10. The Compiler II: Code Generation[1]

"This document describes the usage and input syntax of the Unix Vax-11 assembler "As." As is designed for assembling code produced by the "C" compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended."

Berkeley Vax/Unix Assembler Reference Manual (1983)

In this chapter we complete the development of the Jack compiler. The overall compiler is based on two modules: the VM backend developed in chapters 6 and 7, and the Syntax Analyzer and Code Generator developed in chapters 9 and 10, respectively. Although the second module seems to consist of two separate sub-modules, they are usually combined into one program, as we will do in this chapter. Specifically, in chapter 9 we built a Syntax Analyzer that "understands" -- parses -- source Jack programs. In this chapter we extend this program into a full-scale compiler that converts each "understood" Jack operation and construct into equivalent series of VM operations.

9.1 Background

Symbol Table

A typical high-level program contains many identifiers. Whenever the compiler encounters any such identifier, it needs to know what it stands for: is it a variable name, a class name, or a function name? If it's a variable, is it a field of an object, or an argument of a function? What type of variable is it -- an integer, a string, or some other type? The compiler must resolve these questions in order to map the construct that the identifier represents onto a construct in the target language. For example, consider a C function that declares a local variable named `sum` as a `double` type. When we will translate this program into the machine language of some 32-bit computer, the `sum` variable will have to be mapped on a pair of two consecutive addresses, say `RAM[3012]` and `RAM[3013]`. Thus, whenever the compiler will encounter high-level statements involving this identifier, e.g. `sum+=i` or `printf(sum)`, it will have to generate machine language instructions that operate on `RAM[3012]` and `RAM[3013]` instead.

We see that in order to generate target code correctly, the compiler must keep track of all the identifiers introduced by the source code. For each identifier, we must record what the identifier stands for in the source language, and on which construct it is mapped in the target language. This information is usually recorded in a "housekeeping" data stricture called *symbol table*. Whenever a new identifier is encountered in the source code for the first time (e.g. in variable declarations), the compiler adds its description to the table. Whenever an identifier is encountered elsewhere in the program, the compiler consults the symbol table to get all the information needed for generating the equivalent code in the target language. An example of this technique is given in Prog. 10-1. At this stage we ask the reader to ignore the code generation part of the example (bottom of the figure).

[1] From The Elements *of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

```
// Some common sense was sacrificed in this banking application in order
// to create non-trivial and easy-to-follow compilation examples.
class BankAccount {
   // class variables
   static int nAccounts;
   static int bankCommission;  // as a precentage, e.g. 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;
   ...  // the constructor is discussed later in the chapter
   method int commission(int x) {
      // if id<=1000 the acct owner is a bank employee so commission is 0
      if (id>1000) {return (x*bankCommission)/100;} else {return 0;}
   }
   ...
   method void transfer(int sum, bankAccount from, Date when) {
      var int i,j,k;  // some local variables
      var Date d1;
      ...
      /* add the transferred sum to the balance minus the bank commission,
         which is inflated to make the example more interesting */
      let balance=(balance+sum)-commission(sum*5);
      ...
   }
   ...
}
```

Class-scope symbol table

Name	type	Kind	#
naccounts	int	Static	0
bankCommission	int	Static	1
id	int	Field	0
Owner	String	Field	1
balance	int	Field	2

Method-scope (`transfer`) sym. table

name	Type	kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
k	int	var	2
d1	Date	var	3

```
// VM pseudo code
push balance
push sum
add
push this
push sum
push 5
call multiply
call commission
sub
pop balance
```

```
// Hack VM code
push this 2
push argument 1
add
push argument 0
push argument 1
push constant 5
call Math.multiply 2
call BankAccount.commission 1
sub
pop this 2
```

PROGRAM 10-1: Symbol table and code generation example. We assume that the source language is Jack (top) and the target language is the stack-based VM developed in Ch. 6-7.

