Code Generation & Intro to Software Engineering

Week 4
CS 212 – Spring 2008

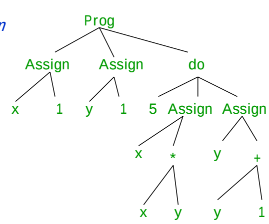
Announcements

- Part 2 Assignment
  - For both Compiler and GBA, assignment will appear online soon (probably Thursday morning)
  - Part 2 is due on or about March 1
    - Exact due date will be specified as it approaches

Recall

- We use *recursive descent parsing* to go from *program* to *AST* (Abstract Syntax Tree)

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```



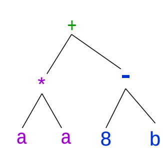
Recursion

- The grammar drives the design of the parser
  - We write a parsing method for each nonterminal
  - Within the method, each terminal token is checked; the nonterminals can take care of themselves (via recursive calls)
- The AST drives the design of the code generator
  - We write a code-generation method for each AST node-type
  - Within the method, we generate code for the node; the subtrees can take care of themselves (via recursive calls)

Code for Expressions

- Goal is to leave expression's value on top of the SaM stack
- For our example, there are 3 kinds of expression nodes:
  - Numbers (e.g., 42)
  - Variables (e.g., x)
    - We assume a is at mem 0, b at mem 1, c at mem 2, etc.
  - Operators (e.g., +)
- Desired code
  - Number
    - PUSHIMM 42
  - Variable
    - PUSHOFF 0
or
PUSHOFF 1
or
PUSHOFF 2
  - Operator
    - <code for left subtree>
<code for right subtree>
ADD

Example Expression Code




```
PUSHOFF 0
PUSHOFF 0
TIMES
PUSHIMM 8
PUSHOFF 1
SUB
ADD
```

Code For Assignment Statements

- Goal is to store the value of the <expression> into the <variable> (e.g., b)
  - We already have the code to place the expression's value on top of the stack
- Desired code

  <code for expression>
  STOREOFF 1

- Example: b = a + 5;

```
PUSHOFF 0
PUSHIMM 5
ADD
STOREOFF 1
```

Code For Do Statements

- This is harder because we have to maintain a counter
- Goal is to
  - Place the do-expression on top of stack to act as counter
  - If counter has reached zero we remove counter from stack and leave the loop
  - Generate code for all statements within the do-statement
  - Decrement the counter

```
<code for expression>
loop: DUP
NOT
JUMPC endloop
<code for statements>
PUSHIMM 1
SUB
JUMP loop
endloop: ADDSP -1
```

- Possible improvement: Code is wrong if <expression> is negative

Code for a Program

- Goal is to
  - Reserve space for the 26 variables (a-z)
  - Print the values of the variables at the end of the program
- Resulting code:

```
ADDSP 26
<code for statements>
WRITE
WRITE
…
WRITE
STOP
```

- Note that each type of AST node produces just a small amount of code
  - The do-statement was the most complicated
  - It produced 7 instructions of its own

Example Program and Resulting Code

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```

```
ADDSP 26
PUSHIMM 1
STOREOFF 23
PUSHIMM 1
STOREOFF 24
PUSHIMM 5
do0: DUP
NOT
JUMPC end0
PUSHOFF 23
PUSHOFF 24
TIMES
STOREOFF 23
PUSHOFF 24
PUSHIMM 1
ADD
STOREOFF 24
PUSHIMM 1
SUB
JUMP do0
end0: ADDSP -1
WRITE
…
WRITE
STOP
```

EBNF

- BNF = Backus-Naur Form
  - A way of representing a grammar for a programming language
  - Originally Backus *Normal* Form
    - Switched at suggestion of Knuth (partly because not really a *normal* form)
    - Naur was editor of Algol-60 document which used BNF
- EBNF = Extended BNF
  - Basically, BNF with some extra simplifying notation
  - There is an official standard, but it's common to modify it
- Typical constructs
  - Way to distinguish between terminals and nonterminals
  - * for repetition
  - [ ] for optional
  - (| |) for choice

