Project 11: Compiler II - Code Generation

Background

In this project we complete the construction of the Jack Compiler that we started building in the previous project.

Contract

Extend the syntax analyzer built in project 10 into a full-scale Jack compiler. The output of the compiler should be VM code designed to run on the virtual machine implemented in projects 7-8. Use your compiler to compile all the Jack programs listed below. Make sure that each translated program executes according to its documentation.

Resources

The relevant reading for this project is book chapter 11. You will need two tools: the programming language with which you implement your compiler, and the supplied VM emulator, for testing the code generated by your compiler. All the test files necessary for this project are available in projects/11 on your computer.

Proposed Implementation

Chapter 11 includes a proposed Compiler API, which can serve as your implementation's blueprint. The proposed implementation is based on morphing the syntax analyzer built in the previous project into a the full-scale compiler. In particular, we propose to gradually replace the software modules that generate XML output with software modules that generate VM code. This can be done in two main development stages, as follows.

Stage I: Symbol Table

We suggest to start by building the compiler's symbol table module and using it to extend the syntax analyzer built in project 10. Presently, whenever an identifier is encountered in the source code, say foo, the syntax analyzer outputs the XML line "<identifier> foo </identifier>". Instead, have your analyzer output the following information as part of its XML output (using some output format of your choice):

- The identifier's *name*, as done by the current version of the syntax analyzer.
- The identifier's *category*: var, argument, static, field, class, or subroutine.
- If the identifier's category is var, argument, static, or field, the *running index* assigned to the identifier by the symbol table.
- Whether the identifier is presently being *defined* (e.g. the identifier stands for a variable declared in a "var" statement) or *used*.

You may test your symbol table module and the above capability by running your extended syntax analyzer on the test Jack programs supplied in project 10. Once the output of your extended syntax analyzer will include the above information, it means that you have developed a complete executable capability to understand the grammatical structure of Jack programs. At this stage you can make the switch to a full-scale compiler, and start generating VM code instead of XML output. This can be done by gradually morphing the code of the extended syntax analyzer into a full-scale compiler.

Stage II: Code Generation

Since we gave many translation examples in chapter 11, we don't provide specific guidelines on how to develop the code generation features of the compiler. Rather, we provide a set of six application programs, designed to unit-test these features incrementally. We strongly suggest to test your compiler on these programs in the given order. This way, you will end up building the compiler's code generation capabilities in stages, according to the unfolding demands of each test program.

Testing Method: normally, when one compiles a program and runs into some problems, one concludes that the program is at fault. In this project though, the setting is exactly the opposite: all the Jack programs that we supply for testing purposes are error-free. Therefore, if their compilation yields any errors, it's the compiler that you have to fix, not the test programs.

For each test program, go through the following routine:

1. Compile the program directory using your compiler. This action should compile all the .jack files in the directory into corresponding .vm files;
2. Inspect the generated .vm files. If there are any visible problems, fix your compiler and go to step 1 (remember: all the supplied test programs are error-free);
3. Test the translated VM program by loading the program directory into the supplied VM emulator and executing it using the "no animation" mode. Each one of the six test programs contains specific execution

guidelines, as listed below; test the program according to these guidelines;
4. If the test program behaves unexpectedly or some error message is displayed by the VM emulator, fix your compiler and go to to step 1.

Testing

Each of the programs listed below is designed to gradually unit-test specific language handling capabilities of your compiler.

Test Program	Description	Purpose
Seven: Main.jack	Computes the value of the expression (3*2)+1 and prints the result at the top left of the screen. To test if your compiler has translated the program correctly, run the translated code in the supplied VM emulator and make sure that it displays 7 correctly.	Tests how your compiler handles a simple program containing three things: an arithmetic expression with integer constants (without variables), a do statement, and a return statement.
Conversion to binary: Main.jack	Unpacks a 16-bit number into its binary representation. The program takes the 16-bit value stored in RAM[8000] and stores its individual bits in RAM[8001..8016] (each location will contain 0 or 1). Before the conversion starts, the program initializes RAM[8001...8016] to -1. To test if your compiler has translated the program correctly, load the translated code into the supplied VM emulator and go through the following routine: • Put (interactively) some value in RAM[8000]; • Run the program for a few seconds, then stop its execution • Check (interactively) that RAM[8001...8016] contain the correct results, and that none of them contains -1.	Tests how your compiler handles all the procedural elements of the Jack language, i.e.expressions, functions , and all the language statements. The program does not test the handling of methods, constructors, arrays, strings, static variables and field variables.
Square Dance: Main.jack Square.jack SquareGame.jack	A simple interactive app, described in detail in project 9. Enables moving a graphical square around the screen using the keyboard's four arrow keys. While moving, the size of the square can be increased / decreased by pressing the "z" and "x" keys, respectively. To quit the app, press the "q" key. To test if your compiler has translated the source code correctly, run the translated code in the supplied VM emulator and make sure that it works according to the above description.	Tests how your compiler handles the object oriented constructs of the Jack language: constructors, methods, fields and expressions that include method calls. The program does not test the handling of static variables.
Average: Main.jack	Computes the average of a user-supplied sequence of integers. To test if your compiler has translated the program correctly, run the translated code in the supplied VM emulator and follow the instructions displayed on the screen.	Tests how your compiler handles arrays and strings.
Pong: Main.jack Ball.jack Bat.jack PongGame.jack	In this classical game, a ball is moving randomly, bouncing off the screen "walls". The user can move a small paddle horizontally by pressing the keyboard's left and right arrow keys. Each time the paddle hits the ball, the user scores a point and the paddle shrinks a little, to make the game	Provides a complete test of how your compiler handles objects and static variables.

Test Program	Description	Purpose
	slightly more challenging. If the user misses and the ball hits the bottom, the game is over. To test if your compiler has translated this program correctly, run the translated code in the supplied VM emulator and play the game. Make sure to score some points, to test the part of the program that displays the score on the screen..	
Complex Arrays: Main.jack	Performs five complex calculations using arrays. For each such calculation, the program prints on the screen the expected result versus the actual result (as performed by the compiled program). To test if your compiler has translated the program correctly, run the translated code in the supplied VM emulator and make sure that the actual results are identical to the expected results.	Tests how your compiler handles complex array references and expressions.