11. The Compiler II: Code Generation[1]

"The syntactic component of a grammar must specify, for each sentence, a deep structure that determines its semantic interpretation…"

Noam Chomsky (b. 1928), U.S. mathematical linguist

(This chapter is work in progress). 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 7 and 8, and the Syntax Analyzer and Code Generator developed in chapters 10 and 11, 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 10 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 on equivalent VM constructs.

1. Background

A program is composed of operations that manipulate data. When we compile a program into a lower level language we must first consider how the data items are mapped into the lower level language and then how each possible operation is translated into a sequence of low-level operations.

1.1. Mapping of data items

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. If we 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 structure called *symbol table*.

[1] From The Elements *of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2004, www.idc.ac.il/csd

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.

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). Specifically, in most languages each identifier has a well defined *scope*, i.e. the region of the program in which the identifier is recognized. Whenever the compiler encounters an identifier `x` in a program, it treats `x` as the one currently in scope, and generate the appropriate code accordingly. The complication in handling different scopes comes from the fact that they can usually be nested within each other. The convention in most languages is that inner-scoped definitions always hides more outer-scoped ones.

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 nested within the next one in the list. 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).

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

Allocation of Variables

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" for the complete duration of the program. In contrast, each object instance of the class should have a different copy of all its instance variables (*fields*). 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 of variables is an intricate task.

That's the bad news. The good news is that we have already handled these difficulties. The VM that we created in Chapters 8-9 has built-in mechanisms for representing and handling the standard kinds of variables needed by high-level languages: static, local, arguments, and fields of objects. All the allocation and manipulation details were already handled at the VM level. 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 – at this point 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.

Arrays

Compilers usually implement arrays as sequences 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. Figure 1 offers a snapshot of the memory organization of a typical array.

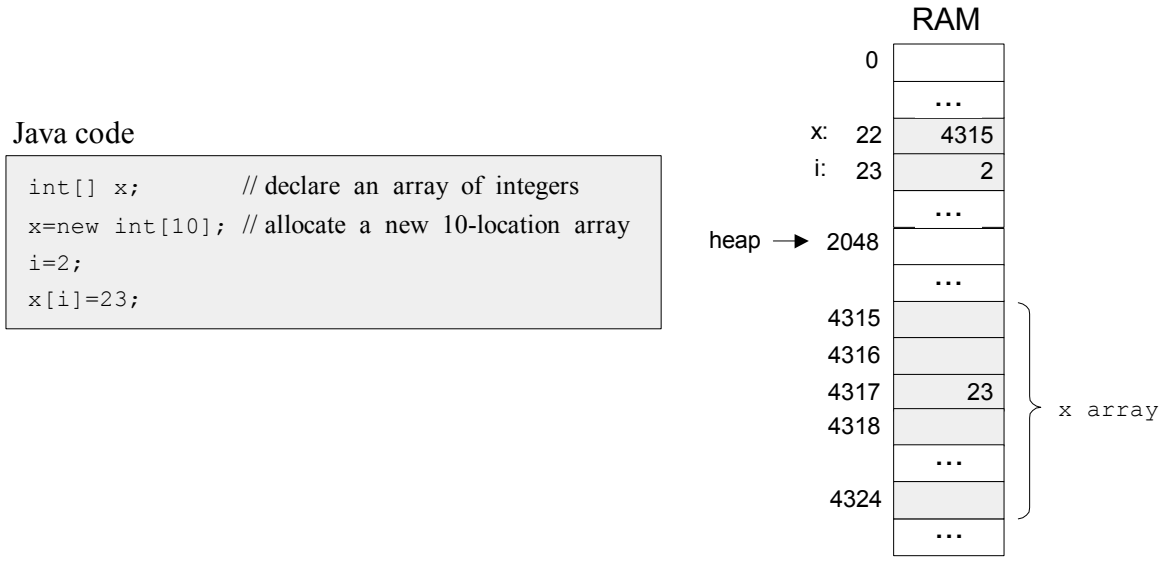

FIGURE 1: Array creation and manipulation. All the addresses in the example were chosen arbitrarily (except that in the Hack platform, the heap indeed begins at address 2048). Note that the basic operation illustrated is `*(x+i)=23`.