Example Grammar Notation: Java

Statement:
  Block
  if *ParExpression Statement [*else *Statement]*
  for ($ForInit_{Opt}$; *[Expression]* ; $ForUpdate_{Opt}$) *Statement*
  while *ParExpression Statement*
  do *Statement* while *ParExpression* ;
  try *Block (Catches* | *[Catches]* finally *Block)*
  switch *ParExpression* { *SwitchBlockStatementGroups* }
  synchronized *ParExpression Block*
  return *[Expression]* ;
  throw *Expression* ;
  break *[Identifier]*
  continue *[Identifier]*
  ;
  ExpressionStatement
  Identifier : *Statement*

Example Grammar Notation: Python

```
if_stmt ::=
        "if" expression ":" suite
          ( "elif" expression ":" suite )*
          ["else" ":" suite]

while_stmt ::=
        "while" expression ":" suite
          ["else" ":" suite]

for_stmt ::=
        "for" target_list "in" expression_list
         ":" suite
          ["else" ":" suite]
```

Software Engineering

- Engineering
  ABET: “the profession in which a knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.”
- Software Engineering
  IEEE: “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.”

Engineering vs. Software Engineering

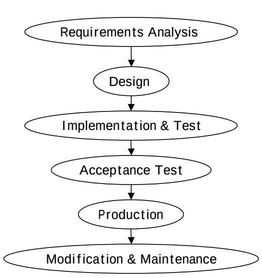
- Engineering
  - Ability to build from prefab components
  - Uses metrics (i.e., measurements, as in physical units)
  - Tolerances are important
- Software Engineering
  - Re-use is encouraged, but is not always practiced; designs are often “from scratch”
  - No clear physical units (although there is a concept of *software metrics*)
  - No real equivalent

Programming in the Large

- How do we design and implement a large program consisting of many modules?

Models for Software Development

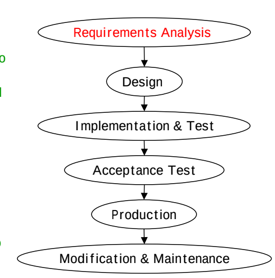
Waterfall model:



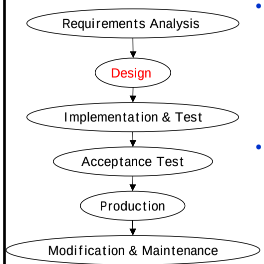
- This model is idealized
  - True development is never entirely sequential
  - There is feedback from each stage of the process
- There are many other models for software development
  - XP, RUP, CMM, SCRUM, FDD

Requirements Analysis

- Requirements analysis consists of
  - Functional requirements
    - What is the program supposed to do?
    - How should the program respond to errors?
  - Performance requirements
    - How fast?
    - How much storage?
  - Determine delivery schedule
    - When does the “customer” need it?
    - How much time can we devote to it?
  - Additional requirements
    - Example: A game should be “fun”



Software Design



- Design goals
  - Meet functional and performance requirements
  - The components are all good abstractions
  - The structure is relatively easy to implement and maintain
- Design is usually done iteratively
  - Select a target abstraction to implement
  - Identify useful helper abstractions (i.e., decompose the problem)
  - Specify behavior for the helpers
  - Sketch implementation plan for the target
  - Iterate

Top-Down vs. Bottom-Up Design

- Top-Down Design
  - Start with what is wanted
  - Determine what is needed to achieve it
- Bottom-Up Design
  - Start with what is implementable
  - Determine how these can be put together to achieve goal
- Top-Down design is usually more effective for all but small programs
- A rule to keep in mind:

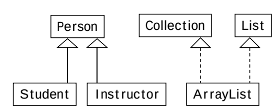
  Avoid implementing an abstraction until its design is complete

Data Models

- As part of the design, it helps to create a *data model*
  - A diagram showing relations between important entities
  - The entities are mostly classes, but they don't have to be
- A data model defines
  - The kinds of data being manipulated
  - How they relate to one another
- One way to describe a data model is to use a graph (e.g., UML)

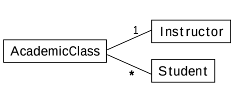
UML

- UML (Unified Modeling Language) is one technique for diagramming data models
  - Each "class" is shown in a box with its (important) fields and methods
- In UML:
  - An open-headed arrow shows inheritance
  - A dashed open-headed arrow shows "implements an interface"