The basic symbol table solution is complicated slightly due to the fact that most languages allow different parts of the program to use the same identifiers for different purposes. For example, two C functions may declare a local variable named `x` for two completely different purposes. The programmer is allowed to re-use such symbols freely in different program units, since the

compiler is clever enough to map them on completely different objects in the target language, as implied by the program's context (and consistent with the programmer's intention). Therefore, whenever the compiler encounters an identifier `x` in a program, it must know which `x` we are talking about, and generate the appropriate code accordingly.

The common solution is to associate each identifier with its own *scope*, i.e. the region of the program in which the identifier is recognized. Thus in addition to all the relevant information that must be recorded about each identifier, the symbol table must also reflect in some way the identifier's scope. The classic data structure for this purpose is a list of *hash tables*, each reflecting a single scope. When the compiler fails to find the identifier in the table of the current scope, it looks it up in the next table, from inner scopes outward. Thus if `x` appears undeclared in a certain code segment (e.g. a method), it may be that `x` is declared in the code segment that owns the current segment (e.g. a class). The need for a list of more than two tables arises in languages that feature arbitrarily deep scoping. For example, in Java one can define local variables whose scope is restricted to the "`{`" `block` "`}`" in which they are defined.

To sum up, depending on the scoping rules of the compiled language, the symbol table is typically implemented as a list of two or more hash tables.

Memory Allocation

One of the basic challenges faced by every compiler is how to map the various types of variables of the source program onto the memory of the target platform. This is not a trivial task. First, different variable *types* require different amounts of memory, so the mapping is not one-to-one. Second, different *kinds* of variables have different life cycles. For example, a single copy of each static variable should be kept "alive" as long as the entire class is being compiled. In contrast, each object instance of the class should have a different copy of all its private variables (called *fields* in Jack). Also, each time a function is being called, a new copy of its local variables must be created -- a need which is clearly seen in recursion. In short, memory allocation to variables is a dynamic and intricate task.

That's the bad news. The good news are that low-level memory allocation can be all but "outsourced" to a virtual machine implementation, operating as the compiler's backend. For example, the VM that we created in Chapters 6-7 has built-in mechanisms for representing static, local, argument, as well as object-, and array-type variables. Further, the VM knows how to dynamically create new copies of variables, when necessary, while keeping on the stack copies of variables of subroutines that did not yet terminate. Recall that this functionality was not achieved easily. In fact, we had to work rather hard to create a VM implementation that maps all these constructs and behaviors on a flat RAM structure and a primitive instruction set, respectively. Yet this effort was worth our while: for any given language *L*, any *L*-to-VM compiler is now completely relieved from low-level memory management; all it has to do is map source constructs on respective VM constructs -- a rather simple translation task. Further, any improvement in the way the VM implementation manages memory will immediately affect any compiler that depends on it. That's why it pays to develop efficient VM implementations and continue to improve them down the road.

Expression Evaluation

How should we generate code for evaluating high level expressions like `x+f(2,y,-z)*5`? First, we must "understand" the syntactic structure of the expression, e.g. convert it into a parse tree like the one depicted in Fig 10-2. Next, we can simply traverse the tree and generate the target code. Clearly, the choice of the code generation algorithm will depend on the target language into which we are translating.