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 pseudo-commands make variable *addr* point to the desired array location should be evident from Figure 1. 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, the VM built in chapters 7-8 handles indirect addressing using its `pointer` and `that` segments.

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 handle its underlying data and code explicitly. This will be illustrated in the context of Program 2.

```
/** A Bank Account */
class BankAccount {

  static int sysID;

  // Fields:
  int id;
  String owner;
  int balance;

  private int nextID() {
    return ++sysID;
  }

  /** construct a new bank account with 0 balance */
  public BankAccount(String name) {
    id = nextId();
    owner = name;
    balance = 0;
  }

  /** deposit money in this bank account*/
  public void deposit(int amount) {
     balance = balance + amount;
  }

  // more methods come here

  public static void main(String args[]) {
    BankAccount joeAcct;
    sysID = 0;
    ...
    joeAcct = new BankAccount(”joe”);
    joeAcct.deposit(5000);
    ...
  }
  ...
} // BankAccount
```

PROGRAM 2

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 typically only allocates a reference (pointer) variable. The memory space for the object proper is allocated only when the object is created via a call to the class constructor. The space for the new object must ultimately be allocated by the operating system that must provide some service like “alloc(size)”

that finds a free memory block of the required size and returns a pointer to its base. When compiling a constructor like `BankAccount(String name)`, the compiler generates code that (i) requests the operating system to find a memory block to store the new object, and (ii) sets a pointer to the base of the allocated block to be called, within the constructor, "this". From this point onward, the object's fields can be accessed linearly, using an index relative to its base. Thus statements like `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 balance=balance+amount` within the method deposit. First, an inspection of the symbol table will tell the compiler that `amount` is an argument, while `balance` is the 2nd field of the `BankAccounts` class (starting the field count from 0). Using this information, the compiler can generate code effecting the operation `*(this+2) = *(this+2) + (argument 1)`. 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 that 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. A standard simple way to handle this is by passing the object reference as a "hidden" argument. I.e. a method call like `xxx.m(y)` is actually compiled as if it were written as `m(xxx,y)`: "`push xxx`, `push y`, `call m`". A syntactic detail that has to be handled is making sure that the called method `m` is really the one defined for xxx's class. In object-oriented languages this determination must be done in run-time due to the possibility of method overriding in a sub-class. When run-time typing is out of the picture, e.g. in languages like Jack, or if `m` was somehow declared not to be virtual, then all that is needed is to ensure that the called method `m` belongs to the correct class. E.g. in our example, if `xxx` was a variable of class `Xxx`, then we may call the method named `Xxx.m`.

1.2. Command translation

We now turn to describe how commands are translated. There are two elements to consider: expression evaluation and flow control.

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 Figure 3. This was already handled in chapter 10. Next, we 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.

