984.

[33] R.K. Brayton, C.T. McMullen, "Synthesis and Optimization of Multistage Logic", Proc. 1984 Int. Conf. on Comp. Des., pp.23-30, Rye, NY ,October 1984.

[34] R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[35] D.Brand,"Redundancy and Don't Cares in Logic Synthesis", IEEE Trans. Computers, Vol.C-32, no.10, pp.947-952, October 1983.

[36] D. Brand, W. Griffin,"Synthesis of Circuit Families with Open Book Set", Proceeding Second Int. Symp. on VLSI Tech. Syst. and Appl. Taipei, Taiwan, May 1985.

[37] A. Bagchi, and A. Mahanti, "Search Algorithms under Different Kinds of Heuristics - A Comparative Study," *JACM,* Vol. 30., 1983, pp. 1-21.

[38] P. Bakowski, J-L. Dubois, and A. Pawlak, "A technique for generating efficient simulators," *Proc. of ICCD'91*, Cambridge, USA, October, 1991, pp. 105-108.

[39] M. Balakrishnan, A.K. Majumdar, D.K. Banerji and J.G. Linders, "Allocation of multi-port memoried in data path synthesis," *IEEE Proceedings,* 1987, pp. 266-269.

[40]

[41] E. Balas, "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," *Operations Research,* Vol. 13, No. 4, pp. 517-546, 1965. (**Many EDA problems can be formulated as problems with zero-one variables. This paper explains how to solve such problems.**)

[42]

[43] E. Balas, "A Note on the Branch and Bound Principle," *Operations Research,* Vol. 16, No. 2, pp. 442-445, 1968. (**The principle of Branch and Bound that we will use in many of our programs, is explained here**).

[44]

[45] M.J. Balinski, and K. Spielberg, "Methods for Integer Programming: Algebraic, Combinatorial and Enumerative," *Progress in Operations Research,* Vol. 3, John Wiley and Sons, New York, 1969. (**A good review of approaches that can be used for many EDA problems**

[46]

[47] R. Banerji, "Theory of Problem Solving. An Approach to Artificial Intelligence," *American Elsevier Publishing Company,* New York, 1969. (**An attempt to formalize various AI approaches. Can be very useful for some classes of problems**).

[48] R.B. Banerji, and G.W. Ernst, "A Comparison of Three Problem-Solving Methods," *Proc. Vth IJCAI,* pp. 442-449, 1977.

[49] D. Barstow, "A Knowledge-Based system for Automatic Program Construction," *Proc. Vth IJCAI*, pp. 382-388, 1977.

[50] D.W. Barron, "Recursive Techniques in Programming." *McDonald*, London, 1969. (**A classic on recursion.**)

[51] M.R. Barbacci, "Instruction Set Processor Specification (ISPS): the Notation and its Applications," *IEEE Trans. on Computers,* Vol. C-30, No. 1, Jan. 1891, pp. 24-39.

[52] J. Batali, E. Goodhue, Ch. Hanson, H.. Shrobe, R.M. Stallman, and G.J. Sussman, "The Scheme-81 Architecture - System and Chip,"' In *Proceedings, Conference on Advanced Research in VLSI,* pp. 69-77, 1982. **This book shows how to design the Lisp computer on a chip. This is directly related to our projects in the sequel.**)

[53] M. Bauer, "A Basis for the Acquisition of Procedures from Protocols," *International Joint Conference on Artificial Intelligence,* IJCAI 1975, pp. 226-231.

[54] E.C. Berkeley, and D.G. Bobrow, (eds.), "The Programming Language LISP: Its Operation and Applications," *Information International, Inc.*, Cambridge, Mass., 1967. **Another classic. This was the first Lisp Manual. Now obsolete, but useful for people interested in history of ideas in computer science.**)

[55] J.B. Bendas, ""Design through Transformation," *Proc. 20th Design Automation Conference,* June 1983, pp. 253-256.

[56] R.G. Bennetts, "Design of Testable Logic Circuits," *Addison-Wesley,* 1984

[57] U.C. Berkeley. "Project Goals: Ultra-Low Power DSP Processing based on Heterogeneous Co-Processors," http://infopad.eecs.berkeley.edu/research/reconfigurable/texts/overview/overview.html.

[58] A. Berlin, and D. Weise, "Compiling scientific code using partial evaluation," *IEEE Computer*, pp. 25-37, 1990. (**Partial evaluation is a classical method for formal semantics and verification**).

[59] H. Berliner, "On the Construction of Evaluation Functions for Large Domains," *Proceedings IJCAI-79,* Tokyo, Japan, 1979, pp. 53-55.

[60] P. Bertolazzi, and A. Sassano, "A class of polynomially solvable set-covering problems," *SIAM J. Discrete Math.,* Vol. 1., No. 3., August 1988, pp. 306-316.

[61] P. Bertolazzi, and A. Sassano, "An O(mn) algorithm for regular set-covering problems," *Theor. Comput. Sci.,* Vol. 54., 2,3, Oct. 1987, pp. 237-247.

[62] Blank, J., Fox, J., Blackman, T., Ciesielski, M., Markov, L.: "The Silc Silicon Compiler", GTE Laboratories Profile , September 1984.

[63] E. Bloedorn and R.S. Michalski, "Data Driven Constructive Induction in AQ17-PRE: A Method and Experiments, *Proceedings of the Third International Conference on Tools for AI*, San Jose, CA, 1991.

[64] D.G. Bobrow, "A Question-Answering System for High-School Algebra Words Problems," *Proc. FJCC,* Vol. 26, pt. 1, pp. 591-614, 1964. (**This paper shows how to develop in Lisp a program that solves problems in High-School Algebra. It can be easily modified to solve puzzles that can be reduced to Boolean equations formulated in English.**)

[65] D.G. Bobrow, "LOOPS: An Object-Oriented Programming System for Interlisp," *Xerox PARC*, 1982. (**One of first papers on object oriented programming in Lisp, which now is a commonplace**).

[66] R.S. Boyer, and J.S. Moore, "Proving Theorems about LISP Functions," *Proc. IJCAI 75,* Session 18. Automatic Programming, pp. 486-493. (**This paper is the first one about so-called Boyer-Moore method that is commonly used for formal verification of software and hardware.**

[67] K.S. Brace, R.L. Rudell and R.E. Bryant, "Efficient Implementation of a BDD Package," *Proc. of 27th Design Automatiom Conference*, pp. 40–45, June, 1990.

[68] I. Bratko, "Prolog Programming for Artificial Intelligence," *Addison-Wesley*, 1990. (**A textbook on Prolog, another main language of AI, by a top world expert. Many of these ideas can be re-written to Lisp, if you have pattern-matching and backtracking implemented in it**).

[69] I. Bratko, "Private communication, University of Ljubljana, 1996.

[70] R. Brayton, G. Hachtel, C. McMullen, and A.L. Sangiovanni-Vincentelli,: "Logic Minimization Algorithms for VLSI Synthesis," *Kluwer Academic Publishers,* 1984.

[71] R. Brayton and F. Somenzi, "An Exact Minimizer for Boolean Relations," *Proc. of ICCAD*, pp. 316–320, 1989.

[72] R. Brayton, P.C. McGeer, J.V. Sanghavi, and A.L. Sangiovanni-Vincentelli, "A New Exact Minimizer for Two-Level Logic Synthesis," *Sasao, T. (ed): "Logic Synthesis and Optimization"*, Kluwer, 1993.

[73] M.A. Breuer, (ed.), "Design Automation of Digital Systems," Vol. 1, *Prentice Hall,* Englewood Cliffs, New Jersey, 1972. .LP

[74] V.M. Briabin, V.A. Serebriakov, and V.M. Yufa,"LORD: LISP-Oriented Resolver and Data Base," *Proc. IJCAI 75,* Section 8. Artificial Intelligence Software, pp. 514-519. **Lisp from former Soviet Union. Integration with a data base**).

[75] R.A. Brooks, "Programming in Common Lisp," *Wiley*, 1985. (**One of first texbooks on Common Lisp from the MIT professor who uses Lisp to program self-learning robots.**)

[76] R. Brooks, "A Model of Human Cognitive Behavior in Writing Code for Computer Programs," *IJCAI, IVth,* pp. 878-884, 1977.

[77] F.D. Brewer, and D.D. Gajski, "An Expert System Paradigm for Design," *Proc. Design Automation Conf.* ACM/IEEE, 1986.

[78] R.E. Bryant, "Symbolic Manipulation of Boolean Functions Using a Graphical Representation," *Proc. 22nd Design Automation Conference,* pp.688-694, IEEE 1985.

[79] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Comput.*, Vol. C-35, No. 8, pp. 667–691, 1986.

[80] R.E. Bryant, "On the complexity of VLSI implementation and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. on Computers*, Vol. 40, pp. 205–213, 1991.

[81] Breitbart, Y., Vairavan, R.: "The Computational Complexity of a Class of Minimization Algorithms for Switching Functions", IEEE Trans. on Comp., Vol. C-20, No. 12, pp. 941-943, December 1979.

[82] M.A. Breuer, (ed.)., "Design Automation of Digital Systems," Vol. 1, *Prentice Hall*, 1972. (**An early good collection of EDA problems with algorithms.**)

[83] B. Buchanan, Sutherland, and E. Feigenbaum, "HEURISTIC DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry," In *"Machine Intelligence 4." Edinburgh University Press*, 1969. (**An early program for learning.**)

[84] B. G. Buchanan, C.R. Johnson, T. M. Mitchell, and R. G. Smith, "Models of Learning Systems," in: Belzer, J. (Ed.), *Encyclopedia of Computer Science and Technology, 11, Marcel Dekker,* New York, 1978, pp. 24-51.

[85] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 1," *IEEE Trans. on Comp.*, Vol. C-18, pp. 1121-1127, December 1969.

[86] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 2," *IEEE Trans. on Comp.* Vol. C-19, pp. 66-73, JAnuary 1970.

[87] J.H. Chang, O.H. Ibarra, M.J. Chung, and K.K. Rao, "Systolic tree implementation of data structures," *IEEE Tran. on Computers,* Vol. 37, No. 6, June 1988, pp. 727-735.

[88] H.A. Curtis, "Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory", JACM, 1963.

[89] E. Cerny, M.A. Marin, "A Computer Algorithm for the Synthesis of Memoryless Logic Circuits", IEEE Trans. on Comp. , Vol. ??, pp. 455-465, May 1974.

[90] W. Cohen, K. Barlett, A.J. deGeus, "Impact of Metarules in a Rule Based Expert System for Gate Level Optimization", Proc. Intern. Symp. on Circuits and Systems, May 1985.

[91] J. Cohoon, S. Sahni, "Heuristics For The Circuit Realization Problem", Proc. 20th Design Automation Conference, IEEE 1983, pp. 560-566.

[92] S.C. Crist,"Synthesis of Combinational Logic Using Decomposition and Probability", IEEE Trans. on Comp., Vol. C-29, No. 11, November 1980, pp. 1013-1016.

[93] H.A. Curtis, "Design of Switching Circuits", Van Nostrand, Princeton N.J., 1962.

[94] B. Carpenter, and IV. N. Davis, "Implementation and performance analysis of parallel assignment algorithms on a hypercube computer," *Proc. "Hypercube Concurrent Computers and Applications,* Vol. 2., Pasadena, CA, Jan. 19-20, 1980, pp. 1231-1235.

[95] K. Cattell, J.C. Muzio, "Analysis of One-Dimensional Linear Hybrid Cellular Automata over GF(q)," *IEEE Tr. Comp.,* Vol. 45, No. 7, July 1996, pp. 782-792.

[96] Champaux, 1975 (**Useful theory of bidirectional search**).

[97] A. Chan, "Using decision trees to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. Comp.,* Vol. C-36, No. 2., Febr. 1987, pp. 212-214.

[98] Ch.L. Chang, "Symbolic Logic and Mechanical Theorem Proving," *Academic Press*, 1973, En 11, 164 C362. (**One of first books on automatic theorem proving, and still very useful**).

[99] E. Charniak, Ch. Riesbeck, D. McDermott, and J. Meehan, "Artificial Intelligence Programming," 2nd edition, *Lawrence Erlbaum*, 1985. (**A classic book showing you the Lisp programming tricks and ideas from top programmers.**)

[100] C. L. Chen, B.W. Curran, "Switching Codes for Delta-I Noise Reduction," *IEEE Tr. Comp.,* Vol. 45, No. 9, Sept. 1996, pp. 1017-1021.

[101] W. Chu, L. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, No. 11, November 1980.

[102] M. J. Ciesielski, S. Yang, and M. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," *Proc. ICCD'89,* pp. 262 - 265, October 1989.

[103] Edward J. McCluskey, "Logic Design Principles – with an emphasis on testable semicustom circuits," *Prentice-Hall*, 1986.

[104] E.G. Coffman, "Computer and Job Scheduling," *J. Wiley and Sons,* New York, 1976.

[105] D. Chyan, and M.A. Breuer, "A Placement Algorithm for Array Processors," *Proc. 20th Design Automation Conference*, pp.182 - 188, June 1983.

[106] M.J. Ciesielski, M.A. Perkowski, S. Yang, "Multiple-Valued Minimization Based on Graph Coloring", report.

[107] J. Cohen, "Constraint Logic Programming Languages," *Communications of ACM*, 33, No. 7, pp. 52-68, 1990. (**An early text on Constraints Programming, a technique very useful in design automation.**)

[108] B. Cohen, "The Mechanical Discovery of Certain Problem Symmetries," *Artificial Intelligence,* Vol. 8, pp. 119-131, February 1977. (**Many EDA problems have various kinds of symmetries.**)

[109] A. Colmerauer, "An Introduction to Prolog III," *Communications of ACM 33,* No.7, pp. 69-90, 1990. (**A successor of Prolog which implements many of ideas that are very useful for projects from this book**).

[110] R.C. Conant, "Detecting Subsystems of a Complex System," *IEEE Transactions on Systems, Man, and Cybernetics*, 1972, pp. 350-353.

[111] R.C. Conant, "Set-Theoretic Structure Modeling," *International Journal of General Systems,* 1981, No. 38, Vol. 7, pp. 93-107.

[112] H. A. Curtis, "A New Approach to the Design of Switching Circuits," *Van Nostrand*, Princeton, 1962.

[113] Deering, M., and C.Collins, "Real-time natural scene analysis for a blind prosthesis," *Proc. 7th Int. Joint Conf. Artificial Intelligence,* pp. 704-709, 1981.

[114] M.R. Dagenais, V.K. Agarwal, N.C. Rumin, "The McBoole Logic Minimizer", Proc. 22nd Design Automation Conference, IEEE 1985, pp. 667-673.

[115] J. Darringer, W.H. Joyner, Jr., "A New Look at Logic Synthesis", Proceedings of 17th DAC, Minneapolis, pp.543-549 , 1980.

[116] J. Darringer, J.W. Joyner, C. Berman, L. Trevillyan "Experiments in Logic Synthesis", Proc. IEEE Intern. Conf. on Circuits and Computers ICCC 80, pp.234-7A, 1980.

[117] J.A. Darringer, J.W.Joyner, C.Berman, L.Trevillyan, "Logic Syntesis Through Local Transformations", IBM J. of Res. and Devel. Vol. 25, no.4., July 1981.

[118] J.A. Darringer et al., "LSS: A System for Production Logic Synthesis", IBM J. of Res. and Dev. vol 128, no.5, pp. 537-545, Sept. 1984.

[119] E.S. Davidson,"An Algorithm for NAND Decomposition Under Network Constraints", IEEE Trans. on Comp., Vol. C-18, pp. 1098-1109, 1969.

[120] A.J. deGeus, W. Cohen, "A Rule-Based System for Optimizing Combinational Logic", IEEE Design and Test, August 1985, pp.22-32.

[121] D.L. Dietmeyer, P.R. Schneider, "A Computer-Oriented Factoring Algorithm for NOR Logic Design", IEEE Trans. on Electr. Comp., Vol. EC-14, pp. 868-874, 1965.

[122] D.L. Dietmeyer, P.R. Schneider, "Identification of Symmetry, Redundancy and Equivalence of Boolean Functions", IEEE Trans. on Electr. Comp. , Vol. EC-16, pp. 804-817, December 1967.

[123] D.L. Dietmeyer, Y.H. Su, "Logic Design Automation of Fan-In Limited NAND Circuits", IEEE Trans. on Comp., Vol. C-18, pp. 11-22, 1969.

[124] D.L. Dietmeyer, "Logic Design of Digital Systems". Boston, Mass: Allyn and Bacon,1971.

[125] J.P. Deschamps, "Binary Simple Decomposition of Discrete Functions", Digital Processes, Vol. 1, pp. 123-140, 1975.

[126] J. Dussault, C.C. Liaw, M. Tong.," A High Level Synthesis Tool for MOS Chip Design". Proc. 2nd Design Automation Conf., Albuquerque, June 1984.