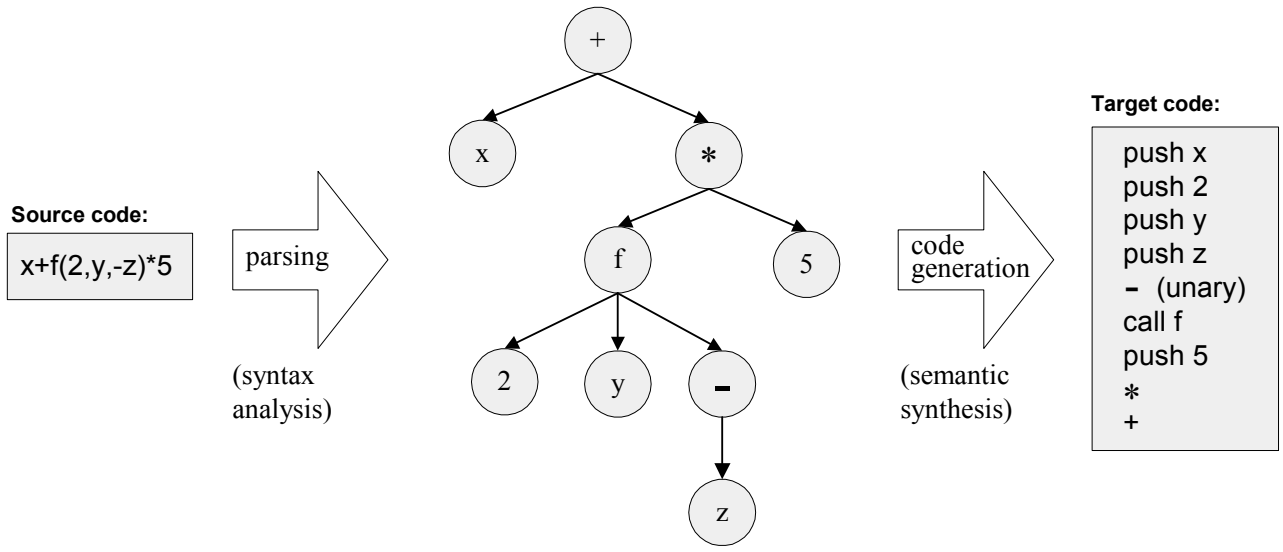

Figure 10-2: Code generation for expressions is based on a syntactic understanding of the expression, and can be easily accomplished by recursive manipulation of the expression tree. Note that the parsing stage was carried out in Chapter 9.

The strategy for translating expressions into a stack-based language is based on a postfix (depth-first) traversal of the corresponding expression tree. One such strategy is used in Algorithm 10-3.

Code(exp):

if exp is a number *n*	then	output "`push n`"
if exp is a variable *v*	then	output "`push v`"
if exp = (exp1 op exp2)	then	Code(exp1); Code(exp2) ; output "`op`"
if exp = op(exp1)	then	Code(exp1) ; output "`op`"
if exp = f(exp1 … expN)	then	Code(exp1) … Code(expN); output "`call f`"

Algorithm 10-3: A recursive postfix traversal algorithm for evaluating an expression tree by generating commands in a stack-based language.

The reader can verify that when applied to the tree in Fig 10-2, Algorithm 10-3 will yield the desired stack-machine code.

Flow Control

Structured programming languages are equipped with a variety of high-level control structures like `if`, `while`, `for`, `switch`, and so on. In contrast, low-level languages typically offer two control primitives: conditional and unconditional goto. Therefore, one of the challenges faced by the compiler is to translate structured code segments into target code that includes these primitives only. Figure 10-4 gives two examples.

Source code	*Generated code*

`if (cond)`
`  s1`
`else`
`  s2`
`…`
	`  code for computing ~cond`
	`  if-goto L1`
	`  code for executing s1`
	`  goto L2`
	`label L1`
	`  code for executing s2`
	`label L2`
	`  …`

`while (cond)`
`  s1`
`…`
	`label L1`
	`  code for computing ~cond`
	`  if-goto L2`
	`  code for executing s1`
	`  goto L1`
	`label L2`
	`  …`

Figure 10-4: Compilation of control structures

Two features of high-level languages make the compilation of control structures slightly more challenging. First, control structures can be nested, e.g. `if` within `while` within another `while` and so on. Second, the nesting can be arbitrarily deep. The compiler deals with the first challenge by generating unique labels, as needed, e.g. by using a running index embedded in the label. The second challenge is met by using a recursive compilation strategy. The best way to understand how these tricks work is to discover them yourself, as you will do when you will build the compiler implementation described below.

