# Workshop 11 - Code Generation

### Code Generation - Virtual Machine Code

##### [Practical Marker](https://cs.adelaide.edu.au/services/pracmarker/)

To simplify the task of writing a code generator for a recursive descent parser, we have provided working parsers, **parser11vm.cpp**, and **parser11xml.cpp**, the test programs from workshop10, **eg1** and **eg2**, as well as some helpful precompiled classes in the zip file attached below. To complete the workshop you can attempt either step 1 or step 2.

#### Step 1 - Code Generation - VM Code

Write a code generator for the simple programming language described in workshop 10 that will produce the equivalent Hack Virtual Machine code. You should do this by adding code to the recursive descent parser, **parser11vm.cpp**, that is in the zip file attached below.

##### Symbol Tables

Before you can generate correct code you need a way of recording where the variables declared in a program are actually stored. To do this you will need to use a symbol table. For this workshop you can use the supplied **symbols\_int** class as follows:

Declare global variables to hold the pointer to the symbol table and the next free location in the local segment:

static symbols\_int \*symbols = NULL ;

static int nextlocal = 0 ;

Add the following as the first line of the start of the parseProgram() function to create the symbol table:

symbols = symbols\_int::newtable() ;

Add two new functions to add a new variable to the table and one to look up a variable:

void declare\_variable(string identifier) // record the identifier and its location

{

if ( !symbols->insert(identifier,nextlocal++) ) // it is an error to declare something twice

{

cout << "Variable: " << identifier << " has already been declared" << endl ;

exit(-1) ;

}

}

int lookup\_variable(string identifier) // lookup the identifier, it is an error if it was not declared

{

int offset = symbols->lookup(identifier) ;

if ( offset == -1 )

{

cout << "Found undeclared variable: " << identifier << endl ;

exit(-1) ;

}

return offset ;

}

##### Virtual Machine Output

The virtual machine code produced should be prefixed by the following function header:

function Main.main x

where the value **x** is the number of identifiers declared in the program. The first identifier should be referred to as local 0, the second as local 1, and so on. You should be able to test your generated code by running your program in the **VMEmulator**.

To generate the code you can simply add appropriate output statements in the parsing functions. For example, the **parseLetStatement()** function could become:

void parseLetStatement()

{

string identifier = tokenvalue ;

int localOffset = lookup\_variable(identifier) ; // you will need to provide this function

mustbe("identifier") ;

mustbe("=") ;

parseExpression() ;

mustbe(";") ;

cout << "pop local " << localOffset << endl ;

}

What the extra code does, is first lookup the name of the variable being assigned to, then look up which position of the variable in its segment, then assumes that the **parseExpression()** function will generate code to place the value being assigned onto the top of the stack and finally it pops that value off the stack and into the variable. A little extra thought may be required when parsing the initial declarations so that the function header is output correctly. You should use the precompiled **symbols\_int** class to create the required symbol table for recording local segment offsets for each variable. The class declaration is provided in the **symbols.h** file.

#### Step 2 - Code Generation - XML

Write a code generator for the simple programming language described in workshop 10 that will produce an XML representation of the original program. You should do this by adding code to the recursive descent parser, **parser11xml.cpp**, that is in the zip file attached below. You should use the precompiled **wkxml** class to produce the XML output. The class declaration is provided in the file **wkxml.h**.

For example, the **parseProgram()** function could become:

void parseLetStatement()

{

wkxml::open\_node("letStatement") ;

string id = tokenvalue ;

wkxml::open\_node("identifier") ;  
 wkxml::node\_text(id) ;  
 wkxml::close\_node("identifier") ;

mustbe("identifier") ;

mustbe("=") ;

parseExpression() ;

mustbe(";") ;

wkxml::close\_node("letStatement") ;  
}

What this code does is, output the start of a new XML node with the name "**letStatement**", it then outputs an XML node with the name "**identifier**" that includes the actual name of the identifier as its content, it then assumes that the **parseExpression()** function will generate the appropriate XML nodes to represent the expression and finally it outputs the close of the "**letStatement**" XML node.