[127] M. Damiani, "Nondeterministic Finite-State Machines and Sequential Don't Cares," *Proc. European Design Automation Conf.*, Paris, France, pp. 192-198, 1994.

[128] A.L. Davis, and R.M. Keller, "Data Flow Program Graphs," *IEEE Computer,* February 1982, pp. 26-41.

[129]

[130] R. Davis, and D. Thomas, "Geometric Arithmetic Parallel Processor," *NCR,* 1984, preprint.

[131] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *Comm. ACM,* Vol. 5, pp. 394-397, 1962

[132] R. Davis, "Interactive Transfer of Expertise: Acquisition of New Inference Rules," *Proc. Vth IJCAI,* pp. 321-328, 1977.

[133] M. DesJardin and D. F. Gordon, "Evaluation and Selection of Biases in Machine Learning," *Machine Learning Journal*, Vol. 20, pp. 5–21, 1995.

[134] J. deKleer, J., and G.J. Sussman, "Propagation of Constraints Applied to Circuit Synthesis," *Proc. IVth IJCAI,* September 1978. (**One of very early AI for circuit design papers. Although written about analog design, these ideas can be used also for digital circuits.**)

[135] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon*, Boston, 1971.

[136] D.L. Dietmeyer, and P.R. Schneider, "Identification of Symmetry, Redundancy and Equivalence of Boolean Functions," IEEE TEC, Vol. EC-16, pp. 804-817, December 1967. (**One way of using symmetry in logic design.**)

[137] G. Drastal, G. Czako and S. Raatz, "Induction in an Abstraction Space: A Form of Constructive Induction," *Proceedings of the IJCAI-89*, pp. 708–712, Morgan Kaufmann, Detroit, MI, 1989.

[138] D. Drusinsky, and D. Harel " State Charts as an Abstract Model for Digital Control Units," *Dept. Applied Math. and Comp. Sci., The Weizmann Institute of Science,* Rehovot, Israel, 1986.

[139] P. Dysko, "Implementation of Multipurpose, Multistrategic, Combinatorial Problem Solver in FORTRAN," *M.Sc. Inst. Autom. Control,* Warsaw Technical Univ. 1978 (in Polish).

[140] C.K. Erdelyi, W.R. Griffin, R.D. Kilmoyer, "Cascode Voltage Switch Design", VLSI Design, pp. 78-86, October 1984.

[141] J. Edmonds, and R. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *JACM,* Vol. 19, No. 2, pp. 248-264, 1972.

[142] W. Emde and C.U. Habel and C.R. Rollinger, "The Discovery of the Equator or Concept Driven Learning," *Proceedings of IJCAI-83*, pp. 455–458, Karlsruhe, Germany, Morgan Kaufmann, 1983.

[143] P. Emanuelson, and A. Haraldsson, "On Compiling Embedded Languages in Lisp," *Lisp Conference*, Standford, CA, pp. 208-215, 1980. (**This paper shows how a language can be defined using Lisp and its compiler developed in Lisp.**)

[144] C. Engelman, "MATHLAB - A Program for On-line Machine Assistance in Symbolic Computations," *Proc. FJCC,* Vol. 27, pp. 413-422, November 1965. (**This early paper has a very good explanation how to build an interactive program for symbolic computations.**)

[145] W. Emde and C.U. Habel and C.R. Rollinger, "The Discovery of the Equator or Concept Driven Learning," *Proceedings of IJCAI-83*, pp. 455–458, Karlsruhe, Germany, Morgan Kaufmann, 1983.

[146] T.G. Evans, "A Heuristic Program to Solve Geometric-Analogy Problems," *Proc. SJCC*, Vol. 25, pp. 327-338, April 1964. (**Another classic in AI. A Program that solves problems based on analogies. Demonstrates how analogical reasoning can be implemented in Lisp.**)

[147] B. Evans, and D. Fisher, "Overcoming process delays with decision-tree induction," IEEE Expert, 1994, No.9, pp. 60–66,

[148] G.F. Fritsnovich, "Synthesis of a Microinstruction Decoder Using Programmable Logic Arrays," *Avtomatika i Vychislitel'naya Tekhnika,* Vol. 15, no.2, pp.45-53, 1981.

[149] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers,* Vol. C-30, No. 7, July 1981.

[150] C.M. Fiduccia, R.M. Mattheyeses, "A Linear-Time Heuristic for Improving Network Partitions", 19th Design Automation Confrerence Proc., IEEE 1983, pp.175-181.

[151] B.C. Falkenhainer and R.S. Michalski, "Integrating Quantitative and Qualitative Discovery in the ABACUS System," Machine Learning: An Artificial Inteligence Approach, Vol. 3, Ed. Y. Kodratoff and R.S. Michalski, *Morgan Kaufmann*, Palo Alto, CA, 1990.

[152] A. Farley, "Constructive Visual Imaginery and Perception", *Proc. IJCAI 75*, pp. 885-892.

[153] W. Faught, "Affect as Motivation for Cognitive and Conative Processes," *IJCAI 75,* pp. 893-899

[154] U.M. Fayyad, P. Smyth, N. Weir and S. Djorgovski, "Automated analysis and exploration of image databases: Results, progress, and challenges," *Journal of Intelligent Information Systems*, 1993, No. 4, pp. 1–19,

[155] C.M. Fiduccia, and R.M. Mattheyeses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conference Proc.*, IEEE 1983, pp. 175-181.

[156] R.E. Fikes, "Deductive Retrieval Mechanisms for State Description Models," Proceedings of Fourth International Conference on AI, pp. 99-106, September 1975.

[157] R. Fikes, and N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Proc. 2nd. Joint Intern. Conf. on AI*, pp. 608-620, London, 1977. (**Shows a Lisp-based system to control robots, that found next many other applications.**)

[158] =

[159] C. Files, "Using a Search Heuristic in an NP-Complete Problem in Ashenhurst-Curtis Decomposition," *Graduate Summer Research Program,* Wright Laboratory, August, 1994.

[160] C. Files, R. Drechsler and M. Perkowski, "Functional Decomposition of MVL Functions using Multi-Valued Decision Diagrams," *Proc. of ISMVL'97*, pp. 27-32, Halifax, Nova Scotia, Canada, May 28-30, 1997.

[161] N.V. Findler, and B. Meltzer, (eds.)., "Artificial Intelligence and Heuristic Programming," *Edinburgh University Press*, 1971. (**Interesting examples of heuristic programming.**)

[162] R. Finkel, and U. Manber, "DIB - A Distributed Implementation of Backtracking," *Proc. of the International Conference on Distributed Computing Systems,* Denver, May 1985

[163] N.S. Flann, "Improving Problem Solving Performance by Example Guided Reformulation of Knowledge," *Change of Representation and Inductive Bias*, D.P. Benjamin, Kluwer Academic, Boston, MA, 1990.

[164] B.C. Falkenhainer and R.S. Michalski, "Integrating Quantitative and Qualitative Discovery in the ABACUS System," Machine Learning: An Artificial Inteligence Approach, Vol. 3, Ed. Y. Kodratoff and R.S. Michalski, *Morgan Kaufmann*, Palo Alto, CA, 1990.

[165] J.M. Foster,"List Processing," *Elsevier*, London, 1969. (**Another early classical text in list processing.**)

[166] Franz, Inc., "Common Lisp: The Reference," *Addison-Wesley*, 1988. (**Official Reference book from one of top Lisp companies. Look for new editions.**)

[167] J. Franco, "On the Probabilistic Performance of Algorithms for the Satisfiability Problem," *Technical Report No. 167,* March 1985, Indiana University Computer Science Department.

[168] D.P. Friedman, "The Little Lisper," *Science Research Associated, Inc.*, Palo Alto, Calif., 1974. (**This is an excellent interactive primer for a useful subset of Lisp. There is a new edition available.**)

[169] R. Freivald, "Probabilistic Machines Can Use Less Running Time," *Proc. IFIP 77*, North-Holland, Amsterdam, 1977, pp. 839-842.

[170] J. Frenk, M. Van Houweninge, and R. Kan, "Order statistics and the linear assignment problem", *Computing,* N.Y., Vol. 39, April 1, 1987, pp. 165-174.

[171] S. Ginsburg, "A Synthesis Technique for Minimal State Sequential Machines," *IRE Trans. Electron. Computers,* Vol.EC-8, no. 1, pp. 13-24, March, 1959.

[172] S. Ginsburg, "On the Reduction of Superfluous States in a Sequential Machine," *J. Assoc. Computing Machinery,* Vol. 6, pp. 259-282, April, 1959.

[173] R. Goering, "Silicon compilers bridge gap between concepts and silicon," *Computer Design,* Nov. 1987.

[174] K. Garrison, D. Gregory, W. Cohen, A. deGeus, "Automatic Area and Performance Optimization of Combinatorial Logic", ICCAD 1984, pp.212-214, 1984.

[175] J.L. Gilkinson, S.D. Lewis, B.B. Winter, A. Hekmatpour, "Automated Technology Mapping", IBM J. Res. Develop. Vol. 28, no.5, September 1984.

[176] D. Gregory, K. Bartlett, A.J. deGeus, "Automatic Generation of Combinatorial Logic from a Functional Specification", Proc. IEEE Int. Symp. Circuits and Systems, May 1984, pp. 986-989.

[177] M.R. Garey, and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness," *W.H. Freeeman and Company,* San Francisco 1979.

[178] W.R. Garner, "The Processing of Information and Structure." *Halstead Press*, NY, 1974.

[179] R. S. Garfinkel, and G.L. Nemhauser, *Integer Programming,* Wiley, N.Y., 1972.

[180] W.R. Garner, and W.J. McGill, "The Relation Between Information and Variance Analyses," *Psychometrica,* 1956, Vol. 21, No. 3., pp. 219-228.

[181] J. Gaschnig, "Exactly How Good Are Heuristics? Toward a Realistic Predictive Theory of Best-First Search," *Proc. Vth IJCAI,* pp. 434-441, 1977.

[182] Grasselli, 1966.

[183] R. Gerritsen, "The Application of Artificial Intelligence to Data Base Management," *Proc. IJCAI 75*, pp. 521-527. **(Shows techniques that can be used to build a CAD data base with intelligent processing.)**

[184] (GOSH?) A. Ghosh, S. Devadas, and A. Newton, "Verification of Interacting Sequential Circuits," *Proceedings of the 27th A.C.M./I.E.E.E. Design Automation Conference,* pp. 213-219, 1990.

[185] J. Gilbert and L. Gilbert, "Elements of Modern Algebra," *Prindle, Weber & Schmidt*, Boston, 1984.

[186] A. Giordana, R. Neri and L. Saitta, "Fielded Applications of Machine Learning," San Francisco CA: Morgan Kaufmann, P. Langley and Y. Kodratoff, 1997,

[187] Gold Hill/Intel "Concurrent Common Lisp," Data sheet, order number CC 0186-01, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton , Oregon 97006, May 1986.

[188] A. Goldberg, P. Purdom, and C. Brown, "Average Time Analyses of Simplified Davis-Putnam Procedures," *Information Processing Letters,* Vol. 15, No. 2, pp. 72-75, Sept. 6, 1982.

[189] I.P. Goldstein, and E. Grimson, "Annotated Production Systems. A Model for Skill Acquisition," *Proc. Vth IJCAI,* pp. 311-317, 1977.

[190] Lawrence H. Goldstein and Evelyn L. Thigpen, "SCOAP: Sandia controllability/observability analysis program," *Proceedings of the 17th Design Automation Conference,* Jun. 1980, pp. 190-196.

[191] M. Goodman, "Fido: The shopping doggie!," 1996,

[192] W. Golumb, "Shift Register Sequences," Revised Edition, *Aegean Park Press, Laguna Hills, Calif., 1982.*

[193] G.H. Greene, "The Abacus 2 System for Quantitative Discovery: Using Dependencies to Discover Non-Linear Terms," Reports of Machine Learning and Inference Laboratory, MLI 88-4, *Center for Artificial Intelligence, George Mason University, Fairfax, VA, 1988.*

[194] Graspe functions.

[195] G. Green, and D. Barstow, "Some Rules for the Automatic Synthesis of Programs," Proc. IJCAI 75, *pp. 232-239.* **(Shows techniques that can be used to synthesize Lisp programs automatically.)**

[196] E.M. Greenawalt, J. Slocum, and R.A. Amsler, "UT LISP . UT LISP Documentation. Version 4.0," Computation Center, *Univ. of Texas at Austin, May 1975.* **(Lisp with three pointers. Now obsolete.)**

[197] A. K. Griffith, "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers," Artificial Intelligence, *Vol. 5., 1974, pp. 137-148.*

[198] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba and L. Jozwiak, "Cube Diagram Bundles: a new representation of strongly unspecified multiple-valued functions and relations," Proc. of ISMVL'97, *Halifax, Nova Scotia, Canada, pp. 287–292, May 28-30, 1997.*

[199] S. Grygiel, M. Perkowski, P. Burkey and M. Burns, "New representation for Storing and Decomposition of MV Relations," Design Automation Conference, unublished, *1997.*

[200] S. Grygiel and M. Perkowski, "Multiple-Valued Incompletely Specified Functions and Relations. Part I: Representation," 1997.

[201] S. Grygiel and M. Perkowski, "New compact representation of multiple-valued functions, relations, and non-deterministic state machines," Proc. ICCD-98, *Austin, Texas, October, 1998.*

[202] C. Guilfoyle, "Ten minutes to lay the foundations," Expert Systems User, *August, 1986, pp. 16–19,*

[203] J.E. Hopcroft, "An nlogn algorithm for minimizing states in a finite Automaton," In Kohavi, Z and Paz, A(eds).,Theory of Machines and Computations, Academic Press, *New York, pp. 189-196, 1971.*

[204] M. Hirayama, "A Silicon Compiler System Based on Asynchronous Architecture," IEEE Trans. on Computer Aided Design, *Vol. CAD-6, No.3. May 1987.*

[205] P. Hou, R.M. Owens, and M.J. Irwin, "DECOMPOSER: A Synthesizer for Systolic Systems," Proc. of Design Automation Conference, *ACM/IEEE, 1988.*

[206] M. Hoffman, R. Newton, "A Synthesis System for CMOS Domino Logic", Proc. 1985 Int. Symp. on Circ. and Syst. Kyoto, Japan, June 1985.

[207] T. Hoshino, M. Endo, O. Karatsu, "An Automatic Logic Synthesizer for Integrated VLSI Design System", Proc. 1984 Cust. Int. Circ. Conf., pp. 356-360, Rochester, NY, May 1984.

