CS 320: Principles of Programming Languages

Mark P Jones & Andrew Tolmach, Portland State University

Winter 2019

Week 2: Programs as Data
The computer scientist at work…

Let’s use our computer to solve a problem!

Data → Programs → Answers
It works!

Hmm, could this help me too?

But how do I make programs run?

Data → Programs → Answers
Program Analysis

- Interpreters
- Static analysis tools
- Documentation generators
- Browsers
- Testing tools
- Debuggers
- Profilers
- …

Programs → Programs → Answers
• Application template generators
• Code “wizards”
• GUI builders
• Modeling tools
• Embedded languages (e.g., dynamic web pages)
• …
Program Translation

- Compilers
- Code formatters
- Code update tools
- Macro processors
- Optimizers
- Partial evaluators
- Instrumentation
- Code editors
- ...

Programs → Programs → Programs → Programs
How do we make high-level programs run on low-level hardware?
What makes a language “high-level”?

• Complex expressions (arithmetic, logical,...)
• Structured control (loops, conditionals, cases,...)
• Composite types (arrays, records, ...)
• Type declarations and type checking
• Multiple data storage classes (global/local/heap/GC?)
• Procedures/functions (private scope, closures,...)
• Non-local control (exceptions, threads,...)
• Data abstraction (ADTs, modules, objects...)


What does hardware give us?

• Low-level machine instructions

• Control flow based on labels and conditional branches

• Explicit locations (e.g. registers) for values and intermediate results of computations

• Flat memory model

• Explicit memory management (e.g., stacks for procedure local data)
How can we bridge the gap?

High-level language

Low-level machine
Interpreters and compilers
Interpreters and compilers

In conventional English:

• **interpreter**: somebody that translates from one language to another.
  • Example: “I need an interpreter when I’m in Japan”

• **compiler**: somebody who collects, gathers, assembles, or organizes information or things.
  • Latin root: compilare, “plunder or plagiarize”
Interpreters and compilers

According to my dictionary:

• **interpreter** (noun) Computing: a program that can analyze and execute a program line by line

• **compile** (verb) Computing (of a computer): convert (a program) into a machine-code or lower-level form in which the program can be executed

Derivatives: **compiler** (noun)
Interpreters and compilers

In computer science:

• An interpreter **executes** (or runs) programs
  • An interpreter for a language $L$ might be thought of as a function: $\text{interp}_L : L \rightarrow M$, where $M$ is some set of meanings of programs

• A compiler **translates** programs
  • A compiler from a language $L$ to a language $L'$ might be thought of as a function $\text{comp} : L \rightarrow L'$

• By “language”, we mean the set of all strings that correspond to valid programs
Interpreters and compilers

• Interpreters **execute** programs (turning syntax to semantics)

  ![Diagram of an interpreter](image)

  - **input**
  - **source**
  - **output**
  - **interpreter**
  - **source**
  - **output**

• Compilers **translate** programs (turning syntax into syntax)

  ![Diagram of a compiler](image)

  - **input**
  - **source**
  - **output**
  - **program**
  - **compiler**
  - **source**
  - **output**
“Doing” vs “Thinking about doing”

• Compilers translate programs (turning syntax to syntax)
• Interpreters run programs (turning syntax to semantics)

• Example:
  • Interpreter (Doing something):
    Use your calculator to evaluate \((1+2)+(3+4)\):
    Answer: 10
  • Compiler (Thinking about doing something):
    Tell me what buttons to press to evaluate \((1+2)+(3+4)\):
    Answer:
    \[
    \begin{array}{ccccccc}
    1 & + & 2 & = & M & 3 & + & 4 & + & MR & = \\
    \end{array}
    \]
Basic terminology

source programs

many possible source languages, from traditional, to application specific languages.

target programs

usually another programming language, often the machine language of a particular computer system.
Example

source program

// A simple mini test program
int i = 0;  // initialize
while (i <= 10) {
    print i*i;  // print a square
    i = i + 1;
}

// execute

execute

semantics

compile

target program

Example source program

target program

$ ./squares
0
1
4
9
16
25
36
49
64
81
100
$
Compiler correctness

- A compiler should produce valid output for any valid input
- The output should have the same semantics as the input