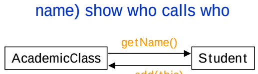


More UML

- Composition
  - Edges without arrow-heads are used to show containment
  - The edge is labeled to show how many
    - 0..1 (0 to 1)
    - 1 (exactly one)
    - * (zero or more)



- Arrows with a closed head (and labeled with a method name) show who calls who

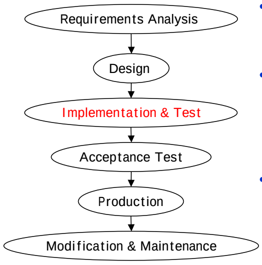


- Goal is to have a convenient picture showing relations between objects
  - The examples here showed just a few parts of UML
    - There are whole books on UML
  - There are several other data modeling schemes

Evaluating a Design

- A team conducts a Design Review
  - Design Review: evaluating functionality
    - Explain how design captures the data model
    - Do a walk-through on symbolic test-data
    - Do this for entire design, and for individual modules or groups of modules
  - Design Review: evaluating program structure
    - Each abstraction should be coherent
      - A specification with lots of &&'s or lots of ||'s might indicate a single procedure that is trying to handle several abstractions
    - Abstraction interfaces should be no wider than necessary

Implementation & Testing



- Implementation:
  - Often the *least expensive* task
- Code Review:
  - Systematic examination of code intended to find and fix bugs at an early stage of development
- Testing:
  - Unit testing vs. integration testing
  - Glass-box testing vs. black-box testing

Another Testing Tool: Profiling

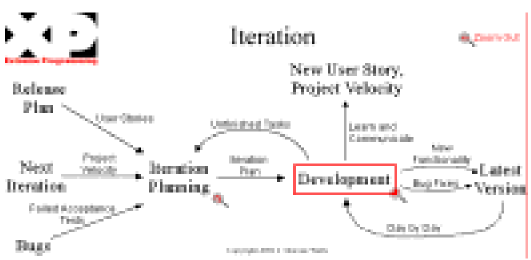
- People are notoriously bad at predicting the most computationally expensive parts of a program
  - Rule of thumb (Pareto Principle): 80% of the time is spent in 20% of the code
  - No use improving the code that isn't executed often
  - How do you determine where your program is spending its time?
- Part of the data produced by a *profiler* (Python)

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   2521    0.227    0.000    1.734    0.001 Drawing.py:102(update)
   7333    0.355    0.000    0.983    0.000 Drawing.py:244(transform)
   4347    0.324    0.000    4.176    0.001 Drawing.py:64(draw)
   3649    0.212    0.000    1.570    0.000 Geometry.py:106(angles)
     56    0.001    0.000    0.001    0.000 Geometry.py:16(__init__)
 343160    9.818    0.000   12.759    0.000 Geometry.py:162(_determinant)
   8579    0.816    0.000   13.928    0.002 Geometry.py:171(cross)
   4279    0.132    0.000    0.447    0.000 Geometry.py:184(transpose)
```

- Java has a built-in profiler (hprof); there are many others

Another Model for Software Development

- This is a diagram from a website promoting *extreme programming* (http://www.extremeprogramming.org/)



Some Features of *Extreme Programming*

- All code is written in response to a *user story* (4x6 card describing requirements)
- Start with smallest set of useful features; release early and often
- Simple design
  - Use simplest possible design that gets the job done
- Continuous testing
  - Tests are written *before* programming
  - When the tests are passed, the job is done
- Continuous integration
  - New code is added daily, but *all* tests must be passed
- *Pair programming*
  - Two programmers at one machine

Pair Programming

- Two programmers share one computer
  - One is the *driver*
    - Controls keyboard and mouse
    - Does all the writing of code
  - The other is the *observer*
    - Watches and guides
    - Focuses on strategic issues (e.g., how this module fits with others)
    - Is usually the *better or more experienced* programmer
- Claim: pair programming is *more productive* than having two separate programmers
- I've never tried it, but you might want to try this with your group

The Tower of Babel (1563). Pieter Bruegel the Elder