[208] G.D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby, " Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines," European Design Automation Conference, *Amsterdam, The Netherlands, pp. 184-191, 1991.*

[209] P.L. Hammer, and S. Rudeanu, "Pseudo-Boolean Programming," Operations Research, *Vol. 17, No. 2, 1969.*

[210] M. Harrison, "Introduction to Switching and Automata Theory," McGraw-Hill, *New York, 1965.*

[211] J. Hartmanis, and R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," Prentice-Hall, *New York, 1966.*

[212] S. Hardy, "Synthesis of LISP functions from Examples," Proc. IJCAI 75, *pp. 240-245. (***Title is self-explanatory. Very interesting paper.***)*

[213] T.P. Hart, "Macro Definitions for LISP," AI Project, R.L.E. and M.I.T. Computation Center, *Memo 57, 1963. (***One of early papers on macro-definitions in Lisp. Only of historical value.***)*

[214] R. Hart, and E.B. Koffman, "A Student-Oriented Natural Language Environment for Learning LISP," Proc. IJCAI 75, *pp. 391-396. (***Good ideas for building natural language interface to a system in Lisp. Can be used to build WWW interface for your CAD programs.***)*

[215] F. Hayes-Roth, and J. McDermott, "Knowledge Acquisition from Structural Descriptions," Proc. Vth IJCAI, *pp. 356-362, 1977.*

[216] A.C. Hearn, "Computation of Algebraic Properties of Elementary Particle Reactions Using a Digital Computer," CACM, *August 1966. (***Shows applications of Lisp in physics. It is just one of many excellent papers on using symbol manipulation in sciences***).*

[217] =

[218] F.C. Hennie, "Finite-State Models for Logical Machines," John Wiley, New York, *1968.*

[219] C. Hewitt, "Description and Theoretical Analysis of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," AI Report TR-258, *MIT AI Lab., Cambridge, Mass., 1972. (***One more classic of AI. Demonstrates how you can build a language on top of Lisp that proves theorems, controls a robotic hand and communicates about this in plain subset of English***).*

[220] =

[221] F.J. Hill, and G.R. Peterson, "Introduction to Switching Theory and Logical Design," John Wiley & Sons, Inc., *2nd Ed., New York, 1974.*

[222] D. S. Hochbaum, "Approximation algorithms for the weighted set covering and node covering problems," SIAM J. Comput., *Vol. 11, 1982, pp. 535-556.*

[223] S. Hoelldobler, "Foundations of Equational Logic Programming," Springer-Verlag Lecture Notes in Artificial Intelligence, *1987. (***One of first papers on Equational Logic Programming, a technique that may be very useful for EDA problems, but not much is published on this topic.***)*

[224] J.E. Hopcroft, and D. Ullman, "Formal Languages and Their Relation to Automata," Addison-Wesley Publ. Company, *Reading, Mass., 1969.*

[225] J.E. Hopcroft, "An nlogn Algorithm for Minimizing States in a Finite Automaton," In Kohavi,Z. and Paz, A. (eds.), Theory of Machines and Computations, Academic Press, *New York, pp. 189-196, 1971.*

[226] L. J. Hubert, "Assignment Methods in Combinatorial Data Analysis," Marcel Dekker Inc., *New York/Basel, 1987.*

[227] L. J. Hubert, "Statistical applications of linear assignment," Psychometrica, *Vol. 49, 1984, pp. 449-473.*

[228] S.L. Hurst, D.M. Miller, and J.C. Muzio, "Spectral Techniques in Digital Logic," Academic Press, *London and New York, 1985.*

[229] S. L. Hurst, The Logical Processing of Digital Signals. *Crane-Russak, New York and Edward Arnold, London, 1978.*

[230] T. Ibaraki, and N. Katoh, "Resource Allocation Problems: Algorithmic Approaches," MIT Press, *Cambridge MA, 1988.*

[231] Ichikawa, Y., Ozaki, N., and K. Sadakane, "A hybrid locomotion vehicle for a nuclear power plant," IEEE Trans. Syst. Man, Cybern., *Vol. SMC-13, No. 6., pp. 1089-1093, Nov. 1983.*

[232] T. Ibaraki, "Theoretial comparisons of search strategies in branch-and-bound algorithms," Intern. Journ. Comp. Sci., *5, 1976, pp. 315-344.*

[233] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms," International Journal of Computer and Information Sciences, *Vol. 5, No. 4, pp. 315-344, 1976. (***Useful if you want to compare quantitatively your new branch-and-bound algorithm to existing branch-and-bound algorithms.***)*

[234] H. Jiang, J. C. Majithia, "Suggestion for a New Representation for Binary Function," IEEE Tr. Comp., *pp. 1445-1449, Volume 45, Number 12, December 1996.*

[235] D. Johnson, "The NP-Completeness Column: An Ongoing Guide," Journal of Algorithms, *Academic Press, each issue.*

[236] R. Jonker, and Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," Computing, *N.Y., Vol. 38., No. 4., March 1987, pp. 325-340.*

[237] Kimbler, D.L., "Robots and Special Education: The Robot as Extension of Self," Peabody Journal of Education, *vol. 62, No.1., p.67-76, 1984.*

[238] R.M. Karp,"Functional Decomposition and Switching Circuit Design", J.SIAM, 1963.

[239] B.W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", Bell Syst. Tech. J. Vol.??, Feb. 1970, pp.291-307.

[240] J.H. Kim, D.P. Siewiorek, : "Issues in IC Implementation of High Level, Abstract Designs", Proceedings of 17th DAC, Minneapolis, pp.85-91, 1980.

[241] S. Kirkpatrick, C.D. Gelatt Jr, M.P. Vecchi, "Optimizaton by Simulated Annealing", Science, Vol. 220, no. 4598, 13 May , pp. 671-680.

[242] G.J. Klir, "Introduction to the Methodology of Switching Circuits", D. Van Nostrand Co., New York , 1972.

[243] T.J. Kowalski, W.H. Geiger, W.H. Wolf, W. Fichtner, "The VLSI Design Automation Assistant: From Algorithms to Silicon", IEEE Design & Test, August 1985, pp. 33-43.

[244] R.H. Krambeck, C.M. Lee, H.S. Law, "High-Speed Compact Circuits with CMOS", IEEE Journal of Solid-State Circuits SC-17, No. 3., pp. 614-619, June 1982.

[245] N. Karba, and R. Drole, "Expert systems for the cold rolling mill of the Steel Works Jesenice," In Proceedings of the Thirteenth Symposium on Information Technologies, *Sarajevo, Yugoslavia: Unpublished, 1989.*

[246] =

*[247] Karp, R.M.: "Reducibility Among Combinatorial Problems". Complexity of Computer Computation, Plenum Press, ed. Miller, pp. 85-103, New York, 1972. (***One of most important early papers in complexity theory***).*

*[248] A. Kaufman, "Introduction a la Combinatorique en vue des Applications," Dunod, Paris, 1968. (***A source of excellent project ideas, unfortunately in French***.)*

[249] Kerntopf, P., Michalski, A.: "Selected Problems in Synthesis of Combinatorial Logic Circuits". PWN, Warsaw, 1972 (in Polish).

[250] P. Kerntopf, and A. Michalski, "Selected Problems in Synthesis of Combinatorial Logic Circuits," PWN, *Warsaw, 1972 (in Polish).*

[251] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi, "Optimizaton by Simulated Annealing," Science, *Vol. 220, no. 4598, 13 May, pp.671-680.*

[252] R.E. Kling, "A Paradigm on Reasoning by Analogy," Proc. IJCAI 77. *(***One more classic on programming reasoning by analogy in Lisp. It can find applications in EDA applications.***)*

[253] G.J. Klir, "Introduction to the Methodology of Switching Circuits," D. Van Nostrand Co., *New York , 1972.*

[254] D. Knoke, and P.J. Burke, "Log-Linear Models," Sage Publications, *Beverly Hills, California, 1980.*

[255] G.J. Klir, "Architecture of Systems Problem Solving," Plenum Press, *New York, 1985.*

[256] Z. Kohavi, "Secondary State Assignment for Sequential Machines," IEEE Trans. on Elect. Comp., *pp. 193-203, June 1964.*

[257] Z. Kohavi, "Switching and Finite Automata Theory," Mc Graw-Hill, *1970.*

[258] W.H. Kohler, and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," JACM, *Vol. 21, pp. 140-156, 1974.*

[259] M.M. Kokar, "Discovering Functional Formulas Through Changing Representation Base," Proceedings of the AAAI-86, *pp. 455–459, Philadelphia, PA, 1986.*

[260] C.A. Knoblock, "A Theory of Abstraction for Hierarchical Planning," Change of Representation and Inductive Bias, Ed. D.P. Benjamin, Kluwer Academic, Boston, MA, 1990.

[261] C.A. Knoblock, S. Minton and O. Etzioni, "Integrating Abstraction and Explanation Based Learning in PRODIGY," Proceedings of AAAI-91, *pp. 541–546, AAAI Press/MIT Press, 1991.*

[262] R. Kohavi, "Bottom-up Induction of Oblivious Read-Once Decision Graphs," Proc. of ECML-94, *1994.*

[263] R. Kohavi, "Bottom-up induction of oblivious read-once decision graphs," Europeean Conference on Machine Learning, *1994.*

[264] W.H. Kohler, K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," JACM, *Vol. 21, pp. 140-156, 1974. (***Some of EDA problems are permutation problems. Here is the source of ideas.***)*

[265] T. C. Koopmans, and M. Beckmann, "Assignment problems and the location of economic activities," Econometrica, *25., pp. 53-76, 1957.*

[266] T. Koschman, "The Common Lisp Companion," Wiley, *1990. (***Good textbook.***)*

[267] K. Krippendorf, "On the Identification of Structures in Multivariate Data by the Spectral Analysis of Relations," Proc. of the 23rd Annual Meeting of the Society for General Systems Research, *1979, pp. 82-91, Louisville, Kentucky.*

[268] K. Krippendorf, "Information Theory: Structural Models for Qualitative Data," Sage Publications, *Beverly Hills, California, 1986.*

[269] H. W. Kuhn, "The hungarian method for the assignment problem," Naval Res. Logist. Quart., *2., pp. 83-87, 1955.*

[270] E.E. Lange, "Lower Bound For The Number of Terms in The System of DNF That Describes The Logical Structure of An Automaton," Avtomatika i Vychislitel'naya Tekhnika, *Vol. 15, no. 4, pp. 27-32, 1981.*

[271] D. Lewin, "Computer-Aided Design of Digital Systems", Crane Russak, New York, 1977.

[272] G.W. Leive, D.E. Thomas, "A Technology Relative Logic Synthesis and Module Selection System", Proc. 18th Design Automation Conf., June 1981, pp. 479-485.

[273] E.L. Lawler, "An Approach to Multilevel Minimization", JACM, Vol. 11, No. 3, pp.283-295, July 1964.

[274] H.P. Lee, E.S. Davidson, "A Transform for NAND Network Design", IEEE Trans. on Comp. , Vol. C-21, N0.1, pp.12-20, January 1972.

[275] P. Langley, H.A. Simon, G.L. Bradshaw and J.M. Zytkow, "Scientific Discovery: Computational Explorations of the Creative Process," MIT Press, *Cambridge, MA, 1987.*

[276] P. Langley, G.L. Bradshaw and H.A. Simon, "Rediscovering Chemistry with the BACON System," Machine Learning: An Artificial Inteligence Approach, *Ed. R.S. Michalski and J.G. Carbonell and T.M Mitchell,* Morgan Kaufmann, *Los Altos, CA, 1983.*

[277] P. Langley, H.A. Simon, G.L. Bradshaw and J.M. Zytkow, "Scientific Discovery: Computational Explorations of the Creative Process," MIT Press, *Cambridge, MA, 1987.*

[278] =

[279] E.L. Lawler, and D.E. Wood, "Branch-and-Bound Methods. A Survey," Operators Research, *Vol. 14, No. 4, pp. 699-719, 1966. (***A very good early survey of Branch-and-Bound methods. Read it!***).*

[280] E. L. Lawler, J. K. Lenstra, A.H.G. Rinooy Kan, and D.B., Shmoys, "The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization," Wiley, *1985.*

[281] J.B. Larson and R.S. Michalski, "Inductive Inference of VL Decision Rules," ACM SIGART Newsletter, *No. 63, pp. 38–44, 1977.*

[282] R.J. Lechner, "Harmonic analysis of switching functions," in "Recent Developments in Switching Theory" (A. Mukhopadhay, ed.). Academic Press, *New York, 1971.*

[283] S.C. Lee, "Digital Circuits and Logic Design," Prentice Hall, *Englewood Cliffs, New Jersey, 1976.*

[284] E.B. Lee, and M.A. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," Proc. of Intern. IEEE Conf. on Systems, Man, and Cybernetics, *Halifax, Nova Scotia, pp. 248-260, 1986.*

[285] D. Levin, "Computer-Aided Design of Digital Systems," Crane, Russak & Co. Inc., *1977.*

[286] K. F. Lee, and S. Mahajan, "A Pattern Classification Approach to Evaluation Function Learning," Artificial Intelligence, *Vol. 36., pp. 1-25, 1988.*

[287] W.J. Leech, "A rule-based process control method with feedback," Advances in Instrumentation, *No. 41, pp. 169–175, 1986.*

[288] D.B. Lenat, "On Automated Scientific Theory Formation: A Case Study Using AM Program," Machine Intellience, *Ed. J.E. Hayes, D. Mitchie and L.I. Mikulich, Vol. 9, Halsted Press, New York, 1977.*

[289] D. Lenat, "The Ubiquity of Discovery," Proc. Vth IJCAI, *pp. 1093-1105, 1977. (***How to write programs that "invent" new ideas. Applications in Math.***)*

[290] D. Lenat, "Automated Theory Formation in Mathematics," Proc. Vth IJCAI, *pp. 833-842, 1977. (***Another important paper from Lenat***).*

[291] D.B. Lenat, "Learning from Observation and Discovery," Machine Learning: An Aritficial Intellience Approach, *Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.*

[292] V.R. Lesser, and L.D. Erman, "A Retrospective View of the Hearsay-II Architecture," Proc. Vth IJCAI, *pp. 790-800, 1977. (***Early paper on blackboard model of solving problems.***)*

[293] G. Levi, Sirovich., "A Problem Reduction Model for Not Independent Subproblems," Proc. IVth A.I. Conf., *Tbilisi, USSR, Georgia, pp. 340-344, 1975.*

[294] K.J. Lieberherr, and E. Specker, "Complexity of Partial Satisfaction," Journal of the Association for Computing Machinery, *Vol. 28, No. 2, April 1981, pp. 411-421.*

[295] K.J. Lieberherr, and S.A. Vavasis, "Analysis of Polynomial Approximation Algorithms for Constraint Expressions," Lecture Notes in Computer Science, *Vol. 145, pp. 187-198.*

[296] T. Luba, and H. Selvaraj, "A General Approach to Boolean Function Decomposition and its Applications in FPGA-based Synthesis," VLSI Design. Special Issue on Decompositions in VLSI Design, *1995.*

[297] G.F. Luger, and Stubblefield, "Artificial Intelligence and the Design of Expert System," Benjamin/Cummings, *1989. (***This is my favourite AI textbook. It has plenty of ready programs in Lisp and Prolog with good explanations. Most of them can be easily modified for the purposes of this book.***)*

[298] R. Lupinski, "Analysis of LISP 1.5.9 Properties," M.Sc.Th., *Warsaw University, Department of Mathematics, 1976. (***This is an M.S. from Warsaw, that I took my algebraic simplifier from.***)*

[299] G. De Micheli, R. Brayton, and A.L. Sangiovanni-Vincentelli, "KISS: A Program for Optimal State Asignment of Finite State Machines," Int. Conf. on Comp. Aid. Design., *Santa Clara, November 1984.*

[300] G. De Micheli, "Computer-Aided Synthesis of PLA-based Systems," Ph.D Dissertation, *University of California, Berkeley 1983.*

[301] G. De Micheli, R. Brayton, and A.L. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," IBM Research Report, *RC 10599.*

[302] G. De Micheli, M. Hoffman, A.R. Newton, and A.L. Sangiovanni-Vincentelli, "A Design System for PLA-based Digital Circuits," in Advances in Computer Engineering Design, Jai Press, *1984*

[303] G. De Micheli, "Optimal Encoding of Control Logic," Int. Conf. on Circ. and Comp. Des., *Rye NY, Sept. 1984.*

[304] S.A. Majorov, "Design of Digital Computers," Energia, *Leningrad 1972 (in Russian).*

[305] E.J. McCluskey,Jr., "Introduction to the Theory of Switching Circuit," McGraw-Hill, *1965.*

[306] Madarasz, R.L., Heiny, L.C., Cromp, R.F., Mazur, N.M., "The Design of an Autonomous Vehicle for the Disabled," IEEE Journal of Robotics and Automation, *Vol. RA-2, No. 3., September 1986.*

[307] Mavaddat, F., "WATSON/I: Waterloo's Sonically Guided Robot," J. Microcomputer Application, *Vol. 6., pp. 37-45, 1983.*

[308] R.E. Miller, "Switching Theory. Vol. 1 + 2 ", John Wiley, New York 1965.

[309] D.E. Muller, "Application of Boolean Algebra to Switching Circuit Design and Error Detection", IRE Trans. 1954, pp. 6-12.

[310] S. Muroga, T. Ibaraki, "Design of Optimal Switching Networks by Integer Programming", IEEE Trans. on Comp., Vol. C-21, pp. 573-582, June 1972.

[311] Z. Manna, and R. Waldinger, "The Automatic Synthesis of Systems of Recursive Programs," Proc. Vth IJCAI, *pp. 405-411, 1977.*

[312] Z. Manna, and R. Waldinger, "The Automatic Synthesis of Recursive Programs," SIGPLAN Notices, *Vol. 12, No. 8, August 1977, pp. 29-36.*

[313] A. Martelli, and U. Montanari, "From Dynamic Programming to Search Algorithms with Functional Costs," Proc. IVth A. I. Conf., *Tbilisi, USSR, Georgia, pp. 345-350, 1975.*

[314] Z. Manna, and R. Waldinger, "Knowledge and Reasoning in Program Synthesis," Proc. IJCAI 75, *pp. 288-295. (***This paper is more advanced but has deep ideas.***)*

[315] M. Manove, S. Bloom, and C. Engelman, "Rational Functions in MATHLAB," IFIP Working Conf. Symbol Manipulation Languages, *Pisa, September 1966. (***Illustrates how to expand symbol manipulation system for a new algebraic system.***)*

[316] J. Martinek, "Synthesis of Programs through Generation of Symbolic Expressions," Ph.D. Thesis, *Poznan Techn. Univ., Poznan, Poland 1972. (***Early ideas of program synthesis in Lisp with good examples.***)*

[317] J. Martinek, "LISP, Description, Realization and Applications," WNT, *Warsau, 1980. (in Polish). (***You will not read it because it is in Polish, but this is a source of excellent project ideas for me ;-)***).*

[318] W.D. Maurer, "The Programmer's Introduction to LISP," McDonald, *London, 1972. (***Many good examples of techniques. Short.***)*

[319] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," CACM, *Vol. 3, New York, April 1960, pp. 184-195. (***This is the famous paper of McCarthy which introduced Lisp. Read it if you want to see how great ideas were created.***)*

[320] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.J. Levin, "LISP 1.5 Programmers Manual," MIT Press, *1962. (***The first Lisp manual from MIT.***)*

[321] J. McCarthy, "A Basis for a Mathematical Theory of Computation, Computer Programming and Formal Systems," North Holland Company, *Amsterdam, 1963. (***The first theory of Lisp.***)*

[322] V. Malyugin, "Habilitation Thesis," Institute of Control Systems, *Russian Academy of Sciences, Moscow, 1980.*

[323] M.P. Marcus, "Derivation of Maximal Compatibles Using Boolean Algebra," IBM J. Res.Dev., *Vol. 8, pp. 537-538, 1964.*

[324] C. Matheus, "Feature Construction: An Analytic Framework and Application to Decision Trees," University of Illinois, PhD Dissertation, Urbana-Champaign, 1989.

