Homework #8: CMPT-379

Distributed on Thu, Nov 24; Due on Thu, Dec 8

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) **Semantic Checking**

Use the parse trees produced by your parser to perform the following semantic checks in an input **Decaf** program:

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

To do these semantic checks you will need to traverse the parse tree. In addition, you will need to implement a symbol table which stores each identifier, the type of the identifier or return type, and the node in the parse tree which indicates the scoping for the identifier.

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

Submit a program that takes parse trees for **Decaf** 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.

(2) **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 dealing with control flow statements like `for` loops, `if` statements and the like, dealing with function calls, especially recursive functions, and the use of arrays in **Decaf**.

In this assignment, you can choose 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`. Create the entire compiler pipeline which accepts **Decaf** code and produces MIPS assembly that can then be run using `spim` as shown above.

Save the MIPS assembly program to file `filename.mips` and run the simulator `spim` as follows:

```
spim -file <filename.mips>
```

(3) **Compiler Contest**:

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 ranking.

Each compiler will be tested with all the programs submitted and some other **Decaf** programs. The compilers will be ranked informally, and the ranking of your compiler will depend on several factors. One will be how many of the positives successfully produce legitimate output how many of the negatives exit with an error. In addition, a compiler with an elegant implementation will be ranked higher, as will those who do not use code that was released as solutions to previous homeworks.