Arrays

An array is usually implemented as a sequence of consecutive memory locations. The array name is usually treated as a pointer to the beginning of the array's allocated memory block. In some languages (e.g. Pascal), the entire memory space is allocated when the array is declared. In other languages (e.g. Java), the array declaration results in the allocation of a single pointer only. The array proper is created in memory later, when the array is explicitly constructed during the program's execution. This type of *dynamic memory allocation* is done from the heap, using the memory management services of the operating system. An obvious advantage of dynamic memory allocation is that the size of the array does not have to be pre-determined, resulting in

better memory utilization. Fig. 10-5 offers a snapshot of the memory organization of a typical array.

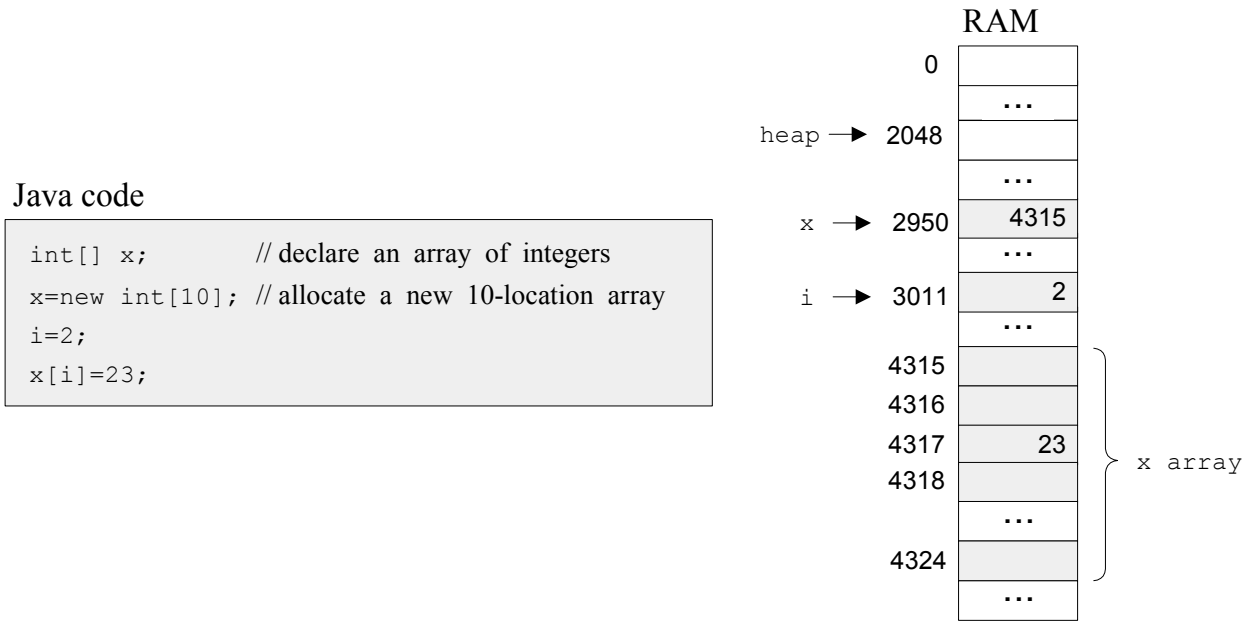

Figure 10-5: Array creation and manipulation. All the addresses in the example were chosen arbitrarily (except that in the Hack platform, the heap indeed begins at address 2048). Note that the basic operation illustrated is `*(x+i)=23`.

Thus storing the value 23 in the *i*'th location of array x can be done by the following pointer arithmetic:

```
push x
push i
+
pop addr    // at this point addr points to x[i]
push 23
pop *addr   // store the topmost stack element in RAM[addr]
```

Explanation: The fact that the first four commands make variable *addr* point to the desired array location should be evident from Figure 10-5. In order to complete the storage operation, the target language must be equipped with some sort of an indirect addressing mechanism. Specifically, instead of storing a value in some memory location *y*, we need to be able to store the value in the memory location whose address is the current contents of *y*. In the example above, this operation is carried out by the "`pop *addr`" pseudo-command. Different virtual machines feature different ways to accomplish this indirect addressing task. For example, our VM handles indirect addressing using the `pointer` and `that` segments (see Fig. 6-13).