[325] J.T. McCall, J.G. Tront, F.G. Gray, R.T. Haralick, and W.M. McCormack, "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem," IEEE TC., *Vol. C-34, No. 11, November 1985.*

[326] Min Wen Du, "A Way to Find a Lower Bound for the Minimal Solution of the Covering Problem," IEEE TC, *Vol. C-21, pp. 317-318, March 1972.*

[327] A. Merchant, B. Melamed, E. Schenfeld, B. Sengupta, "Analysis of a Control Mechanism for a Variable Speed Processor," IEEE Tr. Comp., *Vol. 45, No. 7, July 1996, pp. 793-801.*

[328] R.S. Michalski and J.B. Larson, "Inductive Inference of VL Decision Rules," Workshop in Pattern-Directed Inference Systems, *Hawaii, May, 1977.*

[329] R.S. Michalski, "Pattern Recognition as Knowledge-Guided Computer Induction," Technical Report No. 927, *Department of Computer Science, University of Illinois, Urbana-Champaign, 1978.*

[330] R.S. Michalski, "A Theory and Methodology of Inductive Learning," Machine Learning: An Artificial Intelligence Approach, *Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.*

[331] T.M. Mitchell, P.E. Utgoff and R. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," Machine Learning: An Artificial Intelligence Approach, *Ed. R.S. Michalski and J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.*

[332] P. Miller, "An Adaptive Natural Language System that Listens, Asks and Learns," Proc. IJCAI 75, *pp. 406-413.*

[333] G.A. Miller, "What is Information Measurement," American Psychologist, *1963, No. 8, pp. 3-11.*

[334] T. Mitchell, J. Carbonell, and R. Michalski, (eds), Machine Learning: A Guide to Current Research. (Knowledge Representation, Learning, and Expert Systems). *Kluwer Academic Publishers, Nowell, MA, 1986.*

[335] R. Mirchandaney, and J. Stankovic, "Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems," J. Parallel Distrib. Comput., *Vol. 3., No. 4., Dec. 1987, pp. 527-552.*

[336] Ming-Te Chao, and J. Franco, "Probabilistic Analysis of a Generalization of the Unit Clause Literal Selection Heuristic for the k-Satisfiability Problem," Technical Report No. 165, *Indiana University Computer Science Department, January 1985.*

[337] D. Mitchie, "Problem Decomposition and the Learning of Skills," Proceedings of the Europeean Conference on Machine Learning, Lecture Notes in Artificial Intelligence, *Vol. 912, Spriger Verlag, Berlin, Heidelberg, New York, Inc., pp. 17–31, 1995.*

[338] D. Michie, "Problems of computer-aided concept formation," Ed. R. Quinlan, Applications of Expert Systems (Vol. 2), *1989, Wokingham, UK, Addison-Wesley.*

[339] S. Minato, "Graph-Based Representations of Discrete Functions," Proc. Reed-Muller'95 Workshop, *pp. 1–10, Chiba, Japan, August, 1995.*

[340] C. Montagero, G. Pacini, and F. Turini, "MAGMA-LISP: A machine Language for Artificial Intelligence," Proc. IJCAI 75, *pp. 556-561. (***Example of one of many Lisp extensions from the 80-ties. Only Prolog survived, because it had better semantics and was simpler.***)*

[341] J.B. Morris, and D.J. Singleton, "6400/6600 LISP 1.5 an Adaptation of MIT LISP 1.5," The University of Texas at Austin, *1968. (***I give it only for historical reasons, because our programs were written in it.***)*

[342] J. Moses, "Solutions of Systems of Polynomial Equations by Elimination," CACM, *August, 1966. (***Useful Lisp techniques, if you have this type of problem to solve.***)*

[343] K. Morik, "Sloppy Modeling," Knowledge Representation and Organization in Machine Learning, *Ed. K. Morik, Springer-Verlag, Berlin Heidelberg, 1989.*

[344] S. Muggleton, "Duce, and Oracle-based Approach to Constructive Induction," Proceedings of IJCAI-87, *pp. 287–292, Milan, Italy, Morgan Kaufmann, 1987.*

[345] S. Muggleton and W. Buntine, "Machine Invention of First Order Predictes by Inverting Resolution," Proceedings of the 5th International Conference on Machine Learning, *pp. 339–352, Ann Arbor, MI, Morgan Kaufmann, 1988.*

[346] S. Muggleton and C. Feng, "Efficient induction of logic programs," Proc. First Conf. on Algorithmic Learning Theory, *Ed. S. Arikawa, S. Goto, S. Ohsuga and T. Yokomori, Tokyo, Japan, Soc. Art. Intell., pp. 368–381, 1990.*

[347] J. Munkres, "Algorithm for the Assignment and Transportation Problems," J. SIAM, *Vol.5., pp. 32-58, 1957.*

[348] A.R. Newton, and A.L.Sangiovanni-Vincentelli, "Computer-Aidded Design for VLSI Circuit," IEEE Computer, *Apr. 1986.*

[349] L. Nguyen, M. Perkowski, N. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers", Proceedings of 24th Design Automation Conference, June 28 - July 1, 1987, Miami, Florida, Paper 33.3.

[350] S. Nandi, P. Pal Chaudhuri, "Analysis of Periodic and Intermediate Boundary," 90/150 Cellular Automata. 1-12 Proc. 10th International Symposium on System Synthesis, *Antwerp, Belgium, September 17-19, 1997.*

[351] N.J. Nilsson, "Problem Solving Methods in Artificial Intelligence," McGraw-Hill, *New York, 1971.*

[352] I. Noda and M. Nagao, "A Learning Method for Recurrent Networks Based on Minimization of Finite Automata," Proc. IJCNN'92, *Baltimore, pp. 127-132, Jun., 1992.*

[353] P. Norvig, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp," Morgan Kaufmann Publ., *1995. (***A recent Lisp Superbook, with many excellent examples; comphrehensive, advanced.***)*

[354] A. Newell, "Heuristic Programming: Ill-Structured Problems," Progress in Operations Research, Vol. 3, John Wiley and Sons, *New York, 1969. (***One of very useful papers by Newell, one of creators of AI.***)*

[355] A. Newell, and H.A. Simon, "Human Problem Solving," Prentice Hall, *Englewood Cliffs, New Jersey, 1972. (***The fundamental book by Newell and Simon.***)*

[356] A. Newell, "Production Systems: Models of Control Structure," Visual Information Processing, *Chase, W. G. (ed.), Academic Press, New York, 1972. (***Very useful paper about rule-based systems.***)*

[357] N.J. Nilsson, "Problem-Solving Methods in Artificial Intelligence," McGraw Hill, *New York, 1971. (***Good explanation of search and automatic theorem proving.***)*

[358] N. J. Nilsson, Learning Machines, *McGraw-Hill, New York, 1965.*

[359] =

[360] N.J. Nilsson, "Problem-Solving Methods in Artificial Intelligence," Mc Graw Hill, *New York, 1971.*

[361] T. Ohtsuki, H. Mori, E.S. Kuh, T. Kashiwabara, T. Fujisawa, "One -Dimensional, Logic Gate Assignment and Interval Graphs", IEEE Transactions on Circuits And Systems, Vol. CAS, 26 September 1979, pp.675-684.

[362] A.L. de Oliveira, "Inductive Learning by Selection of Minimal Complexity Representations, University of California at Berkeley, 1994.

[363] E.W. Page, and P.N. Marinos, "Programmable Array Realizations of Synchronous Sequential Machines," *IEEE Trans. on Computers,* Vol. C-26, No.8, pp. 811-818, August 1977.

[364] A.C. Parker, D.E. Thomas, D.P. Siewiorek, M. Barbacci, L. Hafer, G. Leive, J. Kim, "The CMU Design Automation System - An Example of Automated Data Path Design", Proc. 16th Design Automation Conf., June 1979, pp. 73-80.

[365] A. Papoulis, "Probability, random variables, and stochastic processes," *McGraw-Hill, Inc.*, 1991.

[366] M. Perkowski, "A system for automatic design of digital systems". Proceedings of the FCIP Symposium INFORMATICA 74, Bled, Yugoslavia, 7-12 October 1974, paper 4.4.

[367] M. Perkowski, "Synthesis of multioutput three level NAND networks". Proceedings of the Seminar on Computer Aided Design. Budapest, Hungary, 3-5 November 1976, pp. 238-265.

[368] M. Perkowski, "The state-space approach to the design of multipurpose problem-solver for logic design". "Artificial Intelligence and Pattern Recognition in Computer-Aided Design", J. C. Latombe (ed.), North Holland, Amsterdam, pp. 124-140, 1978.

[369] M. Perkowski, "Digital Devices Design by Problem-Solving Transformations". Journal on Computers and Artificial Intelligence. Vol. 1, No. 4, August 1982, pp. 343-365.

[370] M. Perkowski, "A Method for Parallel Minimization of Strongly Unspecified Multiple-Valued Input Two-Valued Output Boolean Functions", report.

[371] M. Perkowski, D. Smith, R. Krzywiec, "Logic Simulation/Design/Verification Environment in Prolog", Proceedings of the 17th Annual Pittsburgh Conference on Modelling and Simulation, April 24- 25, 1986, University of Pittsburgh.

[372] M. Perkowski, "Minimization of Two-Level Networks from Negative Gates", Proc. Midwest 86 Conference on Circuits and Systems, Lincoln, Nebraska, 1- 12 August 1986.

[373] M. Perkowski, "A Parallel Programming Approach to the Design of Two-Level Networks with Negative Gates", International Workshop on Logic Synthesis, Research Triangle Park, North Carolina, May 12 - 15, 1987.

[374] M. Perkowski, L.J. Ming, A. Wieclawski, "An Expert System for Optimization of Multi-Level Logic", IASTED Conference, Applied Simulation and Modeling, ASM '87, Santa Barbara, CA, May 26 - 29, 1987.

[375] M. Perkowski, J. Liu, "A System for Fast Prototyping of Logic Design Programs", 1987 Midwest Conference on Circuits and Systems, Syracuse, New York.

[376] M. Perkowski, H. Uong, H. Uong, "Automatic Design of Finite State Machines with Electronically Programmable Devices", record of Northcon 87, Portland 1987, paper 13/4.

[377] M. Perkowski, J. Liu, J. Brown, "Quick Software Prototyping: CAD Design of Digital CAD Algorithms", In G. Zobrist (ed) "Progress in Computer Aided VLSI Design", Ablex Publishing Corp., 1988.

[378] M. Perkowski, J.E. Brown, "An Unified Approach to Designs Implemented with Multiplexers and to the Decomposition of Boolean Functions", accepted to Proceedings of 1988 ASEE National Conference, Portland, Oregon, June 19-23, 1988.

[379] M.A. Perkowski, K.A. Pirkl, J.E. Brown, "Exact Minimization of Strongly Unspecified Boolean Functions with Multiple-Valued Inputs on a Shared Memory Parallel Computer", report. 1988.

[380] M. Perkowski, J. Brandenburg, "Parallel Algorithms for Basic Problems of Boolean Algebra", report, 1989.

[381] M. Perkowski, "Synthesis of Multioutput Three Level NAND Networks", Proceedings of the Seminar on Computer Aided Design, Budapest, Hungary, 3-5 November 1976, pp.238-265.

[382] M. Perkowski, A. Rydzewski, P. Misiurewicz, "Theory of Logic Circuits. Selected Problems", Publishers of Warsaw Technical University, Ed.1, 1977, Ed.2, 1978 (in Polish).

[383] M. Perkowski, "Digital Devices Design by Problem-Solving Transformations", J. Comp. & Artif. Intel., Vol.1, No.4, August 1982, pp. 345-365.

[384] M. Perkowski, N.B. Goldstein, L. Nguyen, "PLA with Tail Gives Estimate to Minimum Solution and Has a Decreased Area", Report, PSU 1985.

[385] M. Perkowski, N.B. Goldstein, L. Nguyen, "Synthesis of Multi-Level PLAs by Recursive Graph-Coloring", Report, 1985.

[386] M. Perkowski, "The State-Space Approach to the Design of Multipurpose Problem-Solver for Logic Design," *Proc. of IFIP, WG. 5.2, Conference, "Artificial Intelligence and Pattern Recognition in Computer Aided Design,"* North Holland Publ. Co., pp. 123-140, Grenoble, March 17-19, 1978.

[387] M. Perkowski, "The Method of Solving Combinatorial Problems in the Automatic Design of Digital Systems," *Institute of Automatic Control,* Warsaw Technical University, 1980 (in Polish).

[388] M.A. Perkowski, A. Rydzewski, and P. Misiurewicz, *Theory of Logic Circuits. Selected Problems,* Publishers of the Technical University of Warsaw (1977) (book in Polish).

[389] M.A. Perkowski, J. Liu, and J.E. Brown, *Rapid Software Prototyping: CAD Design of Digital CAD Algorithms Progress in Computer-Aided VLSI Design. Tools. Vol. 1,* G. W. Zobrist, ed., Ablex, pp. 353-401, 1989.

[390] M.A. Perkowski, P. Dysko, and B. Falkowski, "Two Learning Methods for a Tree-Search Combinational Optimizer," *Proc. Intern. Phoenix Conference on Computers and Communication*, Scottsdale, Arizona, March, pp. 606-613, 1990.

[391] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Zhi Wang, and Jin S. Zhang, "Decomposition of multiple-valued relations," *Proc. of ISMVL'97*, Halifax, Nova Scotia, Canada, pp. 13–18, May 28-30, 1997.

[392] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. Reed-Muller'95 Workshop*, pp. 102–109, Chiba, Japan, August, 1995.

[393] M. Perkowski, T. Luba, S. Grygiel, M. Kolsteren, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *PSU Report*, unpublished, Version III, September 1995.

[394] M. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and its Application to Multi-Level AND/OR/EXOR Synthesis," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 143-151.

[395] M.A. Perkowski, "The State-Space Approach to the Design of Multipurpose Problem-Solver for Logic Design," *Proceedings of IFIP WG.5.2. Conference "Artificial Intelligence and Pattern Recognition in Computer Aided Design,* Grenoble, March 17-19, North Holland Publishing Company, 1978. (**My early system that uses AI in various CAD problems.**)

[396]

[397] M. Perkowski, "The state space approach to the design of multipurpose problem solver for logic design," *Proc. IFIP WG 5.2. Conference "Artificial Intelligence and Pattern Recognition in Computer Aided Design,* Grenoble 17-19 March, North Holland Publ. Comp., Amsterdam, 1978, pp. 123- 140.

[398] T.W. Pratt, and D. Friedman, "A Language Extension for Graph Processing and Its Formal Semantics," *CACM*, Vol. 14., No. 7., July 1971.

[399] C. Perdue, and H.J. Berliner, "EG - A Case Study in Problem Solving with King and Pawn Endings," *Proc. Vth IJCAI,* pp. 421-427, 1977.

[400] U. Petersohn, K. Voss, and K.H. Weber, "Genetische Adaptation - ein Stochastisches Suchverfahren fuer Diskrete Optimiesirungsprobleme," *Math. Operationsforschung und Statistik,* 1974, Vol. 5, No. 7-8. (**Early German work on Genetic Adaptation.**)

[401] I. Pohl, "Bi-directorial Search," *Machine Intelligence*, 6, Meltzer, B. and Michie, D. (eds.), American Elsevier, New York, pp. 127-140, 1971. (**Very useful paper if you want to implement bi-directional search.**)

[402] Pohl 71

[403] Pohl 73

[404] Popo 76

[405] *Proc. of International Workshop on Logic Synthesis,* Vol. 1 + 2, Research Triangle Park, North Carolina, May 12-15, 1987.

[406] S.R. Petrick, "On the Minimization of Boolean Functions," *Proc. Symposium on Switching Theory,* IFIP, Paris, France, June 1959.

[407] P.W. Purdom, Jr., "Search Rearrangement Backtracking and Polynomial Average Time," *Artificial Intelligence,* No. 21, pp. 117-133, 1983.

[408] D. Pertsekas, "The auction algorithm: a distributed relaxation method for the assignment problem," *Oper. Res.,* Vol. 14., 1-4, June 1988, pp. 105-123.

