CS321/322
Programming Languages and Compiler Design

Compiler Design & Implementation
| Language Design & Implementation

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.
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...)
“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

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

Tokens

Syntax Analysis
 (& "Semantic Analysis")

"Parse Trees"

Intermediate Code Generation

"Front End"
 (Language-dependent)

"Back End"
 (Machine-dependent)

Symbol Management

Error Handling

"Parse Trees"  
CS321

Interpreter

I. R.
 (Bytecode, Trees, etc.)

Code Optimization

I.R.

Code Generation

Assembly/Object Code
Example

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

Lexical Analysis
Theory: regular languages, FAs
"linear"
Tools: lex, JLex, etc.

Token stream: \texttt{IF ' (' (ID a) LE (ID b) ' [' (ID i) ' ] ) ' (ID a) ASSGN (FCONST 4.5) ' ; '}

Syntax Analysis
Theory: context-free languages, PDA
"hierarchical"
Tools: yacc, javaCup, javaCC, etc.

Parse tree:
(real or conceptual)

\begin{verbatim}
statement
  
  IF-THEN
  
  predicate
  <=
  expr expr
  
  &
  expr
  ID a
  
  &
  array-lookup
  ID b
  
  &
  expr
  ID i

  &
  statement
  
  ASSIGN
  var expr
  
  &
  expr
  ID a
  
  &
  const
  FCONST 4.5
\end{verbatim}
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.