Homework #7: CMPT-379

Distributed on Tue, Nov 9; Due on Tue, Nov 23

Anoop Sarkar – anoop@cs.sfu.ca

(1) **Producing Output from the Parser**

This homework takes the LR parser you have built so far for **Decaf** and augments the parser with the backend step, implementing semantic analysis and the code generation portion of the compiler. You have a couple of options in how to implement semantic checking and code-generation on top of your LR parser:

- Implement synthesized and/or inherited attribute passing on top of your LR parser. A reduce action can pass synthesized attributes in the parser stack (information such as MIPS assembly code fragments). For inherited L-attributes, goto actions store information on the stack where subsequent rule predictions can access the inherited attributes by peeking into the stack. Source code to implement the attribute grammar can be stored along with the CFG rules and compiled into the parser.
- Implement code generation by reading in the parse tree and producing code via tree rewriting using the techniques and algorithms given in Chapter 9 of the Dragon book. In this option, you will have to write tree patterns which match the output **Decaf** parse trees and produce output information (like MIPS code fragments and a target register location). The output will be created by pasting individual tree patterns until it covers the parse tree. The matching can be done top-down (greedy) or bottom-up (dynamic programming).

(2) **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.

(3) **Code Generation**:

The target of the code generation step will be MIPS R2000 assembly language. We will treat MIPS assembly code as a *virtual machine* and use an simulator for MIPS assembly called `spim` that takes MIPS assembly and simulates (runs) it on x86 Linux. `spim` is available for your use from the location mentioned on the course web page.

Chapter 8 in the Dragon Book provides a case-by-case treatment of code generation issues for each kind of statement in **`Decaf`**. For this assignment, you can assume that function calls will have no more than four arguments. In MIPS assembly, upto four arguments can be passed directly to a subroutine in the registers `$a0-$a3`.

In addition to stack/tree manipulation, you have to manage the register names used in the output assembly code. For this assignment, we will ignore some of the complexities of code generation by assuming that we have a sufficient number of temporary registers at hand. The MIPS target machine allows the use of the following registers: `$a0-$a3, $t0-$t9, $s0-$s7`. Your program should use the algorithm for stacked temporary registers explained in Section 8.3 (page 480) of the Dragon book. However, if despite using this algorithm if your code generation step runs out of registers to use, your program can exit with an error message.

The standard input-output library is provided through the `syscall` interface (compiled into `spim`).

I/O library service	`syscall` code	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_string`	4	`$a0` = string	
`read_int`	5		integer in `$v0`
`read_string`	8	`$a0` = buffer, `$a1` = length	
`exit`	10		

I/O should be done only using the `syscall` service. Do not use the `jal printf` idiom used in some examples in the MIPS/`spim` documentation.

Once you have implemented procedure calls, if/while/for-statements, array lvalues & rvalues and all arithmetic and boolean operators then 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` as shown above.

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

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

(4) **Submission Procedure:** Create a shell script called `decafcc` or `decafcc.sh` which should do all the phases of compilation, from lexical analysis, to parsing, to the code generation step, and running `spim` on the MIPS assembly (assume `spim` is in the PATH).