[409] L. Pyber, "Clique covering of graphs," *Combinatorica,* Vol. 6., No. 4., 1986, pp. 393-398.

[410] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, No. 1, pp. 81–106, 1986.

[411] J. R. Quinlan, "C4.5: Programs for Machine Learning," *Morgan Kaufmann*, Los Altos, California, 1992.

[412] J.R. Quinlan, "Predicting the Length of Solutions to Problems," *Proc. IVth IJCAI,* pp. 363-369, 1975.

[413] C. Rowen, "Multi-Level Logic Array Synthesis", Technical Report No. 85-279, Computer Systems Lab., Stanford University, July 1985.

[414] R.H. Risch, "Staggered Input Networks: An Approach to Automatic Logic Decomposition", Proc. of 1982 ISCAS Symp., Rome, pp. 55-57, May 1982.

[415] J.P. Robinson, C.L. Yeh, "A Method for Modulo-2 Minimization", IEEE Trans. on Comp., Vol. C-31, No. 8., August 1982, pp. 800-801.

[416] J.P. Roth, "Computer Logic, Testing, and Verification", Computer Science Press, Potomac, MD., 1980.

[417] M.O. Rabin, "Probabilistic Algorithms," In J.Traub (ed.) *Algorithms and Complexity: New Directions and Recent Results,* Academic Press, New York, 1976, pp. 21-39.

[418] R. Reboh, and E. Sacerdoti, "A Preliminary QLISP Manual," *Technical Note 81,* SRI Project 8721, October 1974. (**One more of the AI languages built on Lisp. From Stanford. It was used in robotics research.**)

[419] L. Rendell, "Substantial Constructive Induction Using Layered Information Compression: Tractable Feature Formation in Search," *Proceedings of IJCAI-85*, pp. 650–658, 1985.

[420] L. De Raedt and M. Bruynooghe, "Constructive Induction by Analogy," *Proceedings of EWSL-89*, pp. 189–200, Pitman, Montpellier, France, 1989.

[421] M. Reiter, and G. Sherman, "Discrete Optimizing," *J. Soc. Industr. and Appl. Math.,* Vol. 13., No. 3., 1965.

[422] D. Ribbens, "Programmation Non Numerique LISP 1.5," *Dunod,* Paris, 1969. (**If you read French. I took some good examples from this book.**)

[423] L. Rendell, "Substantial Constructive Induction Using Layered Information Compression: Tractable Feature Formation in Search," *Proceedings of IJCAI-85*, pp. 650–658, 1985.

[424] L. Rendell, "A New Basis for State-Space Learning Systems and a Successful Implementation," *Artificial Intelligence,* Vol. 20., 1983, pp. 369-392.

[425] Ch. Rich, "Plan Recognition in a Programmers Apprentice," *MIT Al Lab.*, Working Paper 147. May 1977.

[426] P. Riddle, R. Segal, and O. Etzioni, "Representation design and brute-force induction in a Boeing manufacturing domain," *Applied Artificial Intelligence*, 1994, No. 8, pp. 125–148,

[427] C. Riese, "Transformer fault detection and diagnosis using RuleMaster by Radian," Austin, TX, Radian Corporation, 1984.

[428] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[429] E. Ruf, and D. Weise, "LogScheme: Integrating Logic Programming into Scheme," *Lisp and Symbolic Computation*, 3, No. 3, pp. 245-288, 1990. (**This paper shows how you can add logic programming to Scheme, the same can be done for Lisp.**)

[430] M.D. Rychener, "Control Requirements for the Design of Production System Architectures," *SIGPLAN Notices*, Vol. 12, No. 8, August 1977, pp. 37-44.

[431] G. Saucier, M. Crastes de Paulet, P. Socard, "ASYL: A Rule Based System for Controller Synthesis," *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No. 6, Nov. 1987.

[432] J.R. Southard, A. Domic, K.W. Crouch, "Report on the Lincoln Boolean Synthesizer", Digest of ICCAD , pp.192-193, IEEE, September 1983.

[433] A.L. Sangiovanni-Vincentelli, "An Overview of Synthesis Systems", Proc. IEEE 1985 Custom Integrated Circuits Conference, pp.221-225, May 1985.

[434] K.K. Saluja, R.M. Reddy, "Fault Detecting Test Sets for Reed-Muller Canonic Networks", IEEE Trans. Computers, Vol. C-24, Oct. 1975, pp. 995-998.

[435] D.C. Schmidt, G. Metze, "Modular Replacement of Combinational Switching Networks", IEEE Trans. on Comp., Vol. C-24, pp. 29-48, 1975.

[436] V. Shen, A. Mc Kellar, "An Algorithm for the Disjunctive Decomposition of Switching Functions", IEEE Trans. on Comp. Vol. C-19, pp. 239-248, 1970.

[437] V. Shen, A. Mc Kellar, "A Fast Algorithm for the Disjunctive Decomposition of Switching Functions", IEEE Trans. on Comp. Vol. C-20, pp. 304-309, 1971.

[438] T.Shinsha et al., "POLARIS: Polarity Propagation Algorithm For Combinational Logic Synthesis", 21st Design Automation Conference, IEEE 1984.

[439] I. Shirakawa, N. Okuda, T. Harada, S. Tani, H. Ozaki, "A Layout System for Random Logic Portion of MOS LSI", Proceedings of 17th DAC, Minneapolis, pp.92-99 , 1980.

[440] S.G.Shiva, "Combinational Logic Synthesis from HDL Description", Proceedings of 17th DAC, Minneapolis, pp. 550-555 , 1980.

[441] J.R. Slagle, "Artificial Intelligence: The Heuristic Programming Approach," *Mc Graw Hill*, New York, 1971.

[442] K. Sasidhar, S. Chattopadhyay, P. P. Chaudhuri, "CAA Decoder for Cellular Automata Based Byte Error Correcting Code," *IEEE Tr. Comp.,* Vol. 45, No. 9, Sept. 1996. pp. 1003-1016

[443] A. Svoboda, "Parallel Processing in Boolean Algebra," *IEEE TC,* Vol. C-22, No. 9, pp. 848-851, Sept 1973.

[444] J.C. Schlimmer, "Learning and Representation Change," *Proceedings of AAAI-87*, pp. 511–515, Morgan Kaufmann, 1987.

[445] A. Svoboda, "Advanced Logical Circuit Design Techniques", Garland STMP Press, New York, 1979.

[446] M.J.E. Sternberg, R.D. King, R.A. Lewis and S. Muggleton, "Application of machine learning to structural molecular biology," *Phil. Trans. R. Soc. Lond. B*, Vol. 344, pp. 365–371, 1994.

[447] Saunders, R., "The thallium diagnostic workstation: Learning to diagnose heart imagery from examples," *Proceedings of the Second Innovative Applications of Artificial Intelligence Conference*, pp. 105–118, Menlo Park, CA, AAAI Press, 1991.

[448] S. Salzberg, R. Chandar, H. Ford, S.K. Murthy and R. White, "Decision trees for automated identication of cosmic-ray hits in Hubble Space Telescope images," *Publications of the Astronomical Society of the Pacific*, No. 107, pp. 279–288, 1995.

[449] S. Slocombe, K. Moore, and M. Zelouf, "Engineering expert system applications," *Presented at the British Computer Scociety Annual Conference*, 1986.

[450] A. Srinivasan, T. Kam, S. Malik, and R. Brayton, "Algorithms for discrete function manipulation," *IEEE International Conference on CAD*, pp. 92–95, 1990.

[451] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *IEEE Transactions on Computers*, Vol. C-30, pp. 635–643, September, 1981.

[452] D.C. Smith, and H.J. Enea, "Backtracking in MLISP2," *Proc. IJCAI 75*, Session 25: Hardware and Software for AI, pp. 677-685. (**This paper explains one of the methods to program backtracking. Backtracking is the most important and useful technique that we emphasize in this book for EDA applications.**)

[453] G.L. Steele, "Design of LISP-Based Processors, or SCHEME: A Dielectric LISP or, Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode," *AI Lab Memo 379*, MIT, 1980. (**One of sources of good ideas on designing Lisp machines in silicon. We will use it.**)

[454] G.L. Steele, and G.J. Sussman, "Design of a LISP-Based Processor," *Communications of the ACM 23*, No.11, pp. 628-645, 1980. (**One more of them from Steel and Sussman, top Lisp authorities.**)

[455] E. Sandewall, "Ideas about Management of LISP Data Base," *Proc. IJCAI 75*, pp. 585-591. (**Important paper about managing data bases in Lisp. These techniques are still very useful for our projects.**)

[456] E. Sandewall, "Programming in an Interactive Environment: The LISP Experience," Linkoping, 1977. *Computing Surveys*, March, 1978. (**Lisp interactivity from one of top Lisp programmers.**)

[457] E. Sandewall, "Artificial Intelligence - A Background Paper for the AI/PR/CAD Conference," *Preprint.*

[458] D.E. Shaw, and C.G. Green, "Inferring LISP Programs from Examples," *Proc. IJCAI 75*, pp. 260-267.

[459] S.Y.H. Su, C.W. Nam, "Computer Aided Synthesis of Multiple Output Multilevel NAND Networks With Fan-In and Fan-Out Constraints", IEEE Trans. on Comp., Vol. C-20, pp. 1445-1455, 1971.

[460] L. Siklossy, and D.A. Sykes, "Automatic program Synthesis from Example Problems," *Proc. IJCAI 75*, pp. 268-273.

[461] L. Siklossy, and J. Dreussi, "An Efficient Robot Planner Which Generates Its Own Procedures," *Proc. Third International Joint Conference on Artificial Intelligence*, Stanford, California, August 1973.

[462] L. Siklossy, "Let's talk LISP," *Prentice-Hall, Inc.*, Englewood Cliffs, New Jersey, 1976. (**This is the book from which I learned Lisp. You will find solutions to problems from this book on the CD ROM.**)

[463] J.R. Slagle, "A Multi-Purpose Theorem Proving Heuristic Program that Learns," *Proceedings of IFIP Congress,* 1965, Vol. 2. (**Nice combination of theorem proving and learning from one of AI pioneers.**)

[464] J.R. Slagle, "Experiments with a Deductive Question-answering program," *CACM*, December 1965.

[465] J.R. Slagle, "Artificial Intelligence: The Heuristic Programming Approach," *McGraw-Hill*, 1972. (**A classical textbook in AI. Easy to understand, good for beginners.**)

[466] D.C. Schmidt, and L.E. Druffel, "An Extension of the Clause-Table Approach to Multi-Output Combinational Switching Networks," *IEEE TC,* Vol. C-23, April 1974.

[467] J.R. Slagle, Ch..L. Chang, and R.C.T. Lee, "A New Algorithm for Generating Prime Implicants," *IEEE TC.* Vol. C-19, No. 4, pp. 304- 310, April 1970.

[468] E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, Vol. 5, No. 2, pp. 115-135, 1974.

[469] E.D. Sacerdoti, "Structure for Plans and Behavior," *SRI Al Center Technical Note*, 109, August 1975.

[470] R. Sedgewick, "Algorithms," *Addison-Wesley*, 1988. (**Standard textbook. A must, if you do not have it yet.**)

[471] H.E. Shrobe, "Plan Verification in a Programmer's Apprentice," *MIT Al Working Paper*, 158, January 1977.

[472] R.G. Smith, T.M. Mitchell, R.A. Chestek, and B.G. Buchanan, "A Model for Searching Systems," *Proc. Vth IJCAI*, pp. 338-343, 1977.

[473] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J.,* Vol. 3., 1959, pp. 210-229.

[474] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," II, *IBM J.,* Vol. 11., 1967, pp. 601-617.

[475] C.E. Shannon, and W. Weaver, "The Mathematical Theory of Communication," *University of Illinois Press,* 1975 (first published in 1949).

[476] J. R. Slagle, "Artificial Intelligence: the Heuristic Programming Approach," *McGraw Hill,* New York 1970.

[477] H.C. Torng, "An Algorithm for Finding Secondary Assignments of Synchronous Sequential Circuits," *IEEE Trans. on Comp.*, Vol C-17, pp.416-469, May, 1968.

[478] J.H. Tracey, "Internal State Assignment for Asynchronous Machines," *IEEE Trans. on Electr. Comp.*, Vol. EC-15, pp. 551-560, August 1966.

[479] Ch-J. Tseng, A.M. Prabhu, C. Li, Z. Memood, and M.M. Tong, "A Versatile Finite State Machine Synthesizer," *IEEE ICCAD*, pp. 206-209, 1986.

[480] C. Tseng, and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on CAD,* Vol. CAD-5, No. 3, July 1986.

[481] A. Thayse, "A Fast Algorithm for Proper Decomposition of Boolean Functions", Philips Res. Rep. No. 27, pp. 140-150, 1972.

[482] A. Thayse, "Boolean Calculus of Differences", Springer Verlag, Berlin-New York, 1981.

[483] A. Thayse, "Boolean Calculus of Differences," *Springer Verlag,* Berlin-New York, 1981.

[484] H.C. Torng, "Introduction to the Logical Design of Switching Systems," *Addison-Wesley*, Reading, Massachusetts, 1964.

[485] Traczyk, 1974

[486] H.C. Torng, "Introduction to the Logical Design of Switching Systems," *Addison-Wesley,* Reading, Massachusetts, 1964.

[487] H.C. Torng, "An Algorithm for Finding Secondary Assignments of Synchronous Sequential Circuits," *IEEE Trans. on Comp.*, Vol. C-17, pp. 416-469, May 1968.

[488] S. Tanimoto, "The Elements of Artificial Intelligence using Common Lisp," *Computer Science Press*, 1990. (**One of several useful textbooks that combine explanation of AI with Lisp.**)

[489] D. Touretzky, "Common Lisp: A Gentle Introduction to Symbolic Computation," *Benjamin Cummings*, 1989. (**A popular textbook of Common Lisp.**)

[490] Torng, H.C.: "Introduction to the Logical Design of Switching Systems," Addison-Wesley, Reading, Massachusetts, 1964.

[491] R. L. Thorndike, "The problem of classification of personnel," *Psymetrika,* 15, 1950, pp. 215-235.

[492] P.E. Utgoff, "Shift of Bias for Inductive Concept Learning," Rutgers University, PhD Dissertation, 1984.

[493] P.E. Utgoff, "Shift of Bias for Inductive Concept Learning, *Machine Learning: An Artificial Intelligence Approach*, Vol. 2, Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1986.

[494] U.C. Irvine, "U.C. Irvine, Repository of Machine Learning Databases and Domain Theories," `ftp://ftp.ics.uci.edu/pub/ machine-learning-databases/`

[495] M.E. Ulug, and B.A. Bowen, "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems," *IEEE TC*, pp. 255-267, March 1974.

[496] Ulug, *Proc. IEEE Intern. Conference on Computers and Communications*, Arizona, 1985.

[497] S.H. Unger, "The Essence of Logic Circuits," *Prentice-Hall, Inc.*, 1989.

[498] P. Weiner, and E.J. Smith, "Optimization of Reduced Dependencies for Synchronous Sequential Machines," *IEEE Trans. on Electr. Comp.*, Vol. EC-16, pp. 835-847, December 1967.

[499] Welsh R.L., Blasch, B.B., "Foundations of Orientation and Mobility," *American Foundation for the Blind*, New York, 1980.

[500] J.M. Williams, "Technology and the Handicapped," *American Education,* Vol. 20., No.5., pp. 27-30, June 1984.

[501] A. Wieclawski, and M. Perkowski, "Optimization of Negative Gate Networks Realized in Weinberger-like Layout in a Boolean Level Silicon Compiler," *Proc. 23rd Design Automation Conference,* Albuquerque, June 25-27, 1984.

[502] A. Wieclawski, and M. Perkowski, "Optimization of Negative Gate Networks," *Department of Electrical Engineering, Portland State University, Technical Report,* version 2, May 1984.

[503] A. Wieclawski, M. Perkowski, "Optimization of Negative Gate Networks Realized in Weinberger-like Layout in a Boolean Level Silicon Compiler", Proceedings of 21st Design Automation Conf., IEEE , Albuquerque, pp. ?? , 25-27 June, 1984.

[504] A. Wieclawski, M. Perkowski, "A Cost Function for Layout in A Boolean Level Silicon Compiler", Proc. Midwest 86 Conference on Circuits and Systems, Lincoln, Nebraska, 1- 12 August 1986.

[505] A. Wieclawski, M. Perkowski, "Optimization of Negative Gate Networks Realized in Weinberger-like layout in a Boolean Level Silicon Compiler". Proceedings of 21st Design Automation Conference, ACM and IEEE, Albuquerque, 25 - 27 June, 1984.

[506] R.F. Wheeling, "Heuristic Search: Structural Problem," *Progress in Operations Research,* Vol. 3., John Wiley and Sons, New York, 1969.

[507] B.W. Wah, and Y.W.E. Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE TC,* Vol. C-33, No. 5, pp. 377 - 390, May 1984.