In symbols: \( \forall p. \text{interp}_L(p) = \text{interp}_{L'}(\text{comp}_{L\rightarrow L'}(p)) \)
Desirable properties of a compiler

• Performance:
  • Of compiled code: time, space, power, …
  • Of the compiler: time, space, …

• Diagnostics:
  • High quality error messages and warnings to permit early and accurate diagnosis and resolution of programming mistakes
Desirable properties, continued

• Support for large programming projects, including:
  • Separate compilation, reducing the amount of recompilation that is needed when part of a program is changed
  • Use of libraries, enabling effective software reuse

• Convenient development environment:
  • Supports program development with an IDE or a range of useful tools, for example: profiling, debugging, cross-referencing, browsing, project management (e.g., make)
Compiler examples

Compilers show up in many different forms:

- Translating programs in high-level languages like C, C++, Java, etc… to executable machine code
- Just in time compilers: translating byte code to machine code at runtime
- Rendering an HTML web page in a browser window
- Printing a document on a Postscript printer
- Generating audio speech from written text
- Translating from English to Spanish/French/…
- …
Interpreter characteristics

Common (but not universal) characteristics:

- More emphasis on interactive use:
  - Use of a read-eval-print loop (REPL)
  - Examples: language implementations designed for educational or prototyping applications

- Less emphasis on performance:
  - Interpretive overhead that could be eliminated by compilation
  - Performance of scripting code, for example, is less of an issue if the computations that are being scripted are significantly more expensive
Interpreter characteristics, continued

• Portability:
  • An interpreter is often more easily ported to multiple platforms than a compiler because it does not depend on the details of a particular target language

• Experimental platforms:
  • Specifying programming language semantics
  • More flexible language designs; some features are easier to implement in an interpreter than in a compiler
Interpreter examples

• Programming languages:
  • Scripting languages: PHP, python, ruby, perl, bash, Javascript, ...
  • Educational languages: BASIC, Logo, ...
  • Declarative languages: Lisp, Scheme, ML, Haskell, Prolog, ...
  • Virtual machines: Java, Scala, C#, VB, Pascal (P-Code)

• Document description languages:
  • Postscript, HTML, ...
Interpreters and Machines

• A virtual machine is one important kind of interpreter
  • Executes programs written in a virtual (i.e. software-defined) instruction set
  • Example: Java Virtual Machine (JVM) executes (interprets) a language of byte codes
• There is no fundamental difference between this and a high-level language interpreter: both execute programs in software
• A CPU executes (machine) programs in hardware:
  • So it is a kind of interpreter too!
  • Faster, but harder to change
Another look at this question…

How can we bridge the gap?

High-level language

Low-level machine
We can compile…

High-level language

Compiler

Low-level language

Low-level machine
We can interpret…

High-level language

High-level machine

Interpreter

Low-level machine
We can do both…

High-level language

Compiler

Mid-level machine/language

Interpreter

Low-level machine
Run-time systems

• Even with a completely compiled approach, we usually need a fixed library of code available at run time, e.g. for:

  • Interfacing to the OS, e.g. to do IO

  • Managing memory, e.g. via garbage collection

  • Managing exception handlers

• This run-time system code is effectively like a (small) virtual machine layer on top of the real hardware and OS process abstraction

• Moral: Every real system involves some elements of interpretation
Language vs implementation

- Be very careful to distinguish between languages and their implementations
- C is a widely used language
- Haskell is an expressive language
- Java is a well-defined language
- Python is a slow language (NO: speed is a property of an implementation, not a language)
- C++ is a compiled language: (NO: “compiled” describes a property of an implementation, not a language)
Goals for Compiler Construction
What is a compiler?

Compilers are translators:

source programs

compiler

diagnostics

target programs
Why translation is needed

• We like to write programs at a higher-level than the machine can execute directly

  • Spreadsheet: \( \text{sum [A1:A3]} \)
  
  
  • Machine language:
    
    \[
    \begin{align*}
    \text{movl } & $0, \%eax \\
    \text{addl } & 4(a), \%eax \\
    \text{addl } & 8(a), \%eax \\
    \text{addl } & 12(a), \%eax \\
    \end{align*}
    \]

• High-level languages let us describe what is to be done without worrying about all the details

• In machine languages, every step must be carefully spelled out
Ideas:

- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

Machines:

- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

High Level

How do we turn high level ideas in to running programs on low level machines?

