Lecture 18

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

Read S&G ch. 10
(Model 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

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

OK
Not OK (error messages)

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:

if x > y
output x
else
output y

load x
compare y
jump next
out y
jump next
next:....
Recognizing Constructs: Parsing (2)

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

```
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:

```
if/else
```

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. `(assignment statement) ::= (variable) = (expression)`
2. `(expression) ::= (variable)`
   ` l (expression) (arith op) (variable)`
   ` l ( expression )`
3. `(arith op) ::= + | - | * | /`
4. `(variable) ::= x | y | z`  \(\Leftarrow\) Unrealistic!

Additional Example

1. `(if statement) ::= if ( (Boolean expression) ) (assignment statement) ; (else clause)`
2. `(Boolean expression) ::= (variable)`
   ` l (expression) (relation) (expression)`
3. `(relation) ::= == | < | >`
4. `(else clause) ::= else (assignment statement)`
   ` l \lambda`
Keeping Track of Symbols: Symbol Tables

- When a symbol is first set, compiler should declare it:
  ```
  Compile
  Set sum to 0
  sum: .data 0
  ```

- Subsequent references to symbol don't need a new declaration:
  ```
  Compile
  Set sum to sum + count
  load sum
  add count
  store sum
  ```

Symbol Tables

- In a language like Java, symbol table would contain info about variable types:
  ```
  a: int
  b: int
  s: String
  x: double
  ```

- Then compiler can use symbol table to detect "type errors":
  ```
  a = a + s;  
  can't set int to String
  ```

Optimization: Generating Efficient Assembly Code

- “Laundry metaphor”: 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

- Note coordination required between first and third steps of pseudocode