Compiler II: Code Generation

Friday, April 8, 2022 7:42 PM

"When I am working on a problem, I never think about beauty. But when I have finished, if the solution is not beautiful, I know it is wrong."

- R. Buckminster Fuller (1895-1993)

11.1 Code Generation

The Jack compiler is a two-tier compiler. It used the compiler to translate .jack files into .vm files, then it uses the VMTranslator to translate .vm files into .asm files.

Compiler = Compiler's front end

VMTranslator = Compiler's Back end

11.1.1 Handling Variables

In Jack there are no mapping complications as all primitives are represented using a 16 bit value.

Static variables must be visible to all subroutines within the execution of the program

field variables must be visible within the subroutine they were defined

Object Instance variables must be freed to clear the space they took up prior to instantiation.

Luckily this has been implemented into our VMTranslator

Symbol Table

static -> static
field -> this

local -> local
argument -> argument

The compiler will search through the symbol table to read the properties of the code written to see what is represents.

Separating the namespace of each scope is important as you could have two variables named the same in two different scopes.

Thus we will associate the scope of the variable within the symbol chart

Nested scopes can be implemented using a linked list

scope1 -> scope2 -> scope3 -> …

Handling Variable Declaration

When the class declaration is run the compiler will generate a class level symbol table and then

name	*type*	*kind*	*#*
x	*int*	*field*	*0*
y	*int*	*field*	*1*
pointCount	*int*	*static*	*0*

<name, type, kind, #>

Handling Variables in statements

When a variable is found in a statement the compiler will look up the variable in the subroutine's symbol table. If it is not found it checks the class's symbol table.

11.1.2 Compiling Expressions

infix = x + y
postfix = x y +

Jack language uses infix but our VMTranslator is written using postfix:
push x
push y

add

11.1.3 Compiling Strings

Character arrays

When printing

Output.printString("Hello World")

The string will persist in memory until the program terminates.

Java, C#, python will use their internal garbage collectors to free the memory allocated to variables that don't hold a value.

Operating System Services

The compiler can user any function, method, or class within the OS API (Appendix 6) as usable VM code

11.1.4 Compiling statements

Compiling return statements

call compileExpression to generate the VM code to put the return expression onto the stack. We add the expression no matter what as it makes compiling do statements easier.

Then we simply call return.

Compiling Let statements

let varName = expression

compileExpression to generate the VM code to add the expression the stack

pop varName

varName = local 3, static 0, field 2, etc.

Which will be pulled from the symbol table.

Compiling do statements

do className.functionName(exp1, exp2, exp3,…)

Ignoring the return means ignoring what is on the top of the stack thus popping it at the end.

Compiling while and if statements

We need to bridge the cap of if, while, for, switch (jack only uses if and while) to conditional go-to and unconditional go-to's.

This can be done using labels

Labels will use a running counter to make them globally unique

11.1.5 Handling Objects

Objects are stored on the heap

objects are stored as a pointer to the memory address that is allocated for them. We must allocate and deallocate this memory later by freeing it. Much like C.

11.1.5.1 Compiling Constructors

Compiling Constructor Calls

2 step process

declare variable
    var ClassName varName
Instantiation
    let varName = ClassName.new()

Some high level languages let you write this in one step… but it is always later broken down into two

let p = Point.new(2, 3);

Have the Point.new constructor allocate a two-word memory block for representing a new Point instance, initialize the two words of this block to 2 and 3, and have p refer to the base address of this block.

Compiling Constructors

A constructor is first and foremost a subroutine!

It is the same as a subroutine except the compiler must also:

i. *Create a new object*
ii. *Make the new object the current object (AKA this)*

Creating a new object means, allocating memory for said object in RAM.

ptr = Memory.Alloc(size);

Constructors in Jack must end with " return this; "

While in languages like Java, this is implied. It's the same in C when we want to create a new Object.

This bunch of low-end tasks within the VM translator and assembler that must be done to create an easy dev environment for the high-level developers.

11.1.5.2 Compiling Methods

Compiling Method Calls

Suppose we have two points p1 and p2

In POP these points represent composite data types (think of a struct)

Thus we would use some function
distance(p1, p2);

In OOP these points represent instances of a class.

Thus we would use a class method call
p1.distance(p2);

POP - procedural
find distance() of p1, p2
OOP - data driven
use p1's distance method to find distance to p2

varName.methodCall(exp1, exp2, exp3);
methodCall(exp1, exp2, exp3);

push varName (symbol in table map otherwise push this)
call compileExpressionList
call className.methodName n + 1

Compiling Methods

```
int distance( Point other ) {
    int dx, dy;
    dx = x  - other.x;
    dy = y - other.y;
    return Math.sqrt( (dx*dx) + (dy*dy) );
}
```

operates on the current object.

```
dx = x - other.x; -> dx = this.x - other.x;
```

fields of objects can only be manipulated using getters and setters.

Since we pass n + 1 arguments through ever method call… We will always at least pass one argument. Argument 0 on the stack being the reference to the object we are currently operating on.

Thus argument 0 will always hold the pointer to the object we are to define as "this" within jack.

function Point.distance 2 (one being ptr and one being the other object)
push argument 0
pop pointer 0

This will push ptr onto stack and then pop it to pointer 0 to be used as this.

11.1.6 Compiling Arrays

Arrays are the same as objects. The key difference is the allowance of indexing via [i]

The compiler is what makes this unique. There is no way to write jack code to change the syntax of the Jack language. Thus we must Implement this into out

compiler.

Using pointer notation we can see

*arr[1] = *(arr + 1)*

We compute

Let x = arr[i]
push arr
push i
add (putting arr + i onto the stack)
pop pointer 1 (storing the address in the THAT pointer)
push that 0
pop x

Let arr[expression 1] = expression2;
push arr
call expression to compile the expression and add it to the stack
add (arr + expression)
call expression to compile the expression and add it to the stack
pop temp 0
pop pointer1
pop temp 0
pop that 0

11.2 Specification

prompt> JackCompiler source

source = filename (Xxx.jack) or a directory containing one or more jack files.

11.3 Implementation

Jack files (Xxx.jack) -> VM files (Xxx.vm)

Jack subroutines (Yyy) in a jack file (Xxx) -> VM subroutine Xxx.Yyy

Variables start indexing at 0 except for methods as you must pass the ptr of the object operated on, so method arguments start indexing at 1.

Mapping Object Fields

push argument 0
pop pointer 0

Mapping Array Elements

See Compiling arrays

Mapping Constants

null and false = push constant 0

true = push constant 1

constant this

push pointer 0