Objects

Object-oriented languages allow the programmer to encapsulate data and the code that operates on the data within programming units called *objects*. This level of abstraction does not exist in low-level languages. Thus, when we translate code that handles objects into a primitive target language, we must "flatten" the object structure and handle its underlying data and code explicitly. This will be illustrated in the context of Prog. 10-6.

```
class BankAccount {
   ...
   field int id;
   field String owner;
   field int balance;
   ...
   constructor BankAccount new(int a, String b, int c) {
      let id=a;  let owner=b, let balance=c;
   }
   ...
   method void main() {
      var BankAccount joeAcct,janeAcct;  // some accounts
      var Date d;
      // to save clutter we skip the construction of d
      ...
      let joeAcct=BankAccount.new(1,"joe",5000);  // create Joe's acct
      let janeAcct=BankAccount.new(2,"jane",2000); // create Jane's acct
      ...
      do janeAcct.transfer(3000,joeAcct,d); // move $ from Joe to Jane
      ...
   }
   ...
}
```

Program 10-6: Some additional code segments of the `BankAccount` class presented in Prog. 10-1.

Object data (construction): The data kept by each object instance is essentially a list of fields. As with array variables, when an object-type variable is declared, the compiler only allocates a reference (pointer) variable. The object proper is allocated memory later, when the object is created via a call to the class constructor. To explain this better, we note that the operating system that you will build in the next chapter has a method called `alloc(int size)`. This method is designed to find an available memory block of size `size` on the heap, and return a pointer to the block's base address. When compiling a constructor like `BankAccount new(int a, String b, int c)`, the compiler generates code that (i) requests the operating system to find a memory block to store the new object, and (ii) sets the `this` pointer to the base of the allocated block. From this point onward, the object's fields can be accessed linearly, using an index relative to its base. Thus statements like `let id=a` and `let owner=b` can be easily compiled, as we now turn to explain.

Object data (usage): The previous paragraph focused on how the compiler generates code that creates new objects. We now describe how the compiler handles commands that manipulate the data encapsulated in existing objects. For example, consider the handling of a statement like `let owner=b`. First, an inspection of the symbol table will tell the compiler that (i) `this` is a variable of type `BankAccount`, and (ii) `owner` is the 2nd field of the `BankAccounts` class. Using this information, the compiler can generate code effecting the operation `*(this+2)=b`. Of course the generated code will have to accomplish this operation using the target language.

Object code: The encapsulation of methods within object instances is a convenient abstraction which is not implemented for real. Unlike the fields data, of which different copies are indeed kept for each object instance, only one copy of each method is actually kept at the target code level. Thus the trick that stages the code encapsulation abstraction is to have the compiler force

the method to always operate on the desired object. We have chosen to handle this convention as follows: at the VM level, each method assumes that a reference (pointer) to the object on which it has to operate is available in its 0^{th} argument. With that in mind, when compiling a method call like `do a.b(x)`, we generate the code `push a`, `push x`, `call b`. Thus, when the compiler translates the method call `do janeAcct.transfer(3000,joeAcct,d)`, it generates the code `push janeAcct`, `push 3000`, `push joeAcct`, `push d`, `call transfer`. As a result, the `transfer` method will end up operating on the `joeAcct` object, since this is its 0^{th} argument.

Finally, let us go back to Prog. 10-1 and consider the method call `commission(sum*5)` which is part of the bold statement in the figure. When the `commission` method is invoked, how does it "know" that it has to operate on the `this` object? The answer lies at the code generated at the bottom of Fig. 10-1, where we see that the compiler preceded the method call with a "`push this`" command. Thus when the `commission` code starts running, "`id`" ends up being `this.id`, and so on. More precisely, when the `commission` code has been compiled, the compiler has used the symbol table to learn that (i) `this` is of type `BankAccount`, and (ii) `id` is the 0^{th} field of this class. Using these information the compiler has generated code in which references to `id` were replaced with references to the `*(this+0)` memory location.

