Compilers

A compiler is a translator from “high-level” language to assembly code/object language.

Language $L \rightarrow $ TRANSLATOR $\rightarrow$ Language $L'$

Examples of translators:

- Pascal, C, etc. $\rightarrow$ Compiler $\rightarrow$ Machine Code
- Java $\rightarrow$ Compiler $\rightarrow$ Byte Code
- Ratfor $\rightarrow$ Preprocessor $\rightarrow$ Fortran
- Tex $\rightarrow$ Text Formatter $\rightarrow$ Postscript
- SQL $\rightarrow$ DB Optimizer $\rightarrow$ Query plan

We study common features of translators, by building one.

Language Design

We study languages from an implementor's viewpoint.

- How do compilation feasibility and runtime efficiency affect language design?

(There are more “theoretical” approaches to studying programming languages, and there are interesting languages that don’t compile easily...)

Course Goals

- Improve understanding of languages and machines.
- Learn practicalities of translation.
- Learn “anatomy” of programming languages.
- Apply computer science theory to practical problems (using tools).
- Do large programming project.

“Von Neumann” Machine

Key Characteristics

- Sequential control flow + labels + jumps
- Small set of built-in data types and operators (e.g., byte, integer, floating point)
- Flat linear address space.
- Memory hierarchy (registers faster than memory faster than disk).
“High-Level” Languages

E.g., Fortran, Pascal, C, Cobol, Java, ...

Example

\[
\text{PROCEDURE rev (a:real array, n:int) }
\text{LOCAL VAR i,j: int; x: real; }
\text{BEGIN}
\text{\quad i := 0; j := n - 1;}
\text{WHILE i < j DO}
\text{\quad x := a[i]; a[i] := a[j]; a[j] := x;}
\text{\quad i := i + 1; j := j - 1;}
\text{END}
\text{END}
\]

Features

- Expressions (arithmetic, logical)
- Control structures (loops, conditionals, etc.)
- Type declarations and type checking
- Composite types (arrays, records, etc.)
- Procedures/Functions, with private scope
- Abstraction facilities!

How can we make high-level language and Von Neumann machine meet?

Answer:

- Translate HLL into lower-level code (in traditional compiler, to machine code.)
and/or
- Build a “higher level” virtual machine (in traditional interpreter, perhaps a stack machine.)

In practice, we do some of both, even in a compiler, since generated machine code makes use of a runtime library and operating system.

Compiler Structure: Want Simplicity and Flexibility

Source Code

Lexical Analysis

Error Handling

Syntax Analysis

(Lexical Analysis & “Semantic Analysis”)

Intermediate Code Generation

“Parse Trees”

“I.R. (Language-dependent)"

“Front End”

“I.R. (Machine-dependent)"

Intermediate Code Generation

Code Optimization

“Back End”

Code Generation

Assembly/Object Code

Example

Source characters:

\[
\text{if (a} <= \text{b[i]) a} := 4.5 ;
\]

Lexical Analysis

Theory: regular languages, FAs
Tools: lex,JLex, etc.

Token stream:

\[
\text{IF ‘(‘ (ID a) LE (ID b) ‘[‘ (ID i) ‘]‘ ‘)}
\text{(ID a) ASSGN (FCONST 4.5) ‘);’}
\]

Syntax Analysis

“hierarchical”
Theory: context-free languages, PDA
Tools: yacc, javaCup, javaCC, etc.

Parse tree:

(real or conceptual)

\[
\text{IF-THEN}
\]

\[
\text{predicate}
\]

\[
\text{statement}
\]

\[
\text{ASSIGN}
\]

\[
\text{expr expr}
\]

\[
\text{var expr}
\]

\[
\text{ID a array-lookup ID a const}
\]

\[
\text{ID b ID i FCONST 4.5}
\]
Language Definition

Syntax is easy.
• Well-understood.
• Good theory: regular and context-free languages and automata.
• Good tools, even for complex cases.

Semantics are hard.
• Inherently complex.
• Variety of choices:
  - Operational — Definitional interpreter
    (↑ we will focus here)
  - Axiomatic — Logic
  - Denotational — Mathematical functions
    etc.
• Few tools.