Homework #8: CMPT-379

Distributed on Mon, Nov 27; Due on Mon, Dec 4

Anoop Sarkar – anoop@cs.sfu.ca

This assignment finally wraps up the compiler that you have been building throughout this semester. In this final step, you will implement some basic semantic checks, and code generation to MIPS assembly for the entire **Decaf** reference grammar.

(1) **Code Generation**:

As before, the target of the code generation step will be MIPS R2000 assembly language. You should augment your code generator for the sub-grammar from the last assignment to cover the entire **Decaf** syntax. Mostly, this means adding in additional code generation steps that deal with control flow statements like `for` and `while` loops, `if` statements and the like in **Decaf**.

In this assignment, you should try to implement register spilling. However, I will not enforce this strictly, if your code generation step runs out of registers to use, your program can exit with an error message.

Once you have implemented code generation for the remainder of **Decaf** syntax, you can test with **Decaf** programs like `catalan.decaf` and `gcd.decaf` into MIPS and run it using `spim`. Submit the entire compiler pipeline which accepts **Decaf** code and produces MIPS assembly that can then be run using `spim`.

(2) **Semantic Checking**

Perform at least the following semantic checks for any syntactically valid input **Decaf** program:

a. A method called **main** has to exist in the **Decaf** program.

b. Find all cases where there is a type mismatch between the definition of the type of a variable and a value assigned to that variable. e.g. `bool x; x = 10;` is an example of a type mismatch.

c. Find all cases where an expression is well-formed, where binary and unary operators are distinguished from relational and equality operators. e.g. `true + false` is an example of a mismatch but `true != true` is not a mismatch.

d. Check that all variables are defined in the proper scope before they are used as an lvalue or rvalue in a **Decaf** program (see below for hints on how to do this).

e. Check that the return statement in a method matches the return type in the method definition. e.g. `bool foo() { return(10); }` is an example of a mismatch.

To do these semantic checks you will need to use either your attribute grammar implementation or tree-matching implementation. In addition, you will need to use the symbol table which stores with each identifier specification, the type of the identifier (for variables) or typed parameter list and return type (for methods). You will need to know proper scoping, that is, when the variable or method where the variable or method is *alive* in the program. The implementation of mutually recursive methods is optional in this homework. You can assume that each method is defined before it is called in all input **Decaf** programs.

Raise a semantic error if the input **Decaf** program does not pass any of the above semantic checks.

Submit a program that takes a syntactically valid **Decaf** program as input and performs all the semantic checks listed above. You can include any other semantic checks that seem reasonable based on your analysis of the language. Provide a readme file with a description of any additional semantic checks.

(3) **Compiler Contest** (optional; not graded):

Submit your compiler as a self-contained package that can be used to compile **Decaf** programs into MIPS assembly and subsequently execute them using the `spim` simulator for the MIPS processor. Make sure that your compiler can be compiled by running `make` or a script called `compileit`. Create a script called `decafcc` (or `decafcc.sh`) that is used to run the entire compiler chain from lexical analysis to code generation to running the MIPS simulator (assume `spim` is in the PATH).

In your submission, provide in a subdirectory called `positives` any number of **Decaf** programs that work with your compiler (the programs should be valid **Decaf** based on the language definition and execute using `spim`) along with the legitimate output for that **Decaf** program, e.g. for a program called `exprTest.decaf` also include the legitimate output in a file called `exprTest.decaf.output`. Also provide a subdirectory called `negatives` with **Decaf** programs that should exit with an error. Your `makefile` should include an entry such that when `make testall` is run, it should run your **Decaf** compiler on *all* the **Decaf** programs in the `positives` and `negatives` directory.

You could try to break the compilers written by your peers, but only if your compiler can survive those **Decaf** programs itself. For instance, a **Decaf** compiler that implements register spilling will be more robust to **Decaf** programs that use up a lot of registers and could be used as a `positives` **Decaf** program to boost your own (unofficial) ranking.