11.5 Project

Objective Extend the syntax analyzer built in chapter 10 into a full-scale Jack compiler. In particular, gradually replace the software modules that generate passive XML code with software modules that generate executable VM code.

Resources The main tool that you need is the programming language in which you will implement the compiler. You will also need an executable copy of the Jack operating system, as explained below. Finally, you will need the supplied VM Emulator, to test the code generated by your compiler on a set of test programs supplied by us.

Contract Complete the Jack compiler implementation. The output of the compiler should be VM code designed to run on the virtual machine built in the projects in chapters 7 and 8. Use your compiler to compile all the Jack programs given here. Make sure that each translated program executes according to its documentation.

Stage 1: Symbol Table

We suggest that you 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 program, 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 format of your choice):

- the 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 used (e.g., the identifier stands for a variable in an expression);
- whether the identifier represents a variable of one of the four kinds (var, argument, static, field), and the running index assigned to the identifier by the symbol table.

You may test your symbol table module and the preceding capability by running your (extended) syntax analyzer on the test Jack programs supplied in Project 10. Once the output of your extended syntax analyzer includes this information, it means that you have developed a complete executable capability to understand the semantics 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 compiler.

Stage 2: Code Generation

We don't provide specific guidelines on how to develop the code generation features of the compiler, though the examples spread throughout the chapter are quite instructive. Instead, 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 be implicitly guided to build the compiler's code generation capabilities in stages, according to the demands of each test program.

The Operating System The Jack OS—the subject of chapter 12—was written in the Jack language. The source OS code was then translated (by an error-free Jack compiler) into a set of VM files, collectively known as the Jack OS. Each time we want to run an application program on the VM emulator, we must load into the emulator not only the application's .vm files, but also all the OS .vm files. This way, when an application-level VM function calls some OS-level VM function, they will find each other in the same environment.

Testing Method Normally, when you compile a program and run into some problems, you conclude that the program is screwed up and proceed to debug it. In this project the setting is exactly the opposite. All the test programs that we supply 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, we recommend going through the following routine:

1. Copy all the supplied OS .vm files from tools/OS into the program directory, together with the supplied .jack file(s) comprising the test program.

2. Compile the program directory using your compiler. This operation should compile only the .jack files in the directory, which is exactly what we want.

3. If there are any compilation errors, fix your compiler and return to step 2 (note that all the supplied test programs are error-free).

4. At this point, the program directory should contain one .vm file for each source . jack file, as well as all the supplied OS .vm files. If this is not the case, fix your compiler and return to step 2.

5. Execute the translated VM program in the VM Emulator, loading the entire directory and using the "no animation" mode. Each one of the six test programs contains specific execution guidelines, as listed here.

6. If the program behaves unexpectedly or some error message is displayed by the VM emulator, fix your compiler and return to step 2.

Test Programs

We supply six test programs. Each program is designed to gradually unit-test specific language handling capabilities of your compiler.

Seven This program computes the value of (3*2)+1 and prints the result at the top left of the screen. To

est whether your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that it displays 7 correctly. Purpose: Tests how your compiler handles a simple program containing an arithmetic expression with integer constants (without variables), a do statement and a return statement.

Decimal-to-Binary Conversion This program converts a 16-bit decimal number into its binary representation. The program takes a decimal number from RAM [8000], converts it to binary, and stores the 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 whether your compiler has translated the program correctly, load the translated code into the VM emulator and go through the following routine:

- Put (interactively) a 16-bit decimal 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.

Purpose: Tests how your compiler handles all the procedural elements of the Jack language, namely, expressions (without arrays or method calls), 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 This program is a trivial interactive "game" that enables moving a black square around the screen using the keyboard's four arrow keys. While moving, the size of the square can be increased and decreased by pressing the "z" and "x" keys, respectively. To quit the game, press the "q" key. To test if your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that it works according to this description. Purpose: Tests how your compiler handles the object-oriented constructs of the Jack language: constructors, methods, fields and expressions that include method calls. It does not test the handling of static variables.

Average This program 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 VM emulator and follow the instructions displayed on the screen. Purpose: Tests how your compiler handles arrays and strings.

Pong A ball is moving randomly on the screen, bouncing off the screen "walls." The user can move a small bat horizontally by pressing the keyboard's left and right arrow keys. Each time the bat hits the ball, the user scores a point and the bat shrinks a little, to make the game harder. If the user misses and the ball hits the bottom horizontal line, the game is over. To test whether your compiler has translated this program correctly, run the translated code in the 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). Purpose: Provides a complete test of how your compiler handles objects, including the handling of static variables.

Complex Arrays 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 whether your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that the actual results are identical to the expected results. Purpose: Tests how your compiler handles complex array references and expressions.