Low Level
Ideas:
- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

Languages:
- Evaluate an expression
- Execute a computation multiple times
- Call a function
- Save a result in a variable
- ...

Machines:
- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

High Level

Low Level
Ideas:
- Search a database
- Send a message
- Create a song
- Play a game
- etc...

Languages:
- express
- translate

Machines:
- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc...

High Level
- Evaluate an expression
- Execute a computation multiple times
- Call a function
- Save a result in a variable
- ...

Admiral Grace Hopper (1906-1992)
(Photo: via Wikipedia)

Could we program a computer to do this?

Human ingenuity required

Low Level
Could we program a computer to do this?

Yes! The A-0 system for UNIVAC I (1951-52): the first compiler

Admiral Grace Hopper (1906-1992)
(Photo: via Wikipedia)
Ideas:
- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

Languages:
- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

Machines:
- Evaluate an expression
- Execute a computation multiple times
- Call a function
- Save a result in a variable
- etc ...

High Level
- Admiral Grace Hopper (1906-1992)
  (Photo: via Wikipedia)

Low Level
- Human ingenuity required

Languages:
- Express
- Translate

Machines:
- Compiler construction

Admiral Grace Hopper (1906-1992)

(Photo: via Wikipedia)
Ideas:
- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

Languages:
- Express
- Translate

Machines:
- Human ingenuity required

High Level
- Evaluate an expression
- Execute a computation multiple times
- Call a function
- Save a result in a variable
- etc ...

Low Level
- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

Humans:
- Language design
- Compiler construction
- Human ingenuity required
Languages and tools matter

• Language designs empower developers to:
  • Express their ideas more directly
  • Execute their designs on a computer

• Better tools (compilers, interpreters, etc.) will:
  • open programming to more people and more applications
  • increase programmer productivity
  • enhance software quality (functionality, reliability, security, performance, power, ...)

42
Basics of Compiler Structure
How does a compiler work?

source program

```
// A simple mini test program
int i = 0;  // initialize
while (i <= 10) {
    print i*i;  // print a square
    i = i + 1;
}
```

target program

```
.globl _Main_main
_Main_main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $0,%eax
movl %eax,-4(%ebp)
jmp l1
l0:
movl -4(%ebp),%eax
movl -4(%ebp),%ebx
imull %ebx,%eax
movl %esp,-esp0
subl $4,%esp
addl $0xffffffff0,%esp
movl %eax,(%esp)
call _print
movl _esp0,%esp
movl $1,%eax
movl -4(%ebp),%ebx
addl %ebx,%eax
movl %eax,-4(%ebp)
l1:
movl $10,%eax
movl -4(%ebp),%ebx
cmpl %eax,%ebx
jle l0
movl %ebp,%esp
popl %ebp
ret
```

We need to describe this process in a way that is scalable, precise, mechanical/algorithmic, ...
What is this?

False

Dark pixels on a light background
A collection of lines/strokes
A sequence of characters
A single word ("token")
An expression
A boolean expression
A truth value

One thing can be seen in many different ways

We can break a complex process into multiple (hopefully simpler) steps
“Compiling” English

• The symbols must be valid:
  hdk fΩfdh ksdβs dfsjf dslkjé

• The words must be valid:
  banana jubmod food funning

• The text must use correct grammar:
  my walking up left tree dog

• Now we have preliminary abstract syntax:
  This sentence is a complete.
“Compiling” English

• The phrase must make sense
  This sentence is not true. ✗

• The phrase must not be ambiguous
  Close the window. My old friend. ✗

• The sentence must fit in context
  The next song is about geography. ✗

• Finally, we have valid abstract syntax!
  Languages are very interesting. ✓

ready for “code generation”
The compiler pipeline

• Traditionally, the task of compilation is broken down into several steps, or compilation phases:
• Turn data from a raw input source into a sequence of characters or lines
  
  Data might come from a disk, memory, a keyboard, a network, a thumb drive, ...
  