9.2 Specification

Usage: The Jack compiler accepts a single command line argument that specifies either a file name or a directory name:

```
prompt> JackCompiler source
```

If *source* is a file name of the form `xxx.jack`, the compiler compiles it into a file named `xxx.vm`, created in the same folder in which the input `xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `xxx.jack` file in the directory, a corresponding `xxx.vm` file is created in the same directory.

Standard mapping over the Virtual Machine

This section lists a set of conventions that must be followed by every Jack-to-VM compiler.

File and function naming: Each Jack class is compiled into a separate `.vm` file. The Jack subroutines (functions, methods, and constructors) are compiled into VM functions as follows:

- A Jack subroutine `x()` in a Jack class `Y` is compiled into a VM function called `Y.x`.
- A Jack *function* or *constructor* with *k* arguments is compiled into a VM function with *k* arguments.
- A Jack *method* with *k* arguments is compiled into a VM function with *k*+1 arguments. The first argument (argument number 0) always refers to the `this` object.

Returning from void methods and functions:

- VM functions corresponding to void Jack methods and functions must return the constant 0 as their return value.
- By definition, a "`do subName`" statement always invokes a void function or method. Therefore, when translating such statements, the caller of the corresponding VM function must remember to pop the returned value (which is always the constant 0).

Memory allocation and access:

- The static variables of a Jack class are allocated to, and accessed via, the VM's `static` segment of the corresponding `.vm` file.
- The local variables of a Jack subroutine are allocated to, and accessed via, the VM's `local` segment.
- Before calling a VM function, the caller must push the function's arguments onto the stack. If the VM function corresponds to a Jack method, the first pushed argument must be the object on which the method is supposed to operate.
- Within a VM function, arguments are accessed via the VM's `argument` segment.
- Within a VM function corresponding to a method or a constructor, access to the fields of this object is obtained by first pointing the VM's `this` segment to this object and then accessing individual fields via "`THIS` *index*" references.
- Within a VM function, access to array entries is obtained by pointing the VM's `that` segment to the address of the desired array location.

Operating system functions:

When needed, the compiler should use the following built-in functions, provided by the operating system:

- Multiplication and division is handled using the OS functions `Math.multiply()` and `Math.divide()`.
- String constants are handled using the OS constructor `String.new(length)` and the OS method `String.appendChar(nextChar)`.
- Constructors allocate space for constructed objects using the OS function `Memory.alloc(size)`.

9.3 Implementation

We now turn to describe a software architecture for the compiler. The proposed architecture builds upon the Syntax Analyzer described in chapter 9. In fact, the current architecture is based on gradually evolving the Syntax Analyzer into a full-scale compiler. The overall compiler can thus be constructed using five modules:

- A main driver that organizes and invokes everything (class `JackCompiler`);
- A tokenizer (class `JackTokenizer`);
- A symbol table (class `SymbolTable`);
- An output module for generating VM commands (class `VMWriter`);
- A recursive top-down compilation engine (class `CompilationEngine`).

Class JackCompiler

The program receives a name of a directory, and compiles all the Jack files in this directory. For each `xxx.jack` file, it creates a `xxx.vm` file in the same directory. The logic is as follows:

For each `xxx.jack` file in the directory:

1. Create a tokenizer from the `xxx.jack` file
2. Create a VM-output stream into the `xxx.vm` file
3. Compile(`INPUT:` tokenizer, `OUTPUT:` VM-output stream)

Class JackTokenizer

The API of the tokenizer is given in chapter 9.

