CSCI 410
20 –Compiler 2

Compilation Overview

Source Code ➔ Lexical Analysis ➔ Syntactic Analysis ➔ Semantic Analysis ➔ Intermediate Code (optional) ➔ Symbol Table ➔ More Optimization (optional) ➔ Target Code Generation ➔ Target Program

Focus of Chapter 10

Source Code ➔ Lexical Analysis ➔ Syntactic Analysis ➔ Semantic Analysis ➔ Intermediate Code (optional) ➔ Symbol Table ➔ More Optimization (optional) ➔ Target Code Generation ➔ Optimization (optional) ➔ Target Program

Focus of Chapter 11

Source Code ➔ Lexical Analysis ➔ Syntactic Analysis ➔ Semantic Analysis ➔ Intermediate Code (optional) ➔ Symbol Table ➔ More Optimization (optional) ➔ Target Code Generation ➔ Target Program

Syntax Analysis

Typically, generate an Abstract Syntax Tree (AST) based on grammar, source:

Source

```
while (count <= 100) {
  count++;
}
```

Grammar

```
stmt: whileStmt | ifStmt | expr | { stmtSeq }
whileStmt: while ( expr ) stmt
stmtSeq: ; | stmt ; stmtSeq
expr: ...
```

Credit

Most of the slides in this lecture come from www.nand2tetris.org
Code Generation

• Typically, work with abstract syntax tree (AST)
  • All later stages involve simple tree traversals
  • Tree is “decorated” with additional info (e.g., type info)
  • Type checking, simple optimizations directly on tree
  • Operator precedence easily resolved
• However, for our Jack compiler:
  • No type checking required
  • No operator precedence
  • No optimization code

Jack Compiler

Challenge: translate Jack source → VM code
• Simplicity of Jack/VM helps here: we need code for
  • Data manipulation
  • Operations
  • Lots of hard stuff already mostly done – in VM
    • Function call protocol
    • Loops/conditionals

Data Manipulation Example

```
method int foo() {
  var int x;
  let x = x + 1;
  ...
```

Syntax analysis

```
<letStatement>
  <keyword> let </keyword>
  <identifier> x </identifier>
  <symbol> = </symbol>
  <expression>
    <term>
      <identifier> x </identifier>
    </term>
    <symbol> + </symbol>
    <term>
      <constant> 1 </constant>
    </term>
  </expression>
</letStatement>
```

We’ll go straight from source to VM code without intermediate XML.

Handling Variables

When the compiler encounters a variable, say x, in the source code, it has to know:

**What is x’s data type?**

Primitive, or class type?
(need to know in order to properly allocate RAM resources for its representation)

**What kind of variable is x?**

local, static, field, argument?
(we need to know in order to properly allocate it to the right memory segment; this also implies the variable’s life cycle)

Memory Segments Review

VM memory Commands:

```
pop segment i
push segment i
```

Where i is a non-negative integer and segment is one of the following:

- **static:** holds values of global variables, shared by all functions in the same class
- **argument:** holds values of the argument variables of the current function
- **local:** holds values of the local variables of the current function
- **this:** holds values of the private ("object") variables of the current object
- **that:** holds array values (e.g., name, arr)
- **constant:** holds all the constants in the range 0…32767 (pseudo memory segment)
- **pointer:** used to anchor this and that to various areas in the heap
- **temp:** fixed-size entry segment that holds temporary variables for general use; shared by all VM functions in the program.

Variable Mapping Example

```
state BankAccount {
  // State properties:
  static int nAccounts;
  static double bankCommission;
  // Invariant properties:
  Field ownId;
  Field bankAccount;
  Field balance;
  method void transfer(int max, BankAccount from, BankAccount to) {
    var int i; // Some local variables
    var int r1, r2;
    let balance = (balance + max - commission) * 2;
    // More code ...
  }
}
```

When compiling this class, we have to create the following mappings:

<table>
<thead>
<tr>
<th>The class variables</th>
<th>nAccounts, bankCommission</th>
<th>are mapped on</th>
<th>static 0,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>The object fields</td>
<td>id, owner, balance</td>
<td>are mapped on</td>
<td>this 0,1,2</td>
</tr>
<tr>
<td>The argument variables</td>
<td>var, bankAccount, when</td>
<td>are mapped on</td>
<td>argument 0,1,2</td>
</tr>
<tr>
<td>The local variables</td>
<td>i, j, k, x</td>
<td>are mapped on</td>
<td>local 0,1,2</td>
</tr>
</tbody>
</table>
Symbol Tables

| Class | Field Type | Kind | | | |
|-------|------------|------|---|---|
| Account | balance | int | | | |
|          | amount | double | | | | |
Symbol void transfer(int from, int to, double amount) { 
    // BankAccount instance methods
    Account fromAccount = Account[from];
    Account toAccount = Account[to];
    fromAccount.balance -= amount;
    toAccount.balance += amount;
    return nothing?
}

Method scope symbol table

| Name    | Type     | Kind | | | |
|---------|----------|------|---|---|
| this | Account | argument | | | |
| pub | int | param | | | | |
| BankAccount | amount | int | | | | |
Symbol void transfer(int from, int to, double amount) { 
    // BankAccount instance methods
    Account fromAccount = Account[from];
    Account toAccount = Account[to];
    fromAccount.balance -= amount;
    toAccount.balance += amount;
    return nothing?
}

Managing Memory

Note: Java code

class Complex { 
    // Properties (fields): 
    int re; // Real part
    int im; // Imaginary part
    /** Constructs a new complex object */
    public Complex(int re, int im) { 
        this.re = re;
        this.im = im; 
    }
    ... 
    class Foo { 
        public void bla() { 
            Complex x; 
            ... 
        } 
    }
    
    Final state
    
    How to compile:
    x = new Complex(x, y);
    The compiler generates code that: 
    x = Memory.alloc(x);
    Where x is the number of words necessary to represent the object in question.

Accessing Object Fields

Note: Java code

class Complex { 
    // Properties (Fields): 
    int re; // Real part
    int im; // Imaginary part
    /** Constructs a new Complex (int re, int im) */
    public Complex(int re, int im) { 
        this.re = re;
        this.im = im; 
    }
    /** Multiplies this Complex number by the given scalar */
    public Complex(int i) { 
        this.re = this.re * i;
        this.im = this.im * i; 
    }
    ... 
}

How to compile:
getName() = (this + 1) * times (argument 4)
This pseudo-code should be expressed in the target language.

Managing Arrays

Note: Java code

class Bar { 
    // Properties (Fields): 
    Bar[] bar; // array
    /** Constructs a new Bar[] */
    public Bar(int k) { 
        bar = new Bar[k];
        // Construct the array:
        bar[0] = new Bar();
        bar[1] = new Bar();
        ... 
    }
    ... 
    class Foo { 
        public void bar(int i) { 
            bar[i] = new Bar();
            // Call the bar method
            ... 
        } 
    } 
    
    Final state
    
    How to compile:
    bar = new Mem.alloc();
    Generate code effecting:
    bar = Memory.alloc();
Java code:
```java
class Bla {
    ...
    void foo(int k) {
        int x, y;
        int[] bar; // declare an array
        // Construct the array:
        bar = new int[10];
        bar[0] = x;
    }
    ...
}
```

How to compile: `bar[k] = 19`?

VM Code:
```java
// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add // use the that segment to access x[k] push constant 19 push constant 1
```

RAM state, just after executing `bar[k] = 19`:
```
0 1 2 3 4 5 6 7 8 9
x (local 0) y (local 1) bar (pointer 2) k (argument 0)
```

Array Indexing