[508] S. Wrobel, "Demand-Driven Concept Formation," *Knowledge Representation and Organization in Machine Learning*, Ed. K. Morik, Springer-Verlag, Berlin Heidelberg, 1989.

[509] J. Wnek and R.S. Michalski, "Hypothesis-Driven Constructive Induction in AQ17: A Method and Experiments," *Proceedings of IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*, pp. 13–22, Sydney, Australia, 1991.

[510] J. Wnek and R.S. Michalski, "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments," *Machine Learning*, No. 14, pp. 139–168, 1994.

[511] J. Wnek and R.S. Michalski, "Hypothesis-Driven Constructive Induction in AQ17: A Method and Experiments," *Proceedings of IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*, pp. 13–22, Sydney, Australia, 1991.

[512] S. M. Weiss and C. A. Kulikowski, "Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neral Nets, Machine Learning, and Expert Systems," *Morgan Kaufmann Publishers, Inc.*, San Mateo, California, 1991.

[513] Y. Watanabe and R. Brayton, "Heuristic Minimization of Multiple-Valued Relations," *IEEE Transactions on CAD*, Vol. 12, No. 10, pp. 1458–1472, October, 1993.

[514] P. Wegner, "Programming Languages, Information Structures, and Machine organization," *McGraw-Hill*, London, 1971. (**Old but still useful.**)

[515] C. Weissman, "LISP 1.5 Primer," *Dickenson Publishing Company,* Belmont, California, 1967. (**Classic.**)

[516] R. Wilensky, "Common LISPcraft," *Norton*, 1986. (**Classic.**)

[517] Y. Wilks, "Understanding without Proofs," *Proc. IJCAI 75.*

[518] P.H. Winston, "Artificial Intelligence," *Addison-Wesley*, 1977. (**Classic.**)

[519] P. Winston, and R.M. Brown, (eds.): "Artificial Intelligence. An MIT Perspective," 1979.

[520] P. Winston, and H. Hunt, "LISP," 3rd edition, *Addison-Wesley Publishing Company*, 1988. (**Classic.**)

[521] D. Woolridge, "An Algebraic Simplify Program in LISP," *Artificial Intelligence Project*, Stanford University, Memo 11, December 1963. (**Very useful for those interested in writing symbolic simplifiers.**)

[522] D.A. Waterman, "Adaptive Production Systems," *Proc. IVth A.I. Conference*, Tbilisi, USSR, Georgia, pp. 296-303, 1975.

[523] R.B. Wesson, "Planning in the World of the Air Traffic Controller," *Proc. Vth IJCAI*, pp. 473-479, 1977.

[524] P.H. Winston, "Artificial Intelligence," *Addison-Wesley Publishing Company*, Reading, Massachusetts, 1976. (**Reading the subsequent editions of this book will help you to understand evolution of AI.**)

[525] Neil H. E. Weste and Kamran Eshraghian, "Principles of CMOS VLDI Design – a system perspective," *Addison-Wesley*, second edition.

[526] Vanderheiden, G., "Immediately Implementable Strategies for Providing Full Access to Microcomputers for Severely Physically Handicapped Users," *Proc. IEEE Computer Society Workshop on the Computing and the Handicapped,* IEEE Comp. Soc. Press, 1982, pp. 65-68.

[527] E.N. Vavilov, and G.P. Portnoy, "Synthesis of Circuits of Electronic Computers," *Energia Publishers*, Moscov, 1966 (in Russian).

[528] S. Vere, "Relational Production Systems", *Artificial Intelligence,* Vol. 8., pp. 47-68, 1977. (**Interesting ideas, is anybody using them in EDA?**).

[529] S.A. Vere, "Induction of Relational Productions in the Presence of Background Information," *Proc. Vth IJCAI,* pp. 349-355, 1977.

[530] F. Vasko, and F. Wolf, "Solving large set covering problems on a personal computer," *Comput. Oper. Res.,* Vol. 15., No. 2., Febr. 1988, pp. 115-121.

[531] M. Yamamoto, "A Method for Minimizing Incompletely Specified Sequential Machines," *IEEE Trans. Comp.*, Vol C-29, No. 8, p. 732-736, August 1980.

[532] F. Yeaple, "Microcomputer-controlled robot inspects nuclear plant," *Design News*, Vol. 43, March 1987, p.98.

[533] H. Yasura, T. Tsujimoto, and K. Tamaru, "Parallel Exhaustive Search For Several NP-Complete Problems Using Content Addressable Memories," *Proc. IEEE Intern. Conference on Circuits and Systems, ISCAS'88,* pp. 333-336.

[534] C. Yu, and B. Wah, "Learning dominance relations in combinatorial search problems," *IEEE Trans. Software Engng.,* Vol. 14., No. 8., August 1988, pp. 1155-1175.

[535] A.A. Zakrevskij, "Algorithms of Discrete Automata Synthesis," *Nauka*, Moscow, 1971 (in Russian).

[536] Y.Z. Zhang, P.J.W. Rayner, "Minimisation of Reed-Muller Polynomials with Fixed Polarity", IEE Proceedings, Vol. 131, Pt.E, No. 5, September 1984.

[537] B. Zupan, "Machine Learning Based on Function Decomposition," University of Ljubljana, 1997.

[538] R. Zabih, D. McAllester, and D. Chapman, "Non-Deterministic Lisp with Dependency-Directed Backtracking," *Proceedings of the AAAI*, 1987. (**Very useful for those interested in backtracking and non-deterministic programming.**)

[539] Zakrevskij, A.A.: "Algorithms of Discrete Automata Synthesis" Nauka, Moscow, 1971 (in Russian).

[540]

[541] J.C. Hosseini, R.R. Harmon, and M. Zwick, "An Information Theoretic Framework for Exploratory Multivariate Market Segmentation Research," *Decision Sciences,* 1991, Vol. 22, No. 3, pp. 663-677.

[542] C. Matheus, "Feature Construction: An Analytic Framework and Application to Decision Trees," University of Illinois, PhD Dissertation, Urbana-Champaign, 1989.

[543] J. Wnek and R.S. Michalski, "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments," *Machine Learning*, No. 14, pp. 139–168, 1994.

[544] J.B. Larson and R.S. Michalski, "Inductive Inference of VL Decision Rules," *ACM SIGART Newsletter*, No. 63, pp. 38–44, 1977.

[545] D.B. Lenat, "On Automated Scientific Theory Formation: A Case Study Using AM Program," *Machine Intellience*, Ed. J.E. Hayes, D. Mitchie and L.I. Mikulich, Vol. 9, Halsted Press, New York, 1977.

[546] D.B. Lenat, "Learning from Observation and Discovery," *Machine Learning: An Aritficial Intellience Approach*, Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.

[547] G. Drastal, G. Czako and S. Raatz, "Induction in an Abstraction Space: A Form of Constructive Induction," *Proceedings of the IJCAI-89*, pp. 708–712, Morgan Kaufmann, Detroit, MI, 1989.

[548] M.M. Kokar, "Discovering Functional Formulas Through Changing Representation Base," *Proceedings of the AAAI-86*, pp. 455–459, Philadelphia, PA, 1986.

[549] R.S. Michalski, "Pattern Recognition as Knowledge-Guided Computer Induction," *Technical Report No. 927*, Department of Computer Science, University of Illinois, Urbana-Champaign, 1978.

[550] R.S. Michalski, "A Theory and Methodology of Inductive Learning," *Machine Learning: An Artificial Intelligence Approach*, Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.

[551] T.M. Mitchell, P.E. Utgoff and R. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning: An Artificial Intelligence Approach*, Ed. R.S. Michalski and J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1983.

[552] P.E. Utgoff, "Shift of Bias for Inductive Concept Learning," Rutgers University, PhD Dissertation, 1984.

[553] P.E. Utgoff, "Shift of Bias for Inductive Concept Learning, *Machine Learning: An Artificial Intelligence Approach*, Vol. 2, Ed. R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Morgan Kaufmann, Los Altos, CA, 1986.

[554] S. Muggleton, "Duce, and Oracle-based Approach to Constructive Induction," *Proceedings of IJCAI-87*, pp. 287–292, Milan, Italy, Morgan Kaufmann, 1987.

[555] S. Muggleton and W. Buntine, "Machine Invention of First Order Predictes by Inverting Resolution," *Proceedings of the 5th International Conference on Machine Learning*, pp. 339–352, Ann Arbor, MI, Morgan Kaufmann, 1988.

[556] C.A. Knoblock, "A Theory of Abstraction for Hierarchical Planning," Change of Representation and Inductive Bias, Ed. D.P. Benjamin, Kluwer Academic, Boston, MA, 1990.

[557] C.A. Knoblock, S. Minton and O. Etzioni, "Integrating Abstraction and Explanation Based Learning in PRODIGY," *Proceedings of AAAI-91*, pp. 541–546, AAAI Press/MIT Press, 1991.

[558] L. De Raedt and M. Bruynooghe, "Constructive Induction by Analogy," *Proceedings of EWSL-89*, pp. 189–200, Pitman, Montpellier, France, 1989.

[559] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba and L. Jozwiak, "Cube Diagram Bundles: a new representation of strongly unspecified multiple-valued functions and relations," *Proc. of ISMVL'97*, Halifax, Nova Scotia, Canada, pp. 287–292, May 28-30, 1997.

[560] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Zhi Wang, and Jin S. Zhang, "Decomposition of multiple-valued relations," *Proc. of ISMVL'97*, Halifax, Nova Scotia, Canada, pp. 13–18, May 28-30, 1997.

[561] H. A. Curtis, "A New Approach to the Design of Switching Circuits," *Van Nostrand*, Princeton, 1962.

[562] M. DesJardin and D. F. Gordon, "Evaluation and Selection of Biases in Machine Learning," *Machine Learning Journal*, Vol. 20, pp. 5–21, 1995.

[563] S. M. Weiss and C. A. Kulikowski, "Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neral Nets, Machine Learning, and Expert Systems," *Morgan Kaufmann Publishers, Inc.*, San Mateo, California, 1991.

[564] S. Grygiel, M. Perkowski, P. Burkey and M. Burns, "New representation for Storing and Decomposition of MV Relations," *Design Automation Conference, unublished*, 1997.

[565] S. Muggleton and C. Feng, "Efficient induction of logic programs," *Proc. First Conf. on Algorithmic Learning Theory*, Ed. S. Arikawa, S. Goto, S. Ohsuga and T. Yokomori, Tokyo, Japan, Soc. Art. Intell., pp. 368–381, 1990.

[566] M.J.E. Sternberg, R.D. King, R.A. Lewis and S. Muggleton, "Application of machine learning to structural molecular biology," *Phil. Trans. R. Soc. Lond. B*, Vol. 344, pp. 365–371, 1994.

[567] D. Mitchie, "Problem Decomposition and the Learning of Skills," *Proceedings of the Europeean Conference on Machine Learning, Lecture Notes in Artificial Intelligence*, Vol. 912, Spriger Verlag, Berlin, Heidelberg, New York, Inc., pp. 17–31, 1995.

[568] Saunders, R., "The thallium diagnostic workstation: Learning to diagnose heart imagery from examples," *Proceedings of the Second Innovative Applications of Artificial Intelligence Conference*, pp. 105–118, Menlo Park, CA, AAAI Press, 1991.

[569] B. Evans, and D. Fisher, "Overcoming process delays with decision-tree induction," IEEE Expert, 1994, No.9, pp. 60–66,

[570] U.M. Fayyad, P. Smyth, N. Weir and S. Djorgovski, "Automated analysis and exploration of image databases: Results, progress, and challenges," *Journal of Intelligent Information Systems*, 1993, No. 4, pp. 1–19,

[571] A. Giordana, R. Neri and L. Saitta, "Fielded Applications of Machine Learning," San Francisco CA: Morgan Kaufmann, P. Langley and Y. Kodratoff, 1997,

[572] M. Goodman, "Fido: The shopping doggie!," 1996,

[573] C. Guilfoyle, "Ten minutes to lay the foundations," *Expert Systems User*, August, 1986, pp. 16–19,

[574] N. Karba, and R. Drole, "Expert systems for the cold rolling mill of the Steel Works Jesenice," *In Proceedings of the Thirteenth Symposium on Information Technologies*, Sarajevo, Yugoslavia: Unpublished, 1989.

[575] W.J. Leech, "A rule-based process control method with feedback," *Advances in Instrumentation*, No. 41, pp. 169–175, 1986.

[576] D. Michie, "Problems of computer-aided concept formation," Ed. R. Quinlan, *Applications of Expert Systems (Vol. 2)*, 1989, Wokingham, UK, Addison-Wesley.

[577] P. Riddle, R. Segal, and O. Etzioni, "Representation design and brute-force induction in a Boeing manufacturing domain," *Applied Artificial Intelligence*, 1994, No. 8, pp. 125–148,

[578] C. Riese, "Transformer fault detection and diagnosis using RuleMaster by Radian," Austin, TX, Radian Corporation, 1984.

[579] S. Salzberg, R. Chandar, H. Ford, S.K. Murthy and R. White, "Decision trees for automated identication of cosmic-ray hits in Hubble Space Telescope images," *Publications of the Astronomical Society of the Pacific*, No. 107, pp. 279–288, 1995.

[580] S. Slocombe, K. Moore, and M. Zelouf, "Engineering expert system applications," *Presented at the British Computer Scociety Annual Conference*, 1986.

[581] M. DesJardins and D. F. Gordon, "Evaluation and Selection of Biases in Machine Learning," *Machine Learning Journal*, Vol. 20, pp. 5–21, 1995.

[582] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Comput.*, Vol. C-35, No. 8, pp. 667–691, 1986.

[583] R.E. Bryant, "On the complexity of VLSI implementation and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. on Computers*, Vol. 40, pp. 205–213, 1991.

[584] K.S. Brace, R.L. Rudell and R.E. Bryant, "Efficient Implementation of a BDD Package," *Proc. of 27th Design Automatiom Conference*, pp. 40–45, June, 1990.

[585] R. Brayton and F. Somenzi, "An Exact Minimizer for Boolean Relations," *Proc. of ICCAD*, pp. 316–320, 1989.

[586] D. L. Dietmeyer, "Logic Design of Digital Systems," Allyn and Bacon, Boston, MA, 1971.

[587] Y. T. Lai, K.R. Pan, M. Pedram, and S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-up Table Type FPGA Synthesis," *Proc. 30-th DAC*, pp. 642–647, 1993.

[588] T. Luba and J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Objects Reduction," *Foundation of Computing and Decision Sciences*, Vol. 18, No. 3–4, pp. 241–258, 1993.

[589] T. Luba, M. Mochocki and J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bulletin of the Polish Academy of Sciences, Technical Sciences*, Vol. 41, No. 3, 1993.

[590] T. Luba, R. Lasocki, and J. Rybnik, "An Implementation of Decomposition Algorithm and its Application in Information Systems Analysis and Logic Synthesis," *Intern. Workshop on Rough Sets and Knowledge Discovery*, pp. 487–498, Banff, 1993.

[591] T. Luba, "Decomposition of Multiple-Valued Functions," *Proc. 25th ISMVL*, pp. 256–261, 1995.

[592] S. Minato, "Graph-Based Representations of Discrete Functions," *Proc. Reed-Muller'95 Workshop*, pp. 1–10, Chiba, Japan, August, 1995.

[593] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. Reed-Muller'95 Workshop*, pp. 102–109, Chiba, Japan, August, 1995.

[594] T. Sasao, "FPGA Design by Generalized Functional Decomposition," *Logic Synthesis and Optimization*, Ed. T. Sasao, Kluwer Academic Publishers, pp. 233–258, 1993.

[595] V.Y. Shen, A. C. McKellar, and P. Weiner, "An Fast Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Trans. on Comput.*, Vol. C-20, No. 3, pp. 304–309, March, 1971.

[596] W. Wan and M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. Euro-DAC*, pp. 230–235, 1992.

[597] Y. Watanabe and R. Brayton, "Heuristic Minimization of Multiple-Valued Relations," *IEEE Transactions on CAD*, Vol. 12, No. 10, pp. 1458–1472, October, 1993.

[598] H. Selvaraj, A. Czerczak, A. Krasniewski, and T. Luba, "Generalized Decomposition of Boolean Functions and its Application in FPGA-based Synthesis," *IFIP Workshop on Logic and Architecture Synthesis*, pp. 147–166, Grenoble, France, 1993.

[599] J. Gilbert and L. Gilbert, "Elements of Modern Algebra," *Prindle, Weber & Schmidt*, Boston, 1984.

[600] B. Zupan, "Machine Learning Based on Function Decomposition," University of Ljubljana, 1997.

[601] S. Devadas, "Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions," *IEEE Trans. on CAD*, Vol. 12, No. 5, pp. 722–723, May, 1993.

[602] Steinbach, Hesse, Kempe, Rhode and Barthel, "Papers and discussions at the 2nd Workshop Boolesche Probleme," Freiberg, Germany, 19-20 September, 1996.