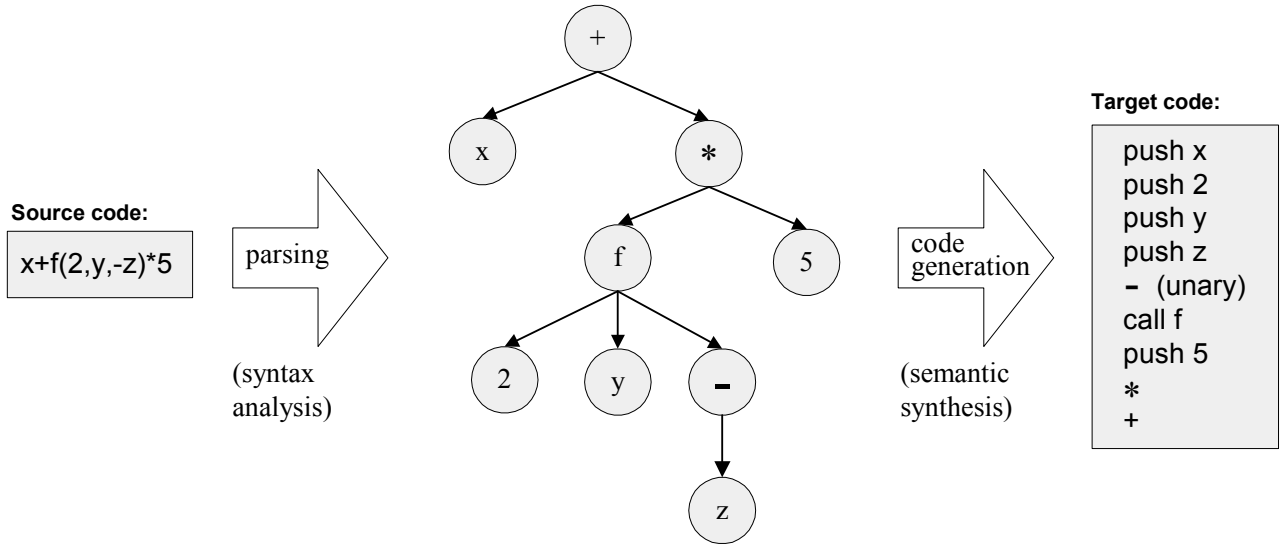

FIGURE 3: Code generation for expressions is based on a syntactic understanding of the expression, and can be easily accomplished by recursive manipulation of the expression tree. Note that the parsing stage was carried out in Chapter 10.

The strategy for translating expressions into a stack-based language is based on a postfix (depth-first) traversal of the corresponding expression tree. This simple strategy is described in Algorithm 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 4: 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 Figure 3, Algorithm 4 yields 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 5 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 5: 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.

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

- A Jack subroutine `xxx()` in a Jack class `Yyy` is compiled into a VM function called `Yyy.xxx`.
- 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.
- When translating a "`do subName`" statement that invokes a *void* function or method, the caller of the corresponding VM function must remember to pop (and ignore) 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 VM functions corresponding to Jack *methods* or *constructors*, access to the fields of the *this* object is obtained by first pointing the VM's `this` segment to the current object (using "`pointer 0`") and then accessing individual fields via "`this` *index*" references. For VM functions corresponding to Jack *methods*, the base of the `this` segment is passed as the 0'th argument and code for setting it is automatically inserted by the compiler at the beginning of the VM function. For constructors, the base of the `this` segment is obtained and set when the space for the object is allocated. The code for this allocation is automatically inserted by the compiler at the beginning of the constructor's code.
- 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.

Constants:

- `null` and `false` are mapped to the constant 0. `True` is mapped to the VM constant –1 (that is obtained via "`push constant 0`" followed by "`neg`".)

Use of 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)`.

3. Implementation

3.1. Compilation Example

We start with a simple example. This examples shows (parts of) a Jack class, the constructed symbol tables, and the generated VM code.

```
// High-level (Jack) code
// Some common sense was sacrificed in this banking example in order
// to create non-trivial and easy-to-follow compilation examples.
class BankAccount {
   // class variables
   static int nAccounts;
   static int bankCommission;  // as a percentage, e.g. 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) {/* code omitted */ }

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;  // some local variables
      var Date due;   // some other date variable
      // … omitted code
      let balance = (balance + sum) - commission(sum * 5);
      return;
   }
   // ... more methods
}
```

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
due	Date	var	2

```
// VM pseudo code

function BankAccount.commission
// … code omitted
function BankAccount.trasnfer
push this-that-was-passed-as-argument
pop this-segment-base
// .. code omitted
push balance
push sum
add
push this
push sum
push 5
call multiply
call commission
sub
pop balance
push 0
return
// … code omitted
```

```
// VM code

function BankAccount.commission 0
 // … code omitted
 function BankAccount.trasnfer 3
 push argument 0
 pop pointer 0
 // … code omitted
 push this 2
 push argument 1
 add
 push argument 0
 push argument 1
 push constant 5
 call Math.multiply 2
 call BankAccount.commission 2
 sub
 pop this 2
 push constant 0
 return
 // … code omitted
```

PROGRAM 6: Symbol table and code generation example.

Compilation examples for arrays and objects can be found as examples in chapter 8.

3.2. Suggested Design

We now turn to propose a software architecture for the compiler. This architecture builds upon the Syntax Analyzer described in chapter 10. 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 (`JackCompiler`);
- A tokenizer (`JackTokenizer`);
- A symbol table (`SymbolTable`);
- An output module for generating VM commands (`VMWriter`);
- A recursive top-down compilation engine (`CompilationEngine`).

