CSC488
ASSIGNMENT 5
TESTING

DANIEL BLOEMENDAL

Contents

1. Overview

For code generation our testing methodology broke into two categories, the methodology for passing tests and the methodology for failing tests.

2. Passing tests

For passing tests we were looking for correct execution of programs. The only way to establish that other than monitoring the machine state during execution is to perform input/output operations. If you think of a program as computing a Turing computable function, then the way to test whether or not the program is indeed computing the function is to ensure that for every input it achieves the expected output. We enhanced the test runner from assignment 3 to with two new commands in the comment meta language, "`@input=<string>`" and "`@output=<string>`". The test runner feeds tested programs the inputs contained in the `@input` comment commands sequentially via `stdin` and then looks for the expected output contained in the `@output` commands via `stdout`.

The passing tests themselves focused on exercising as many language features as possible. For instance, one of our original sample programs A1a.488 used all of the operators available in expressions. To test for correctness we needed only annotate the programs with the output meta commands looking for the expected output. In addition to basic tests of language features we also included a number of more complicated algorithms to stress all aspects of code generation. We also included tests for specific edge cases such as deeply nested minor scopes, to ensure that our coalescence of minor scopes was fully functional.

3. Failing tests

For the failing tests, there were three categories. Tests we expected to fail during code generation, prefixed by `codegen_`, tests we expected to fail at assembly, prefixed by `assembler_` and tests we expected to fail at runtime, prefixed by `runtime_`. For the code generator and assembler errors, we expected graceful handling of the errors. These were all errors related to machine limitations. For instance one failing test has a very large scope, achieved by declaring lots of large arrays, which hits the available stack space limit.

4. Tests

Test	Author	Description
`pass/A1a.io.488`	Daniel Bloemendal	Tests all logical, comparison, and arithmetic operators
`pass/A1b.io.488`	Daniel Bloemendal	Tests using arrays including both forms of array declaration, positive and negative bounds
`pass/A1c.io.488`	Oren Watson	Tests all forms of loop building and loop exit constructs
`pass/ackermann.488`	Oren Watson	Computes the Ackermann function
`pass/bsort.488`	Simon Scott	An implementation of bubble sort
`pass/cash_register.488`	Oren Watson	Performs some functions of a cash register, including computing subtotal, taxes and total
`pass/collatz.488`	Oren Watson	Verifies Collatz conjecture for input numbers
`pass/dectohex.488`	Oren Watson	Converts a decimal number into hexadecimal
`pass/deeply_nested.488`	Oren Watson	A program with deeply nested minor scopes
`pass/euclid.488`	Simon Scott	An implementation of the euclidean algorithm
`pass/four_fn_calc.488`	Oren Watson	Four function calculator
`pass/multi_level_assignment.488`	Oren Watson	Tests assignment to variables at different levels
`pass/outputspec.488`	Oren Watson	Tests the output in a simple proof of concept
`pass/qsort.488`	Mike Qin	An implementation of quicksort
`pass/quicksort.488`	Oren Watson	Quicksort, using inner functions instead of passing variables
`pass/tailcall.488`	Daniel Bloemendal	A program with recursion at the tail of a function (use compiler option “`-OPT t`”, tail call optimization)
`fail/assembler_large_program.488`	Daniel Bloemendal	A large program whose generated code exceeds machine memory
`fail/codegen_large_arrays.488`	Daniel Bloemendal	Defines an array whose size exceeds machine memory
`fail/codegen_large_integer.488`	Daniel Bloemendal	Uses an integer outside of the integer bounds of the machine
`fail/codegen_large_locals.488`	Daniel Bloemendal	Declares a scope whose locals size exceeds available stack memory
`fail/codegen_large_string.488`	Daniel Bloemendal	Uses a string whose length is greater than 255 characters
`fail/runtime_bounds1d.488`	Daniel Bloemendal	Out of bounds access to a 1D array (use compiler option “`-B s`” or “`-B e`”, simple or enhanced bounds checking)
`fail/runtime_bounds2d.488`	Daniel Bloemendal	Out of bounds access to a 2D array (use compiler option “`-B s`” or “`-B e`”, simple or enhanced bounds checking)
`fail/runtime_deeply_nested.488`	Daniel Bloemendal	A program with deeply nested major scopes
`fail/runtime_infinite_recursion.488`	Daniel Bloemendal	Infinitely recursive function, leading to a stack overflow
`fail/runtime_integer_overflow.488`	Daniel Bloemendal	Performs arithmetic operations that lead to an overflow
`fail/runtime_undefined_return.488`	Daniel Bloemendal	Declares a function without a result statement, and attempts to use the undefined return value