  The operating system usually takes care of most of this ...
Lexical analysis

• Convert the input stream of characters into a stream of tokens

• For example, the keyword *for* is treated as a single token, and not as three separate characters

• “lexical”:
  “of or relating to the words or vocabulary of a language”
Parser

• Build data structures that capture the underlying structure (abstract syntax) of the input program
• Determines whether inputs are grammatically well-formed (and reports a syntax error when they are not)
Static analysis

- Check that the program is reasonable:
  - no references to unbound variables
  - no type inconsistencies
  - etc...

source input → lexical analysis → parser → static analysis → validated representation

structured representation
Code generation

- Generate an appropriate sequence of machine instructions as output
- Different strategies are needed for different target machines
Optimization

- Look for opportunities to improve the quality of the output code:

  There may be conflicting ways to “improve” a given program; the choice depends on the context/the user’s priorities

  Producing genuinely “optimal” code is theoretically impossible; “improved” is as good as it gets!
The full pipeline

There are many variations on this approach that you’ll see in practical compilers:

- extra phases (e.g., preprocessing)
- iterated phases (e.g., multiple optimization passes)
- additional data may be passed between phases
Snapshots from a “mini” compiler pipeline
Snapshots from a “mini” compiler pipeline

• In this week’s labs, we’ll trace the results of passing the following program through a compiler for a language called “mini”

• A sample mini program:

```plaintext
// A simple mini test program

int i = 0;    // initialize
while (i <= 10) {
    print i*i;  // print a square
    i = i + 1;
}
```

• The goal here is just to get a sense of how compiler phases work together in practice; you don’t need to understand all of the fine details
// A simple mini test program

int i = 0;    // initialize
while (i <= 10) {
    print i*i;  // print a square
    i = i + 1;
}
Source input (as characters)

```plaintext
int i = 0; // initialize
while (i <= 10) {
    print i * i; // print a square
    i = i + 1;
}
```

Lexical analysis

```plaintext
A simple mini test program

int i = 0; // initialize
while (i <= 10) {
    print i*i;
    i = i + 1;
}

INT | ID(i) | = | INTLIT(0) | Semicolon ";" | WHILE
    | Open parenthesis "(" | ID(i) | <= | INTLIT(10)
    | Close parenthesis ")" | Open brace "{" | PRINT | ID(i)
    | * | ID(i) | Semicolon ";" | ID(i) | = | ID(i) | +
    | INTLIT(1) | Semicolon ";" | Close brace "}" |
```
Parsing

```
| INT | ID(i) | = | INTLIT(0) | Semicolon ";" | WHILE
| Open parenthesis "(" | ID(i) | <= | INTLIT(10)
| Close parenthesis ")" | Open brace "{" | PRINT | ID(i)
| * | ID(i) | Semicolon ";" | ID(i) | = | ID(i) | +
| INTLIT(1) | Semicolon ";" | Close brace "}" |
```
Static analysis