Class JackCompiler

The program receives a name of a file or a directory, and compiles the file, or 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-writer into the `Xxx.vm` file
3. Compile(`INPUT:` tokenizer, `OUTPUT:` VM-writer)

Class JackTokenizer

The API of the tokenizer is given in chapter 10.

Class SymbolTable

This module 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 may 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.

A symbol table that associates names with information needed for Jack compilation: type, kind, and running index. The symbol table has 2 nested scopes (class/subroutine).			
Routine	**Arguments (type)**	**Returns**	**Function**
`Constructor`	--	--	Creates a new empty symbol table
`startSubroutine`	`--`	--	Starts a new subroutine scope (i.e. erases all names in the previous subroutine's scope.)
`define`	`name` (String) `type` (string) `kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	--	Defines a new identifier of a given *name*, *type*, and *kind* and assigns it a running index. `STATIC` and `FIELD` identifiers have a class scope, while `ARG` and `VAR` identifiers have a subroutine scope.
`varCount`	`kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	int	Returns the number of variables of the given *kind* already defined in the current scope.
`kindOf`	`name` (String)	(`STATIC`, `FIELD`, `ARG`, `VAR`, `NONE`)	Returns the *kind* of the named identifier in the current scope. Returns `NONE` if the identifier is unknown in the current scope.
`typeOf`	`name` (String)	String	Returns the *type* of the named identifier in the current scope.
`indexOf`	`name` (String)	int	Returns the *index* assigned to 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.

VMWriter

This class writes VM commands into a file. It encapsulates the VM command syntax.

Emits VM commands into a file			
Routine	**Arguments (type)**	**Returns**	**Function**
`Constructor`	Output file / stream	--	Creates a new file and prepares it for writing VM commands
`writePush`	`Segment` (`CONST`, `ARG`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`) `Index` (int)	--	Writes a VM `push` command
`writePop`	`Segment` (`CONST`, `ARG`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`) `Index` (int)	--	Writes a VM `pop` command
`WriteArithmetic`	`command` (`ADD`, `SUB`, `NEG`, `EQ`, `GT`, `LT`, `AND`, `OR`, `NOT`)	--	Writes a VM arithmetic command
`WriteLabel`	`label` (String)	--	Writes a VM `label` command
`WriteGoto`	`label` (String)	--	Writes a VM `label` command
`WriteIf`	`label` (String)		Writes a VM `If-goto` command
`writeCall`	`name` (String) `nArgs` (int)	--	Writes a VM `call` command
`writeFunction`	`name` (String) `nLocals` (int)	--	Writes a VM `function` command
`writeReturn`	--	--	Writes a VM `return` command
`close`	--	--	Closes the output file

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 10. We suggest gradually morphing the syntax analyzer into a full compiler.

4. Perspective

The fact that Jack is a relatively simple language permitted us to side-step several compilation issues. Here we mention some of the most significant ones.

While Jack looks like a typed language, this is hardly the case: all data types are 16-bit long, and the semantics of the language allow compilers to ignore almost all type information (with the single exception that a method call `x.m()` must know `x`'s type). In most other languages the type system has significant implications for the compiler: different amounts of memory must be allocated for different types; conversion from one type into another requires specific operations; the compilation of a simple expression like `x+y` strongly depends on the types of `x` and `y`; and so on. In particular Jack compilers need not determine the types of expressions, e.g. array entries in Jack are not typed.

Another significant simplification is that Jack does not support inheritance. This major simplification implies that all method calls can be determined statically at compile-time, rather than treating them as virtual methods whose location is only determined at run-time according to the run-time type of the object.

The lack of real typing, of inheritance and of public class fields, allows a truly independent compilation of classes. A class in Jack can be compiled without any access to the code of any other class: the fields of other classes are never accessed and all linking to methods of other classes is "late", done just by name.

Many other simplifications of the Jack language are not very significant and can be relaxed with little effort (but also little pedagogic gain). E.g. one may easily add a "for" statement to Jack, or character constants 'c'.

Finally, as usual, we did not pay any attention to optimization. Optimization is, of course, a main focus of attention in the code generation part of any compilation course.

5. Build it

Project 10 guidelines will be published in the web site.