CS321 Languages and Compiler Design I
Winter 2012
Lecture 1
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.
A **compiler** is a **translator** from “high-level” language to assembly code/object language.

Language L → **TRANSLATOR** → Language L’

Examples of translators:

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

We study common features of translators, by building one.
We study languages mainly 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 and useful languages that don’t compile easily...)
### “VON NEUMANN” MACHINE

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>Control Store</th>
<th>DATA STORE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>instr</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>instr</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>instr</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>instr</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>instr</td>
<td>m-1</td>
</tr>
<tr>
<td></td>
<td>instr</td>
<td>m</td>
</tr>
</tbody>
</table>

- Control Store:
  - instr
  - instr
  - instr

- Operator:
  - operator
  - args

- Data Store:
  - 0
  - 1
  - 2
  - 3
  - m-1
  - m
FEATURES OF LOW-LEVEL CODE

- 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, JavaScript, Python...

Example

```python
func rev (a: @real, n:int) {
    var i := 0;
    var j := n - 1;
    while i < j do {
        var x := a[i];
        a[i] := a[j];
        a[i] := a[j];
        a[j] := x;
        i := i + 1;
        j := j - 1
    }
}
```
FEATURES OF HIGH-LEVEL CODE

- 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 run on a Von Neumann machine?

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

Symbol Management

Error Handling

Lexical Analysis

Tokens

Syntax Analysis

(semantic analysis)

Intermediate Code Generation

"Front End"

(Language-dependent)

"Parse Trees"

(CS321)

(CS322)

Interpreter

I. R.

(Bytecode, Trees, etc.)

"Back End"

(Machine-dependent)

Code Optimization

I.R.

Code Generation

Assembly/Object Code

PSU CS321 W'12 LECTURE 1 © 1992–2012 ANDREW TOLMACH
Front-end Example

Source characters:  \( \text{if} (a \leq b[i]) \ a := 4.5 \)

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

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

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

Parse tree:
- (real or conceptual)
  - \( \text{statement} \)
    - \( \text{IF-THEN} \)
      - \( \text{predicate} \)
        - \( \leq \)
          - \( \text{expr} \)
            - ID a
          - \( \text{expr} \)
            - \( \text{array-lookup} \)
              - ID a
              - \( \text{var} \)
                - ID b
              - \( \text{expr} \)
                - ID i
      - \( \text{statement} \)
        - \( \text{ASSIGN} \)
          - \( \text{var} \)
            - ID a
          - \( \text{const} \)
            - FCONST 4.5
**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
    \( \uparrow \text{we will focus here}\)
  - Axiomatic — Logic
  - Denotational — Mathematical functions
    etc.
- Few tools.