Class SymbolTable

This class provides services for creating, populating, and using a *symbol table*. Recall that each symbol has a scope from which it is visible in the source code. In the symbol table, each symbol is given a running number (index) within the scope, where the index starts at 0 and is reset when starting a new scope. The following kinds of identifiers appear in the symbol table:

Static:	Scope: class.
Field:	Scope: class.
Argument:	Scope: subroutine (method/function/constructor).
Var:	Scope: subroutine (method/function/constructor).

When compiling code, any identifier not found in the symbol table may be assumed to be a subroutine name or a class name. Since the Jack language syntax rules suffice for distinguishing between these two possibilities, and since no "linking" needs to be done by the compiler, these identifiers do not have to be kept in the symbol table. Here is the class API:

- **`SymbolTable():`** create a new empty symbol table.
- **`void startSubroutine(String name):`** start the scope of a new method or function or constructor (ends the scope of the previous subroutine).
- **`String getSubroutineName():`** returns the name of the current method, function, or constructor.
- **`Void define(String name, String type, int kind):`** defines a new identifier of a given *name*, *type*, and *kind* and assigns it a number. *kind* must be one of the class constants `KIND_STATIC`, `KIND_FIELD`, `KIND_ARG`, `KIND_VAR`. *type* should be either a primitive Jack type (`boolean`, `int`, `char`) or a class name.
- **`int varCount(int kind):`** returns the number of variables of the given *kind* that are already defined in the current scope.
- **`int kindOf(String name):`** returns the *kind* of the named identifier. The *kind* should be one of the class constants `KIND_STATIC`, `KIND_FIELD`, `KIND_arg`, `KIND_VAR`, or the class constant `KIND_NONE`, if the identifier is unknown in the current scope.
- **`String typeOf(String name):`** returns the *type* of the named identifier.
- **`int indexOf(String name):`** returns the number assigned to the named identifier.

Comment: you will probably need to use two separate hash tables to implement the symbol table: one for the class-scope and another one for the subroutine-scope. When a new subroutine is started, the subroutine-scope table should be cleared.

class VMWriter

This class writes VM commands into a file. The API is as follows:

- **`VMWriter(Writer w):`** A constructor that accepts a *Writer* object (output stream) into which VM commands will be written.
- **`close():`** finishes the writing and closes the file.
- **Code writing methods:** The remainder of this class is a series of code writing methods, one for each command in the target VM language. Each method is designed to write the necessary VM command to the output.

class CompilationEngine

This class does the compilation itself. It reads its input from a **`JackTokenizer`** and writes its output into a **`VMWriter`**. It is organized as a series of **compilexxx()** methods, where `xxx` is a syntactic element of the Jack language. The contract between these methods is that each **`compilexxx()`** method should read the syntactic construct `xxx` from the input, **`advance()`** the tokenizer exactly beyond `xxx`, and emit to the output VM code effecting the semantics of `xxx`. Thus **`compilexxx()`** may only be called if indeed `xxx` is the next syntactic element of the input. If `xxx` is a part of an expression and thus has a value, then the emitted code should compute this value and leave it at the top of the VM stack.

The API of this module is identical to the API of the Syntax Analyzer's compilation engine, specified in chapter 9. Changing this compilation engine into one that emits VM code instead of XML code should be rather straightforward. You may use the hints in the section 9.5.

9.4 Perspective

The simplicity of the Jack language permitted us to side-step several compilation issues. In particular we did not handle virtual methods which are required in object-oriented languages that support inheritance. This section is work in progress.

9.5 Build it

We suggest to start by first building the symbol table and using it to extend the Syntax Analyzer created in project 9. In particular, whenever an identifier is encountered in the program, output the following information as well:

- Identifier category (var, argument, static, field, class, subroutine).
- Whether the identifier is presently being defined (e.g. the identifier stands for a variable declared in a "`var`" statement) or rather being used (e.g. the identifier stands for a variable in an expression).
- If the identifier represents a variable of one of the four kinds, the running number assigned to the identifier by the symbol table.

At this point you can make the switch to a real compiler, and start generating real VM code. This can be done incrementally.