[603] S. Grygiel and M. Perkowski, "Multiple-Valued Incompletely Specified Functions and Relations. Part I: Representation," 1997.

[604] M. Perkowski and S. Grygiel, "Multiple-Valued Incompletely Specified Functions and Relations. Part II: Decomposition," 1997.

[605] A. Srinivasan, T. Kam, S. Malik, and R. Brayton, "Algorithms for discrete function manipulation," *IEEE International Conference on CAD*, pp. 92–95, 1990.

[606] R. Kohavi, "Bottom-up induction of oblivious read-once decision graphs," *Europeean Conference on Machine Learning*, 1994.

[607] A.L. de Oliveira, "Inductive Learning by Selection of Minimal Complexity Representations, *University of California at Berkeley*, 1994.

[608] J.R. Quinlan, "Induction of decision trees," *Machine Learning*, No. 1, pp. 81–106, 1986.

[609] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *IEEE Transactions on Computers*, Vol. C-30, pp. 635–643, September, 1981.

[610] Y.H. Su and P.T. Cheung, "Computer minimization of multiple-valued switching functions," *IEEE Transactions on Computers*, Vol. C-21, pp. 995–1003, 1972.

[611] Shmerko, Jozwiak and industry, Private communication, 1996.

[612] R.S. Michalski and J.B. Larson, "Inductive Inference of VL Decision Rules," *Workshop in Pattern-Directed Inference Systems*, Hawaii, May, 1977.

[613] U.C. Irvine, "U.C. Irvine, Repository of Machine Learning Databases and Domain Theories," `ftp://ftp.ics.uci.edu/pub/ machine-learning-databases/`

[Brayton 93] R. Brayton, P.C. McGeer, J.V. Sanghavi, and A.L. Sangiovanni-Vincentelli, "A New Exact Minimizer for Two-Level Logic Synthesis," *Sasao, T. (ed): "Logic Synthesis and Optimization"*, Kluwer, 1993.

[Ciesielski 89] M. J. Ciesielski, S. Yang, and M. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," *Proc. ICCD'89,* pp. 262 - 265, October 1989.

[Falkowski 91] B.J. Falkowski, and M. Perkowski, "Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," *Intern. J. of Electronics,* Vol. 70, No. 3, pp. 533 - 538, March 1991.

[Fujita 95] M. Fujita, Y. Kukimoto, and R. Brayton, "BDD Minimization by Truth Table Permutations," *Proc. IWLS '95.*

[Meinel 95] Ch. Meinel, J. Bern, and A. Slobodova, "Efficient OBDD-Based Boolean Manipulation in CAD Beyond Current Limits," *Proc. 32nd DAC,* San Francisco 1995.

[Nguyen 87] L. Nguyen, M.A. Perkowski, and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. 24th. DAC*, 1987, pp. 615-621.

[Perkowski 95b] M. Perkowski, T. Luba, S. Grygiel, M. Kolsteren, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *PSU Report*, unpublished, Version III, September 1995.

[Perkowski 95a] M. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and its Application to Multi-Level AND/OR/EXOR Synthesis," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 143-151.

[614] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "Fundamental Operations on Strongly Unspecified Multi-Valued Functions and Relations," *ISMVL'97.*

[615] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "GUD-MV: Multi-Level Decomposition of Multi-Valued Functions and Relations," *report, 1996.*

[Ross 91] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[Steinbach 95] B. Steinbach, and A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 161-168.

[616] S. Grygiel and M. Perkowski, "New compact representation of multiple-valued functions, relations, and non-deterministic state machines," *Proc. ICCD-98*, Austin, Texas, October, 1998.

[617] I. Bratko, "Private communication, University of Ljubljana, 1996.

[618] Proc. of the 29th IEEE International Symposium on Multiple-Valued-Logic, Freiburg im Breisgau, Germany, 20-22 May, 1999.

[619] M. Kameyama, "Innovation of Intelligent Integrated Systems Architecture -Future Challenge," *Proc. ULSI'99.*

[620] D. Sieling, "Nonapproximability Results for OBDD- and FBDD-Minimization," *Proc. ULSI'99.*

[621] G. Cabodi, P. Camurati, S. Quer, "Improving Reachability Analysis by means of Activity Profiles," *Proc. ULSI'99.*

[622] S. Hoereth, "A Word-Level Graph Representation Package," *Proc. ULSI'99.*

[623] K. Strehl, L. Thiele, "Interval Diagram Techniques and Their Applications," *Proc. ULSI'99.*

[624] S. Yanushkevich, and D. Simovici, "Information Theory Approach in Logic Design: Results, Trends and Non-Solved Problems," *Proc. ULSI'99.*

[625] H. Watanabe, and S. Yanushkevich, "Methods of Information Engine Theory in Simple Examples," *Proc. ULSI'99.*

[626] C. Moraga, J. Kolodziejczyk, M. Opoka, S. Yanushkevich, "Information Measure in Evolvable Algorithm for Synthesis of Combinational Circuits," *Proc. ULSI'99.*

[627] D. Popel, V. Shmerko, and S.Yanushkevich, "Information Theoretical Approach in Reed-Muller Expansion Minimization," *Proc. ULSI'99.*

[628] M. Serfati, "Multivalued Binary Relations and Post Algebras," *Proc. ISMVL'99.*

[629] D. Simovici, S. Jaroszewicz, "On Axiomization of Conditional Entropy of Functions Between Finite Sets," *Proc. ISMVL'99.*

[630] R. Stankovic, D.Milenovic, D. Jankovic, "Quaternion Groups versus Dyadic Groups in Representations and Processing of Large Switching Functions," *Proc. ISMVL'99.*

[631] T. Hanyu , H. Kimura, M. Kameyama, "Multiple-Valued Content-Addressable Memory Using Metal-Ferroelectric-Semiconductor FETs," *Proc. ISMVL'99.*

[632] C. Moraga, R. Heider, "New Lamps for Old! (Generalized Multiple-Valued Neurons)", *Proc. ISMVL'99.*

[633] E. Olson, "Supplementary Symmetrical Logic Circuit Structure," *Proc. ISMVL'99.*

[634] B. Steinbach, M. Perkowski, C. Lang, "Bi-Decomposition of Multi-Valued Functions for Circuit Design and Data Mining Applications," *Proc. ISMVL'99.*

[635] T. Sasao, "Totally Undecomposable Functions: Applications to Efficient Multiple-Valued Decompositions," *Proc. ISMVL'99.*

[636] J. Lou, and J. Brzozowski, "A Generalization of Shestakov's Function Decomposition Method," *Proc. ISMVL'99.*

[637] E. Dubrova, "Evaluation of m-Valued Fixed Polarity Generalizations of Reed-Muller Canonical Forms," *Proc. ISMVL'99.*

[638] D. Debnath, and T. Sasao, "Multiple-Valued Minimization to Optimize PLAs with Output EXOR Gates," *Proc. ISMVL'99.*

[639] T. Hozumi, O. Kakusho, and Y. Hata, "The Output Permutation for the Multiple- Valued Logic Minimization with Universal Literals," *Proc. ISMVL'99.*

[640] N. Takagi, A. Hon-nami, K. Nakashima, "Logic Model for Representing Uncertain Statuses of Multiple-Valued Logic Systems Realized by Min, Max and Literals," *Proc. ISMVL'99.*

[641] G. Dueck, M. Hu, and B. Fraser, "A Super Switch Algebra for Quantum Device based Systems," *Proc. ISMVL'99.*

[642] T. Ninomiya, and M. Mukaidono, "Clarifying the Axioms of Kleene Algebra based on the Method of Indeterminate Coefficients," *Proc. ISMVL'99.*

[643] G. Pogosyan, "The Number of Cascade Functions," *Proc. ISMVL'99.*

[644] D. Basin, "Automata, Circuits and BDDs," *Proc. ISMVL'99.*

[645] M. Abd-El-Barr, and H. Fernandes, "Synthesis of Multiple-Valued Decision Diagrams using Current-Mode CMOS Circuits," *Proc. ISMVL'99.*

[646] H. Md. H. Babu, and T. Sasao, "Shared Multiple-Valued Decision Diagrams for Multiple-Output Functions," *Proc. ISMVL'99.*

[647] R. Stankovic, "Matrix-Valued EXOR-TDDs in Decompositon of Switching Functions," *Proc. ISMVL'99.*

[648] A. Herrfeld, S. Hentschke, "Ternary Multiplication Circuits Using 4-Input Adder Cells and Carry Look-Ahead," *Proc. ISMVL'99.*

[649] J. Shen, K. Tanno, and O. Ishizuka, "Down Literal Circuit with Neuron-MOS Transistors and Its Applications," *Proc. ISMVL'99.*

[650] A.Saed, M. Ahmadi, and G. Jullien, "Arithmetic Circuits for Analog Digits," *Proc. ISMVL'99.*

[651] K. Freitag, L. Hildebrand, C. Moraga, "Quaternary Coded Genetic Algorithms," *Proc. ISMVL'99.*

[652] T. Aoki, K. Hoshi, and T. Higuchi, "Redundant Complex Arithmetic and Its Application to Complex Multiplier Design," *Proc. ISMVL'99.*

[653] A. Ngom, I. Stojmenovic, and J. Zunic, "On the Number of Multilinear Partitions and the Computing Capacity of Multiple-Valued Multiple-Threshold Perceptrons," *Proc. ISMVL'99.*

[654] Y. Nagata, M. Miller, and M. Mukaidono, "B-ternary Logic Based Asynchronous Micropipeline," *Proc. ISMVL'99.*

[655] K. Adams, J. Campbell, L. Maguire, J. Webb, "State Assignment Techniques in Multiple-Valued Logic," *Proc. ISMVL'99.*

[656] L. Jozwiak, "Information Relationships and Measures in Application to Logic," *Proc. ISMVL'99.*

[657] T. Lukasiewicz, "Probabilistic and Truth-Functional Many- Valued Logic Programming," *Proc. ISMVL'99.*

[658] M. Abd-El-Barr, M. Al-Sharif, and M. Osman, "Fault Characterization and Testability Considerations in Multi-Valued Logic Circuits," *Proc. ISMVL'99.*

[659] U. Kalay , M. Perkowski, and D. Hall, "Highly Testable Group Based Logic Circuits," *Proc. ISMVL'99.*

[660] T. Hanyu , T. Ike, and M. Kameyama, "Self-Checking Multiple-Valued Circuit Based on Dual-Rail Current-Mode Differential Logic," *Proc. ISMVL'99.*

[661] H. Thiele, "On the Concept of Qualitative Fuzzy Sets," *Proc. ISMVL'99.*

[662] Y. Hata, and M. Mukaidono, "On Some Classes of Fuzzy Information Granularity and Their Representations," *Proc. ISMVL'99.*

[663] Univ of Ulster, "From a Fuzzy Flip-Flop to a MVL Flip-Flop," *Proc. ISMVL'99.*

[664] E. Bloedorn and R.S. Michalski, "Data Driven Constructive Induction in AQ17-PRE: A Method and Experiments, *Proceedings of the Third International Conference on Tools for AI*, San Jose, CA, 1991.

[665] Ch. Babbage, "Passages from the Life of a Philosopher," *Longman, Green,*, London, 1864. Reprinted in 1968 by Dawsons of Pall Mall (London). (**This is a book from Babbage, the first designer of computers ever. Great motivation booster from a man that was born too early and never completed his work**).

[666] J.F. Baldwin, "A model of fuzzy reasoning through multi-valued logic and set theory," *Intern. J. of Man-Machine Studies,* Vol. 11, 1979, pp. 351-380, No. 5.

[667] E.T. Bell, "Men of Mathematics," New York: Simon & Schuster, 1965. (**Interesting and encouraging stories of great mathematicians.**).

[668] M. Boden, "Artificial Intelligence and Natural Man," New York, Basic Books, 1977. (**Easy introduction to AI and thinking from a well-known author.**).

[669] L. Bolc, P. Borowik, "Many Valued Logics - 1. Theoretical Foundations," *Springer*, 1992.

[670] J.T. Butler, "Multiple-Valued Logic in VLSI," *IEEE Computer Society Press*, 1991. (**Papers' collection: practical approach, mostly to circuit design.**).

[671] Clare, book (**"Industrial" book about state machines, used internally in Intel for many years. Good examples. No theory**).

[672] G. DeMicheli, book (**One of modern graduate textbooks on logic synthesis. Can be read by undergraduates. Good examples.**)

[673] S. Devadas, et al. book (**This book is not as easy to read as DeMicheli and has errors. Uniform presentation of recent and advanced material.**)

[674] G. Epstein

[675] D. Gabbay, F. Guenther (Eds.), "Handbook of Philosophical Logic, Vol. III: Alternatives in Classical Logic," *D. Reidel Publishing Company,* 1986

[676] G.B. Gerace, et al, "TOPI-A Special-Purpose Computer for Boolean Analysis and Synthesis," *IEEE TC,* Vol. C-20, pp. 837-842, Aug. 1971. (**Gerace was a M.S. student of Svoboda and built one of rare computers for logic. We will build some computers like that in this book, too.**)

[677] R.L. Goodstein, "Development of Mathematical Logic," New York, Springer Verlag, 1971. (**To learn about mathematical logic, for those who know computer logic.**)

[678] S. Gottwald, "Fuzzy Sets and Fuzzy Logic: The Foundations of Application from a Mathematical Point of View," *Teknea: Vieweg,* 1993.

[679] XXXX. The paper by Grasselli and Luccio on 2D state minimization.

[680] XXXX. G. Hachtel, book (**This is my favourite logic synthesis textbook, a good companion for this book.**)

[681] P. Hajek, "Metamathematics of Fuzzy Logic," Trends in Logic, Vol. 4, 308 pp. August 1998, *Kluwer Academic Publishers.*

[682] M. Helliwell, and M.A. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms," *Proc. 25-th ACM/IEEE Design Automation Conference,* paper 28.2, pp. 427-432, June 12- June 15, 1988. (**Helliwell was a sophomore when he wrote the paper for DAC conference about ESOP minimization. We hope you can too ;-). The success of him and other of our undergraduates stimulated us to develop an idea of advanced "problem-solving" textbook for undergraduates; this book**).

[683] W.D. Hillis, "The Connection Machine," The MIT Press, Cambridge, Massachusetts. (**Hillis was a MIT student when he started his supercomputer company. This book presents his computer and more. Though provoking ideas on what is computing.**).

[684] P.M. Ho, and M.A. Perkowski, "Systolic Architecture for Solving NP-Hard Combinational Problems of Logic Design and Related Areas," *Proc. ISCAS'89.* (**Simple design of Machine to Solve Satisfiability Problem. We will simulate it and build it.**)

[685] D.R. Hofstadter, "Goedel, Escher, Bach: An Eternal Golden Braid," Vintage Books, 1980. (**Excellent book for people interested in philosophy, human thinking, and their relation to computers. A Classic!**)

[686] S.J. Hong, R.G. Cain, and D.L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. Develop.,* Vol. 18, pp. 443 - 458, Sept. 1974. (**The first really successful two-level SOP minimizer, from IBM. This paper is a classic in logic design. Compare it with Brayton's book to appreciate the recent progress in this area.**)

[687] S.L. Hurst, "Multiple-Valued Logic - Its Status and its Future," *IEEE Tr. Computers,* Dec. 1984, pp. 1160-1179.

[688] P.C. Jackson, "Introduction to Artificial Intelligence," New York, Petrocelli Charter, 1975. (**One of "easy" AI books for beginners. Not technical, but explains ideas.**)

[689] D. Johnson, "The NP-Completeness Column: An Ongoing Guide," *Journal of Algorithms,* Academic Press, each issue. (**A growing collections of NP-Complete Problems.**)

[690] S. Kleene, "Introduction to Mathematical Logic," New York, John Wiley, 1967.

[691] Z. Kohavi, book (**A classic in Finite Machines and Logic Synthesis. Now obsolete, but still interesting.**)

[692] A. Koestler, "The Art of Creation," New York: Dell, 1966.

[693] Y.S. Kuo, "Generating essential primes for a Boolean function with multiple-valued inputs," *IEEE Trans. on Computers,* Vol. C-36, pp. 356-359, March 1987. (**An easy starter on MV logic minimization.**)

[694] K. Morik, "Sloppy Modeling," *Knowledge Representation and Organization in Machine Learning,* Ed. K. Morik, Springer-Verlag, Berlin Heidelberg, 1989.

[695] G. Malinowski, Many-valued logics, *Oxford, England, Clarendon Press; Oxford University Press,* 1993. SERIES :Oxford logic guides.

[696] M.A. Marin, "Investigation of the Field of Problems for the Boolean Analyzer," *Ph.D. Dissertation,* Univ. of California, Los Angeles, 1971. (**Another thesis of Svoboda's student.**)

[697] Mayer, Mechler, Schlindwein, Wolke: Fuzzy Logic, *Addison Wesley*, 1993.

