**CSCI 410**

21 – Compiler 3

---

**Code Generation Notes**

- Use same recursive descent structure as in proj. 10
- Start with simple programs and work up.
- Compilation process:
  - One .jack file at a time → one .vm file
  - Possible due to private fields and static vars
  - Possible due to constructor mechanism
  - Possible due to other Jack simplifications
  - Do entire directory
  - New symbol table for each class (statics, fields)
  - Extend with additional symbols (or separate symbol table) for each subroutine call (locals, args)

---

**class**

Grammar:

```
class className { classVarDec* subroutineDec* }
```

Code generation:
None

Tasks:
- Store class name (for use in function names)
- Make empty global symbol table
- Recursively call compile functions for class body

---

**classVarDec**

Grammar:

```
(static|field) type varName (, varName)* ;
```

Code generation:
None

Tasks:
- Fill in global symbol table for static and field vars

---

**subroutineDec (function)**

Grammar:

```
function (void|type) subroutineName ( parameterList ) subroutineBody
```

Code generation:
None (you don’t have enough info yet)

Tasks:
- Store subroutine name
- Make empty local symbol table
- Process parameterList
- Process subroutineBody

---

**parameterList**

Grammar:

```
((type varName) (, type varName)*)?
```

Code generation:
None

Tasks:
- Add args to local symbol table
subroutineBody

Grammar:
{ varDec* statements }

Code generation:
function className.functionName numLocals

Tasks:
Process varDecs – this will let us determine numLocals
Process statements

varDec

Grammar:
var type varName (, varName)* ;

Code generation:
None

Tasks:
Add local vars to local symbol table (and count them)

statements

Grammar:
statement*
where
statement: letStatement |
ifStatement |
whileStatement |
doStatement |
returnStatement

Tasks:
Process each statement
If no return statement, generate default return (0):
push constant 0
return

Empty Program Example

Jack code:
class Main {
  function void main() {
    return;
  }
}

VM code:
function Main.main @
push constant 0
return

doStatement

Grammar:
do subroutineCall ;

Code generation:
(process subroutineCall)
pop temp 0

Tasks:
Process subroutineCall
Pop return value from subroutineCall

subroutineCall

Grammar:
subroutineName ( expressionList ) |
{ className } varName . subroutineName ( expressionList )

There are 3 possibilities here, each with a different handling on the next 3 slides:
1. subroutineName ( expressionList )
2. className . subroutineName ( expressionList )
3. varName . subroutineName ( expressionList )
subroutineCall (1)

Grammar:
subroutineName ( expressionList )

Code generation:
push pointer 0         // push this object
(process expression list)
call currentClass.subroutineName numberOfExpressions

Tasks:
Construct VM function name as (current class name).subroutineName
Push this object reference on stack (as first, implicit argument)
Process expression list – pushes rest of arguments on stack
Perform call with # of args (implicit + explicit)

subroutineCall (2)

Grammar:
className . subroutineName ( expressionList )

Code generation:
(process expression list)
call className . subroutineName numberOfExpressions

Tasks:
Construct VM function name given className . subroutineName
No implicit push – there is no “this” object for this case
Process expression list – pushes arguments on stack
Perform call with # of args

subroutineCall (3)

Grammar:
varName . subroutineName ( expressionList )

Code generation:
push segment offset     // push object named varName
(process expression list)
call classOfVar.subroutineName numberOfExpressions

Tasks:
Lookup class name, segment, and offset for varName in symbol table
Construct VM function name as (class name).subroutineName
Push varName object reference on stack as first, implicit argument
Process expression list – pushes rest of arguments on stack
Perform call with # of args (implicit + explicit)

expressionList

Grammar:
(expression , expression)*)?

Code generation:
None

Tasks:
Process each expression (each results in one push)
Count # expressions for use above

expression

Grammar:
term ( op term )*

Code generation:
(Process term)
While next token is op:
(process next term) op

Tasks:
Process term
While next token is op:
consume op token
process next term
output code for operator

term

Grammar:
term: integerConstant
| stringConstant
| keyword
| varName
| varName { expression }
| subroutineCall
| ( expression )
| unaryOp term

Tasks:
Big if-else block
term: integerConstant

Code generation:
push constant integerConstant

term: ( expression )

Code generation:
None

Tasks:
Process expression

Program “Seven” Example

Jack code:
class Main {
    function void main() {
        do Output.printInt(1 + (2 * 3));
        return;
    }
}

VM Code:
function Main.main 0
    push constant 1
    push constant 2
    push constant 3
    call Math.multiply 2
    add
    call Output.printInt 1
    pop temp 0
    push constant 0
    return

letStatement (Non-Array Vars)

Grammar:
let varName = expression;

Code generation:
(Process expression)
pop segment offset

Tasks:
Process expression (leaves value on stack)
Lookup segment & offset for varName in symbol table
Emit pop code

letStatement Example

Jack code:
class Foo {
    static int z;
    function void bar(int x, int y) {
        let z = x + y;
    }
}

VM code:
function Foo.bar 1
    push argument 0
    push argument 1
    add
    pop static 0

whileStatement

Grammar:
while { expression } { statements }

Code generation:
label WHILE_<counter>
(expression code)
not
if/goto WHILE_END_<counter>
(statements code)
goto WHILE_<counter>
label WHILE_END_<counter>

Tasks:
Above code, then counter++
ifStatement

Grammar:
if ( expression ) { statements } ( else { statements } )?

Code generation:
(expression code)
not
if-goto IF_FALSE_<counter>
(statements code)
goto IF_END_<counter>
label IF_FALSE_<counter>
(else statements code)
label IF_END_<counter>

String Constants

Jack code:
"Foo"

VM code:
push constant 3 // length of string constant
call String.new 1
call String.appendChar 2
push constant 102 // ascii for 'f'
call String.appendChar 2
call String.appendChar 2

Methods: Calling

• When calling, must push object (this) as first arg
• Must call with # of arguments including this

var Foo x;
let x = Foo.new();
do x.baz(1, 2);

Methods: Declaring

• Mostly like functions, but:
  • Argument 0 is this – need to set in pointer segment
  • Need to offset other arguments in symbol table properly

class Foo {
  field int x, y;
  method void bar() {
    let x = x + 7;
    ...
  }
}

function Foo.bar 0
push argument 0
pop pointer 0 // this
...

Working with Fields

• Remember fields are private
• Assume this has properly been initialized

class Foo {
  field int x, y;
  ...
  method void bar() {
    let x = x + 7;
    ...
  }
}

Constructors

• Need to allocate memory when constructing
• Allocated memory assigned to this
• Programmer should return this at end

class Foo {
  field int x, y;
  ...
  constructor void new() {
    let x = 0;
    let y = 1;
    return this;
  }
}

function Foo.new 0
push constant 2
call Memory.alloc 1
push constant 0
pop pointer 0 // set this
push constant 0
pop this 0
push constant 1
pop this 1
push pointer 0
return
Working with Arrays (as term)

• Use that to temporarily access heap memory
• Dynamically build that value based on var, expr

```
var Array a;
var int j;
let a = Array.new(5);
let j = 0;
let x = a[j+1];
```

Working with Arrays (let assignment)

• Use that to temporarily access heap memory
• Dynamically build that value based on var, expr

```
var Array a;
var int j;
let a = Array.new(5);
let j = 2;
let a[j+1] = 4;
```