Project 11

The compiler built in Nand to Tetris consists of two main modules: a Syntax Analyzer, built in project 10, and a Code Generator, built in this project. The former module is the point of departure for the latter module. In particular, in this project you will morph the Syntax Analyzer built in project 10 into a full-scale Jack compiler. You will do it by gradually replacing the routines that generate passive XML output with routines that generate executable VM code.

Objective

Extend the Syntax Analyzer built in chapter 10 into a full-scale Jack compiler. Apply your evolving compiler to all the test programs described below. Execute each translated program, and make sure that it operates according to its given documentation. This version of the compiler assumes that the source Jack code is error-free.

Resources

The main tool that you need is the programming language in which you will implement the compiler. You will also need the supplied VM emulator, for testing the VM code generated by your compiler. Since the compiler is implemented by extending the syntax analyzer built in project 10, you will also need the analyzer's source code.

Implementation Stages

We propose completing the compiler's development in two main stages: Symbol Table, and Code Generation.

Stage 0: Make a back-up copy of the syntax analyzer code developed in project 10.

Stage 1: Symbol Table: Start by building the compiler's SymbolTable module, and use it for extending the syntax analyzer built in Project 10, as follows. Presently, whenever an identifier is encountered in the source code, say foo, the syntax analyzer outputs the XML line <identifier> foo </identifier>. Extend your analyzer to output the following information about each identifier:

name;

category (field, static, local, arg, class, subroutine);

index: if the identifier's category is field, static, local, or arg,
the running index assigned to the identifier by the symbol table;

usage: whether the identifier is presently being *declared* (in a static / field / var variable declaration, or in a parameter list), or *used* (in an expression).

Have your syntax analyzer output this information as part of its XML output, using markup tags of your choice.

Test your new SymbolTable module and the new functionality just described by running your extended syntax analyzer on the test Jack programs supplied in project 10. If your extended syntax analyzer is capable of outputting the information described above, it means that you've developed a complete executable capability to understand the semantics of Jack programs. At this stage you

can make the switch to developing the full-scale compiler, and start generating VM code instead of XML output. This can be done gradually, as we now turn to describe.

Stage 1.5: Make a back-up copy of the extended syntax analyzer code.

Stage 2: Code Generation: We provide six Jack programs, designed to gradually unit-test the code generation capabilities of your compiler. We advise to develop, and test, your evolving compiler on the test programs in the order given below. This way, you will be implicitly guided to build the compiler's code generation capabilities in sensible stages, according to the demands presented by each test program.

Normally, when one compiles a high-level program and runs into some difficulties, one concludes that the program is screwed up. In this project the setting is exactly the opposite. All the supplied test programs are error-free. Therefore, if their compilation yields any errors, it's the compiler that you have to fix, not the programs. Specifically, for each test program, we recommend going through the following routine:

1. Compile the program folder using the compiler that you are developing. This action should generate one .vm file for each source .jack file in the given folder.

2. Inspect the generated VM files. If there are visible problems, fix your compiler and go to step 1. Remember: All the supplied test programs are error-free.

3. Load the program folder into the emulator, and run the loaded code. Note that each one of the six supplied test programs contains specific execution guidelines; test the compiled program (translated VM code) according to these guidelines.

4. If the program behaves unexpectedly, or some error message is displayed by the VM emulator, fix your compiler and go to step 1.

Test Programs

Seven: Tests how the compiler handles a simple program containing an **arithmetic expression** with integer constants but without variables, a **do** statement, and a **return** statement. Specifically, the program computes the expression 1 + (3 * 2) and prints its value at the top left of the screen. To test that your compiler has translated the program correctly, run the translated code in the VM emulator and verify that it displays 7 correctly.

ConvertToBin: Tests how the compiler handles all the procedural elements of the Jack language: **expressions** (without arrays or method calls), **functions**, and the statements **if**, **while**, **do**, **let** and **return**. The program does not test the handling of methods, constructors, arrays, strings, static variables, and field variables. Specifically, the program gets a 16-bit decimal value from RAM[8000], converts it to binary, and stores the individual bits in RAM[8001],...,RAM[8016] (each location will contain 0 or 1). Before the conversion starts, the program initializes RAM[8001],...,RAM[8016] to -1. To test that your compiler has translated the program correctly, load the translated code into the VM emulator, and proceed as follows:

Enter (interactively, using the simulator's GUI) some decimal value in RAM[8000];

Run the program for a few seconds, then stop its execution;

Check (by visual inspection) that memory locations RAM[8001],...,RAM[8016] contain the correct bits, and that none of them contains -1.

Square (same as projects/9/Square): Tests how the compiler handles the object-based features of the Jack language: **constructors**, **methods**, **fields**, and **expressions** that include **method calls**. Does not test the handling of static variables. Specifically, the program launches a simple 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 anytime by pressing the 'z' and 'x' keys, respectively. To quit the game, press the 'q' key. To test that your compiler has translated the program correctly, run the translated code in the VM emulator and verify that the game unfolds as expected (to play the game, select "no animation" and set the speed slider as needed).

Average (same as projects/9/Average and projects/10/ArrayTest): : Tests how the compiler handles **arrays**, and **strings**. This is done by computing the average of a user-supplied sequence of integers. To test that your compiler has translated the program correctly, run the translated code in the VM emulator and follow the instructions displayed on the screen.

Pong: A complete test of how the compiler handles an object-based application, including the handling of **objects** and **static variables**. In the classical Pong game, a ball is moving randomly, bouncing off the screen "walls." The user tries to hit the ball with a small paddle which can be moved 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, making the game increasingly more challenging. If the user misses and the ball hits the "floor", the game is over. To test that 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 (to play the game, select "no animation" and set the speed slider as needed).

ComplexArrays: Tests how the compiler handles **complex array** references, and **expressions**. To that end, the program performs five complex calculations using arrays. For each such calculation, the program prints on the screen the expected result, along with the actual result, as performed by the compiled program. To test that your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that the expected and actual results are identical.