\[
\text{Stmts} \\
\text{InitVarIntro}(i) \quad \text{While} \quad \text{IntLit, 0} \quad \text{Lte, } \leq \quad \{\ldots\} \\
\text{Id, } i \quad \text{IntLit, 10} \quad \text{Print} \quad \text{ExprStmt} \\
\text{Mul, } * \quad \text{Assign} \\
\text{Id, } i \quad \text{Id, } i \quad \text{Id, } i \quad \text{Add, } + \\
\text{Id, } i \quad \text{IntLit, 1} \\
\{ (i, \text{int} ) \} \quad \checkmark
\]
Code generation

```
_stmts
  |---- InitVarIntro(i)
     |      |---- While
       |      |        |---- IntLit, 0
       |      |        |        |---- Lte, <=
       |      |        |        |        |---- {...}
       |      |        |        |        |        |---- Id, i
       |      |        |        |        |        |        |---- IntLit, 10
       |      |        |        |        |        |---- Print
       |      |        |        |        |        |        |---- ExprStmt
       |      |        |        |        |        |        |        |---- Mul, *
       |      |        |        |        |        |        |        |        |---- Assign
       |      |        |        |        |        |        |        |        |        |---- Id, i
       |      |        |        |        |        |        |        |        |        |---- Id, i
       |      |        |        |        |        |        |        |        |        |---- Id, i
       |      |        |        |        |        |        |        |        |        |---- Add, +
       |      |        |        |        |        |        |        |        |        |        |---- Id, i
       |      |        |        |        |        |        |        |        |        |---- IntLit, 1

.file  "squares.s"
.comm  _esp0, 4
.globl _Main_main
_Main_main:
pushl  %ebp
movl  %esp, %ebp
subl  $4, %esp
movl  $0, %eax
movl  %eax, -4(%ebp)
jmp   l1
l0:
movl  -4(%ebp), %eax
movl  -4(%ebp), %ebx
imull  %ebx, %eax
movl  %esp, _esp0
subl  $4, %esp
andl  $0xffffffff0, %esp
movl  %eax, (%esp)
call  _print
movl  _esp0, %esp
movl  $1, %eax
movl  -4(%ebp), %ebx
addl  %ebx, %eax
movl  %eax, -4(%ebp)
l1:
movl  $10, %eax
movl  -4(%ebp), %ebx
cmpl  %eax, %ebx
je    10
movl  %ebp, %esp
popl  %ebp
ret
```
.file "squares.s"
.comm _esp0, 4
.globl _Main_main
_Main_main:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl $0, %eax
movl %eax, -4(%ebp)
jmp l1
l0:
movl -4(%ebp), %eax
movl -4(%ebp), %ebx
imull %ebx, %eax
movl %esp, _esp0
subl $4, %esp
andl $0xffffffff0, %esp
movl %eax, (%esp)
call _print
movl _esp0, %esp
movl $1, %eax
movl -4(%ebp), %ebx
addl %ebx, %eax
movl %eax, -4(%ebp)
l1:
movl $10, %eax
movl -4(%ebp), %ebx
cmpl %eax, %ebx
jle 10
movl %ebp, %esp
popl %ebp
ret
Modularity in compiler design
Modularity

- Modularity is all about building large systems from collections of smaller components
- Modular implementations can be easier to write, test, debug, understand, and maintain than monolithic implementations
- For example:
  - Components can be developed independently
  - Some components can be reused in other contexts
  - Some components may even be useful as standalone tools
Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:
Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:

  `cpp`: the C preprocessor, expands the use of macros and compiler directives in the source program
Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:

  `cpp` -> `cc1` -> `prog.S`

`cc1`: the main C compiler, which translates C code to the assembly language for a particular machine
Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:

  `cpp` → `cc1` → `as`

  *`as`: the assembler, which translates assembly language programs into machine code*
Advantages of modularity

• Some components (e.g., `as`) are useful in their own right

• Some components can be reused (e.g., replace `cc1` to build a C++ compiler)

• Some components (e.g., `cpp`) are machine independent, so they do not need to be rewritten for each new machine

• Modular implementations can be easier to write, test, debug, understand, and maintain
Disadvantages of modularity?

• **Performance**
  
  It takes extra time to write out the data produced at the end of each stage
  
  It takes extra time to read it back in at the beginning of the next stage
  
  Later stages may need to repeat calculations from earlier stages if the information that they need is not included in the output of those earlier stages
  
• **But modern machines and disks are pretty fast, and compilers are often complex, so modularity usually wins!**
General building blocks

• A **front end** reads source programs (e.g., flat text files) and captures the corresponding abstract syntax in a collection of data structures (e.g., trees, graphs, arrays, …)

• A **middle end** analyzes and manipulates the abstract syntax data structures of a program

• A **back end** generates output (e.g., a flat, binary executable file) from the abstract syntax data structures of a program

• Substantial parts of these components can be shared by multiple tools
  • Example: the ghc (compiler) and ghci (interpreter) for Haskell use the same front and middle end components
  • Example: the g++ compiler for C++ and gcc compiler for C use the same middle and back end components
Multiple languages and targets

- Suppose that we want to write compilers for $n$ different languages, with $m$ different target platforms.

- That’s $n \times m$ different compilers!
An intermediate language

• Alternatively: design a general purpose, shared “intermediate language”:

- C
- C++
- Java
- Ada

• Now we only have \( n \) front ends and \( m \) back ends to write!

• The biggest challenge is to find an intermediate language that is general enough to accommodate a wide range of languages and machine types
Summary

• Basic principles
  programs as data

• Interpreters and compilers
  correctness means preserving semantics

• The compiler pipeline / “phase structure”
  source input, lexical analysis, parsing, static analysis, code generation, optimization

• Modularity
  Techniques for simplifying compiler construction tasks