Homework #7: CMPT-379

Distributed on Tue, Nov 15; Due on Thu, Nov 24

Anoop Sarkar – `anoop@cs.sfu.ca`

(1) **Code Generation (First Steps)**

In this homework, you make the first step towards full code generation in **Decaf**. In the first stage, implement the steps listed below for the following fragment of **Decaf** syntax:

$$
\begin{array}{rcl}
\langle\text{block}\rangle & \rightarrow & \textbf{`\{'}\ \langle\text{var-decl}\rangle^{*}\ \langle\text{statement}\rangle^{*}\ \textbf{`\}'} \\
\langle\text{var-decl}\rangle & \rightarrow & \langle\text{type}\rangle \left\{ \textbf{id} \right\}^{+}\textbf{,}\ \textbf{`;'} \\
\langle\text{type}\rangle & \rightarrow & \textbf{int} \mid \textbf{bool} \\
\langle\text{statement}\rangle & \rightarrow & \langle\text{assign}\rangle\ \textbf{`;'} \mid \langle\text{method-call}\rangle\ \textbf{`;'} \mid \langle\text{block}\rangle \\
\langle\text{assign}\rangle & \rightarrow & \langle\text{lvalue}\rangle\ \textbf{`='}\ \langle\text{expr}\rangle \\
\langle\text{method-call}\rangle & \rightarrow & \textbf{callout}\ \textbf{`('}\ \textbf{stringConstant} \left[ \left\{ \textbf{`,'} \left\{ \langle\text{callout-arg}\rangle \right\}^{+}\textbf{,} \right\} \right] \textbf{`)'} \\
\langle\text{callout-arg}\rangle & \rightarrow & \langle\text{expr}\rangle \mid \textbf{stringConstant} \\
\langle\text{lvalue}\rangle & \rightarrow & \textbf{id} \\
\langle\text{expr}\rangle & \rightarrow & \langle\text{lvalue}\rangle \\
 & \mid & \langle\text{method-call}\rangle \\
 & \mid & \langle\text{constant}\rangle \\
 & \mid & \langle\text{expr}\rangle\ \langle\text{bin-op}\rangle\ \langle\text{expr}\rangle \\
 & \mid & \textbf{`-'}\ \langle\text{expr}\rangle \\
 & \mid & \textbf{`!'}\ \langle\text{expr}\rangle \\
 & \mid & \textbf{`('}\ \langle\text{expr}\rangle\ \textbf{`)'} \\
\langle\text{bin-op}\rangle & \rightarrow & \langle\text{arith-op}\rangle \mid \langle\text{rel-op}\rangle \mid \langle\text{eq-op}\rangle \mid \langle\text{cond-op}\rangle \\
\langle\text{arith-op}\rangle & \rightarrow & \textbf{`+'} \mid \textbf{`-'} \mid \textbf{`*'} \mid \textbf{`/'} \mid \textbf{`<<'} \mid \textbf{`>>'} \mid \textbf{`\%'} \mid \textbf{rot} \\
\langle\text{rel-op}\rangle & \rightarrow & \textbf{`<'} \mid \textbf{`>'} \mid \textbf{`<='} \mid \textbf{`>='} \\
\langle\text{eq-op}\rangle & \rightarrow & \textbf{`=='} \mid \textbf{`!='} \\
\langle\text{cond-op}\rangle & \rightarrow & \textbf{`\&\&'} \mid \textbf{`||'} \\
\langle\text{constant}\rangle & \rightarrow & \textbf{intConstant} \mid \textbf{charConstant} \mid \langle\text{bool-constant}\rangle \\
\langle\text{bool-constant}\rangle & \rightarrow & \textbf{true} \mid \textbf{false}
\end{array}
$$

This fragment of **Decaf** essentially represents the expression-level sub-grammar. Note that you should use your LR(0) or SLR(1) implementation of this reference sub-grammar from your previous work. Here's an example input for this sub-grammar:

```
{
  int x; bool y; x = 2+2*3+5+(43*23-44);
  { int y; y = x * -30 + 40 * 2; x = -y; }
  y = !true; callout("print_int", x);
}
```

You have a couple of options for implementing 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).

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`**.

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. Optionally, if you are done with the rest of the homework, you can use the heap to store values associated with identifiers, and load them when needed into a register. This will allow the re-use of registers (using register *spilling* to the heap). See `using-heap.mips` on the course web page.

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.

Submit the entire compiler pipeline which accepts **`Decaf`** code and produces MIPS assembly. Save the MIPS assembly program to file `filename.mips` and run the simulator `spim` as follows:

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

(2) **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` and the other binaries you need are in the PATH).