[698] J.T. McCall, J.G. Tront, F.G. Gray, R.T. Haralick, and W.M. McCormack, "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem," *IEEE TC.,* Vol. C-34, No. 11, Nov. 1985. (**Interesting example of computers for image processing and other consistent labeling problems.**)

[699] J. Michael, "Validation, Verification, and Experimentation with Abacus 2," *Reports of Machine Learning and Inference Laboratory, MLI 91-8*, Center for Artificial Intelligence, George Mason University, Fairfax, VA, 1988.

[700] P. Morrison, and E. Morrison, (Eds.), "Charles Babbage and His Calculating Engines," New York: Dover Publications, 1961. (**If you are interested in history of computing. See the origins of great ideas.**)

[701] J.C. Muzio and T.C. Wesselkamper, "Multiple-Valued Switching Theory," *Hilger,* 1986.

[702] J.C. Muzio, S. L. Hurst and D. M. Miller, "Spectral Techniques in Digital Logic," *Academic (Press?)*, 1985.

[703] J. Nievergelt, J.C. Farrar, and E.M. Reingold, "Computer Approaches to Mathematical Problems," Englewood Cliffs, N.J., *Prentice Hall*, 1974. (**Now a little bit old, but still useful.**)

[704] G. Pagallo and D. Haussler, "Boolean Feature Discovery in Empirical Learning," *Machine Learning*, No. 5, pp. 71–99, 1990.

[705] M.A. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proc. ICCAD'85,* pp. 133-135. (**A source of early ideas on logic computers. Useful in projects.**)

[706] M.A. Perkowski, M. Helliwell, and P. Wu, "Minimization of Multiple-Valued Input Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," *Proc. ISMVL-89,* (**Some project ideas on using MV logic and EXOR synthesis.**)

[707] S.R. Petrick, "On the Minimization of Boolean Functions," *Proc. Symp. on Switch. Theory,* IFIP, Paris, June 1959. (**The idea of Petrick function used in many of our projects.**)

[708] Polya, The famous books by Polya How to Solve it? and other.

[709] D.K. Pradhan, "A Theory of Galois Switching Functions," pp. 239-248, *IEEE Tr. Computers,* Volume 27, Number 3, March 1978.

[710] V. Pratt, "Thinking Machines. The Evolution of Artificial Intelligence," *Basil Blackwell, Inc.,* Oxford, U.K., 1987. (**Easy and interesting reading. One of few books about history of logic computers.**)

[711] N. Rescher, "Many-valued Logic," New York, *McGraw Hill,* 1969.

[712] D.C. Rine (ed), "Computer Science and Multiple-Valued Logic: Theory and Applications," Amsterdam, The Netherlands, *North-Holland,* 1977.

[713] P. Roth, *"Computer Logic, Testing and Verification,"* Rockville, MD: Computer Science, 1980. (**A classic book in logic synthesis theory and good introduction of cube calculus.**)

[714] R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "ESPRESSO-MV: algorithms for multiple-valued logic minimization," *Proc. IEEE Custom Integrated Circuits Conf.,* 1985. (**A top SOP minimizer programmed by a graduate student from U.C.Berkeley. Now used by many companies in their EDA software.**)

[715] R.L. Rudell, *"Multiple-Valued Logic Minimization for PLA Synthesis,"* M.S. Report, June 5, 1986. University of California, Berkeley California 94720. (**One of the most useful M.S. these written.**)

[716] R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *Proc. Intern. Symp. on Multiple-Valued Logic,* pp. 198-208, May 26-28, Boston, MA, 1987.

[717] T. Sasao, "An application of multiple-valued logic to a design of Programmable Logic Arrays," *Proc. 8th Intern. Symp. on Multiple-Valued Logic (ISMVL),* 1978. (**A useful introduction to MV logic in practical application.**)

[718] T. Sasao, "Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays," *IEEE Trans. Comput.,* Vol. C-30, pp. 635-643, Sept. 1981. (**Another useful paper from the same author.**)

[719] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. Comput.,* Vol. C-33, pp. 879 - 984, Oct. 1984. (**One more paper about two problems important in PLA optimization.**)

[720] T. Sasao, "HART: A Hardware for Logic Minimization and Verification," *Proc. ICCD'85,* Oct. 7-10, 1985. (**Sasao's design of logic computer. Built in hardware using PALs. Good project ideas**.)

[721] T. Sasao, "MACDAS: Multi-level AND-OR circuit synthesis using two-variable function generators," *23-rd Design Automation Conference,* Las Vegas, pp. 86-93, June 1986.

[722] D. Shumake, "The MOS Boolean Analyzer," *M.Sc. Thesis,* UCLA, 1971. (**Another M.S. Thesis of Svoboda's student.**)

[723] A. Svoboda, "Boolean Analyzer," *Proc. Information Processing 68,* Amsterdam, North-Holland, 1969. (**The main paper by Svoboda on his logic computer.**)

[724] A. Svoboda, "Parallel Processing in Boolean Algebra," *IEEE TC,* Vol. C-22, No. 9, pp. 848-851, Sept 1973. (**Another useful paper by Svoboda.**)

[725] P. Suppes, "Introduction to Logic," New York: Van Nostrand Reinhold, 1957.

[726] S. Turing, "Alan M. Turing," Cambridge, U.K.: W. Heffer & Sons, 1959. (**Mother writes about her genius son. Turing has done one of the most important contributions to computing and helped to build code breaking computers during the WW II. Code breaking computers are another great source of design ideas.**)

[727] E. Turunen, "Mathematics Behind Fuzzy Logic," Coming August 1999, from *Springer Verlag.*

[728] M.E. Ulug, and B.A. Bowen, "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems," *IEEE TC,* pp. 255-267, March 1974. (**Formal analysis of cube calculus. May be useful in projects.**)

[729] M.E. Ulug, "VLSI Knowledge Representation Using Predicate Logic and Cubical Algebra," *Proc. IEEE Intern. Conference on Computers and Communications*, Arizona, 1985, pp. 292-297. (**Interesting ideas that can be used to build logic computers.**)

[730] M.E. Ulug, "Application of Cubical Array Operators to a Relational Database," *Proc. of the Minnowbrook Workshop,* July 23-26, 1985, Blue Mountain Lake, New York. (**More interesting ideas from Ulug.**)

[731] M.E. Ulug, "A Real-Time AI System for Military Communications," *Proc. of the Third IEEE Conference on Arificial Intelligence Applications*, February 1987, Orlando, Florida. (**Continuation on these ideas, from Ulug.**)

[732] B.W. Wah, and Y.W.E. Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE TC,* Vol. C-33, No. 5, pp. 377 - 390, May 1984. (**Another computer for fast searching in hardware.**)

[733] T.C. Wesselkamper, "Divided Difference Methods for Galois Switching Functions," pp. 232-238, *IEEE Tr. Computers,* Volume 27, Number 3, March 1978.

[734] H. Yasura, T. Tsujimoto, and K. Tamaru, "Parallel Exhaustive Search For Several NP-Complete Problems Using Content Addressable Memories," *Proc. IEEE Intern. Conference on Circuits and Systems,* ISCAS'88, pp. 333-336. (**Another approach to search in hardware - using CAMs.**)

[735] R.G. Bennetts, and D. Lewin, "Fault diagnosis of digital systems - a review," *Comput. J.,* Vol. 14., pp. 199-206, 1971.

[736] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otten, and J. Van Eijndhoven, "The Yorktown Silicon Compiler System," Chapter 7 in Gajski, D., (ed), *Silicon Compilation,* 1987.

[737] W.C. Carter, A. Wadia, and D.C. Jessep, Jr., "Implementation of Checkable Acyclic Automata by Morphic Boolean Functions," *Proc. of the Symp. on Comp. and Automata*, Polytechnic Institute of Brooklyn, p. 645, 1971.

[738] M. Davio, J.P. Deschamps, and A. Thayse, "Discrete and Switching Functions," *McGraw-Hill Book Co., Inc.*, New York, 1978.

[739] H. Fujiwara, "Logic Testing and Design for Testability," Computer System Series, *The MIT Press,* 1986.

[740] D.H. Green, and K.R. Dimond, "Polynomial representation of nonlinear feedback shift-registers," *Proc. IEE,* 117, No. 1., pp. 56-60, 1970.

[741] D. Green, "Modern Logic Design," *Electronic Systems Engineering Series,* 1986.

[742] S. Hurst, "Logical processing of digital signals," *Edward Arnold,* London: *Crane-Russak,* N.Y., 1978.

[743] M. Perkowski, "Digital Design Automation System DIADES. Documentation," Dept. EE, Portland State University, Portland, OR, 1988.

[744] D.K. Pradhan, "Fault-Tolerant Computing. Theory and Techniques. Vol. I," *Prentice-Hall*, 1987.

[745] P. Roth, "Computer Logic, Testing and Verification," *Computer Science,* Rockville, MD, 1980.

[746] R. Rudell, "Multiple-Valued Logic Minimization for PLA Synthesis," *M.S. Report,* June 5, 1986. University of California, Berkeley, California 94720.

[747] R.K. Brayton, G. D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Boston, MA, Kluwer, 1984.

[748] M.A. Perkowski, and A. Zasowska, "Minimal area MOS asynchronous automata," *Proceedings of the International Symposium on Applied Aspects of Automata Theory,* Varna, Bulgaria, 14-19 May 1979, pp. 284-298.

[749] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R.Malvi, Z. Wang, and J.S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97 Conference*, pp. 13 - 18.

[750] M. Perkowski, L. Jozwiak, "Two-Dimensional Minimization of Nondeterministic State Machines," *Proc. of CAD Conference,* Belarus, Nov. 1997,

[751] M. Perkowski, L. Jozwiak, "Simultaneous Minimization, Decomposition and Encoding of Nondeterministic State Machines," *Proc. of CAD Conference,* Belarus, Nov. 1997,

[752] E.B. Lee, and M.A. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proceedings of the 1984 Intern. Conf. on Systems, Man, and Cybernetics,* IEEE, Halifax, Nova Scotia, Canada, October 9-12, 1984.

[753] G. DeMicheli, R.K. Brayton, A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Trans. on CAD,* Vol. CAD-4, No. 3, July 1985, pp. 268-284.

[754] G. DeMicheli, "Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-Level Logic Macros," *IEEE Trans. on CAD,* Vol. CAD-5, No. 4., Oct. 1986, pp. 597-616.

[755] A.A. Korbut, and J.J. Finkelstein, "Discrete Programming." *Polish Scientific Publishers*, Warsaw 1974.

[756] A. Newell, and H.A. Simon, "Human Problem Solving," *Prentice Hall,* Engl. Cliffs, New Jersey 1972.

[757] U. Peterson, K. Voss, and K.H. Weber, Math. Operationsforschung und Statistik, 5, 7-8, 1974.

[758] S. Rubin, "Computer Aids for VLSI Design," *Addison Wesley*, Reading 1987.

[759] W. Daehn, and J. Mucha, "A Hardware Approach to Self-Testing of "Large Programmable Logic Arrays," *IEEE Trans. on Computers*, Vol. C-30, No. 11, pp. 829-833, November 1981.

[760] M. Perkowski, "Digital Devices Design by Problem-Solving Transformations," *Journal on Computers and Artificial Intelligence (Pocitace a umela intelligencia)*, Vol. 1, No. 4, August 1982, pp. 343-365.

[761] M. Perkowski, "The state-space approach to the design of multipurpose problem-solver for logic design," *Proceedings of the IFIP WG.5.2 Working Conference "Artificial Intelligence and Pattern Recognition in Computer-Aided Design"*, Grenoble, France, 17-19 March 1978, J. C. Latombe (ed.) North Holland, Amsterdam, pp. 124-140, 1978.

[762] M.A. Breuer, (ed.), "Design Automation of Digital Systems," Vol.1, *Prentice Hall,* Englewood Cliffs, New Jersey, 1972.

[763] G. De Michelli, A.L. Sangiovanni-Vincentelli, and T. Villa, "Computer-Aided Synthesis of PLA-Based Finite State Machines," *Proc. IEEE 1983 ICCAD*, pp. 154-156. Sept. 1983.

[764] R. Freivald, "Probabilistic Machines Can Use Less Running Time," *IFIP 77,* North-Holland, Amsterdam, 1977, pp. 839-842.

[765] M.J. Garey, and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness," *W.H. Freeeman and Company,* San Francisco 1979.

[766] D. Johnson, "The NP-Completeness Column: An Ongoing Guide," *Journal of Algorithms,* Academic Press, each issue.

[767] S.R. Petrick, "On the minimization of Boolean Functions," *Proc. Symposium on Switching Theory,* IFIP, Paris, France, June 1959.

[768] J.D. Ullman, "Computational Aspects of VLSI," *Computer Science Press*, 1984.

[769] B. Wah, E.Y.W. Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE TC*, Vol.C-33, No. 5, May 1984, pp. 377-390.

[770] M.O. Rabin, "Probabilistic Algorithms," In J.Traub (ed.) Algorithms and Complexity: New Directions and Recent Results, *Academic Press*, New York, 1976, pp.21-39.

[771] R.A. Rutenbar, T.N. Mudge, and D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation," *IEEE TCAD*, Vol. CAD-3, No. 4., October 1984, pp. 264-278.

[772] N.N. Biswas, "Implication Trees in the Minimization of Incompletely Specified Sequential Machines," *Int. J. Electron.* (GB). No. 2, pp. 291-298, August 1983.

[773] T.A. Dolotta, and E.G. McCluskey, "The Coding of Internal States of Sequential Machines," *IEEE Trans. Electr. Comp.,* Vol EC-13, pp. 549-562, October 1964.

[774] A. Dunworth, and H.V. Hartog, "An Efficient State Minimization Algorithm for Some Special Classes of Incompletely Specified Sequential Machines," *IEEE Trans. Comp.*, Vol. C-28, No. 7, p. 531-535, July 1979.

[775] A.D. Friedman, and P. Menon, "Theory and Design of Switching Circuits," *Computer Science Press, Inc.,* Woodland Hills, California, 1975.

[776] A. Grassselli, and F. Lucio, "A Method of Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," *IEEE Trans. Electr. Comp.,* Vol. EC-14, No. 3, pp. 330-359, June 1965.

[777] G. Hallbauer, "Procedures of State Reduction and Assignment in One Step in Synthesis of Asynchronous Sequential Circuits," *Proc. Intern. IFAC Symp. on Discrete Systems,* pp. 272-282, Riga, September 30 - October 4, 1974.

[778] M. Harrison, "Introduction to Switching and Automata Theory," *McGraw-Hill*, New York, 1965.

[779] J. Hartmanis, "On the State Assignment Problem for Sequential Machines, I," *IRE Trans. Electr. Comp.,* Vol. EC-10, p. 157-165, June 1961.

[780] J. Hartmanis, and R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," *Prentice-Hall*, New York, 1966.

[781] F.C. Hennie, "Finite-State Models for Logical Machines," *John Wiley,* New York, 1968.

[782] F.J. Hill, and G.R. Peterson, "Introduction to Switching Theory and Logical Design," *John Wiley & Sons, Inc.,* 2nd Ed., New York, 1974.

[783] P.K. Lala, "An Algorithm for the State Assignment of Synchronous Sequential Circuits," *Electr. Letters,* Vol. 14, No. 6., 16 March 1978.

[784] R.M. Karp, "Some Techniques of State Assignment for Synchronous Sequential Machines," *IEEE Trans. Electr. Comp.,* Vol. EC-13 No. 5, pp. 507-518, October 1964.

[785] R.M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computation,* Plenum, ed. Miller, etc., pp. 85-103, New York, 1972.

[786] T. Kobylarz, and A. Al-Najjar, "An Examination of the Cost Function for Programmable Logic Arrays," *IEEE Trans. Comp.*, Vol. C-28, No. 8, pp.586-590, August 1979.

[787] S.C. Lee, "Digital Circuits and Logic Design," *Prentice Hall*, Englewood Cliffs, New Jersey, 1976.

[788] C. Mead, and L. Conway, "Introduction to VLSI Systems," *Addison Wesley,* 1981.

[789] N.J. Nilsson, "Problem Solving Methods in Artificial Intelligence," *McGraw-Hill,* New York, 1971.

[790] Ch. A. Papachristou, and D. Sarma, "An Approach to Sequential Circuit Construction in LSI Programmable Arrays," *IEEE Proceedings,* Vol. 130, Pt. E, No. 5, pp. 159-164, September 1983.

[791] M.C. Paull, and S.H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," *IEEE Trans. Electr. Comp.*, Vol. EC-8, 1959.

[792] G.V. Russo, G. Palama, and A.C. Neve, "Really Prime Classes Implying only Really Prime Classes," *Electr. Letters,* Vol. 19, No. 13, 23 June 1983.

[793] H.C. Torng, "Introduction to the Logical Design of Switching Systems," *Addison-Wesley*, Reading, Massachusetts, 1964.

[794] E.N. Vavilov, and G.P. Portnoy, "Synthesis of Circuits of Electronic Computers," *Energia Publishers,* Moscov, 1966 (in Russian).