Lecture 18

Compilers and Language Translation
(S&G, ch. 9)

Read S&G ch. 10
(Models of Computation)
Program Development

Editor
Create high-level language program

Compiler
Check for syntax errors and translate to machine code

Run-time system
Execute the machine language program on any machine

Not OK (error messages)

Compiler: A Programming Language Translator

Set sum to 0
Set count to 1
While count <= 10 do
  Set sum to sum + count
  Increment count
Output sum

Compile

.data
.one: .data 1
.eleven: .data 11
.sum: .data 0
.count: .data 0

.end

.begin
.clear sum
.load one
.store count
.while: .load count
.compare eleven
.jumpeq endwhile
.add sum
.store sum
.increment count
.jump while
.endifwhile: .out sum
.halt

3/17/04 22:24
Compiler: Requirements / Tasks

- Recognize programming constructs (conditionals, loops) and translate them to assembly language (compare, jump)
- Reject and report unrecognized programs
- Keep track of symbols (sum, count) and declare each one using a .data directive
- Generate efficient assembly language code

Recognizing Constructs: Parsing

- Could try to translate, e.g., from every possible if/else to its corresponding assembly code:

```plaintext
if x > y
  output x
else
  output y
```

Rule #18734

```plaintext
load x
compare y
jumpgt outx
out y
jump next
outx: out x
next: ....
```
Recognizing Constructs: *Parsing (2)*

- Could try to translate, *e.g.*, from every possible `if/else` to its corresponding assembly code:

```markdown
if a < b
  output a
else
  output b
```

**Rule #22102**

- Impossible, because we’d need an infinite number of rules!

Recognizing Constructs: *Parsing (3)*

- Instead, we recognize (abstract) structures, and work on their components one at a time:
Recognizing Constructs: *Parsing* (4)

- Now we can work on smaller pieces:

  \[ a < b \]

  **Parse**

  - `<`
  - `load a`
  - `compare b`

  **Generate**

- When parsing fails, compiler reports a syntax error:

  \[ a << b \]

  **Parse**

  - `Unrecognized operator: <<`

BNF Grammar

- A notation for expressing all the allowable statements in a language
- A finite description of an infinite number of possibilities
- Each distinct arrangement is shown in schematic form
- Because languages are nested hierarchically, the definitions are recursive
Example Grammar

1. \( \langle \text{assignment statement} \rangle ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle \)
2. \( \langle \text{expression} \rangle ::= \langle \text{variable} \rangle \)
   \( l \langle \text{expression} \rangle \langle \text{arith op} \rangle \langle \text{variable} \rangle \)
   \( l ( \langle \text{expression} \rangle ) \)
3. \( \langle \text{arith op} \rangle ::= + | - | * | / \)
4. \( \langle \text{variable} \rangle ::= x | y | z \)
   \( \Leftarrow \text{Unrealistic!} \)

Additional Example

1. \( \langle \text{if statement} \rangle ::= \)
   \( \text{if} ( \langle \text{Boolean expression} \rangle ) \langle \text{assignment statement} \rangle ; \langle \text{else clause} \rangle \)
2. \( \langle \text{Boolean expression} \rangle ::= \langle \text{variable} \rangle \)
   \( l \langle \text{expression} \rangle \langle \text{relation} \rangle \langle \text{expression} \rangle \)
3. \( \langle \text{relation} \rangle ::= == | < | > \)
4. \( \langle \text{else clause} \rangle ::= \text{else} \langle \text{assignment statement} \rangle \)
   \( l \Lambda \)
Keeping Track of Symbols: Symbol Tables

• When a symbol is first set, compiler should declare it:

\[
	ext{Set sum to 0} \quad \Rightarrow \quad \text{sum: .data 0}
\]

• Subsequent references to symbol don’t need a new declaration:

\[
\text{Set sum to sum + count} \quad \Rightarrow \quad \text{load sum, add count, store sum}
\]

Keeping Track of Symbols: Symbol Tables (2)

• Failure to set initial value should result in an error:

\[
\text{Set sum to 0} \quad \text{While count } \leq 10 \text{ do } \quad \Rightarrow \quad \text{“count” not declared}
\]

• Compiler uses a symbol table (dictionary; phonebook) to keep track of values we’ve declared.
Symbol Tables

- In a language like Java, symbol table would contain info about variable types:
  
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>x</td>
<td>double</td>
</tr>
</tbody>
</table>

- Then compiler can use symbol table to detect “type errors”:

  \[ a = a + s; \]

  \textit{can't set int to String}

Optimization: Generating Efficient Assembly Code

- “Laundry metaphor”: \textit{Unoptimized}
  1. Load dirty clothes into washer
  2. Run washer
  3. Unload wet clothes from washer to basket
  4. Load wet clothes into dryer from basket
  5. Run dryer

- Unoptimized version allows us to break down the task into simple parts
- Requires less coordination between washer & dryer
**Optimization: Generating Efficient Assembly Code (2)**

- "Laundry metaphor": *Optimized*
  1. Load dirty clothes into washer
  2. Run washer
  3. Load wet clothes from washer to dryer
  4. Run dryer

- Optimized version saves a step and doesn't require a basket
- Requires more coordination between washer & dryer

---

**Optimization**

*unoptimized*

Set a to a + b
Output a
Set a to a + c

*optimized*

load a
add b
store a
out a
load a
add c
store a

• Note coordination required between first and third steps of pseudocode