Homework #4: CMPT-379

Distributed on Nov 12; due on Nov 30

Anoop Sarkar – anoop@cs.sfu.ca

Only submit answers for questions marked with †. Provide a `makefile` such that `make` compiles all your programs, and `make test` runs each program, and supplies the necessary input files.

(1) **Code generation for global variables and methods**

The following fragment of **Decaf** extends Q. 5 of Homework #3 to include the definition of global variables, method definitions and method calls. Note that global array variables are the only kind of arrays allowed in **Decaf**. Extend your previous code generation implementation to deal with this fragment of **Decaf** syntax.

⟨program⟩	→	**class** ⟨class-name⟩ **'{'** ⟨field-decl⟩ * ⟨method-decl⟩ * **'}'**
⟨class-name⟩	→	**id**
⟨field-decl⟩	→	⟨type⟩ { **id** | { **id '['** **intConstant '']'** } }+**,** **';'**
	|	⟨type⟩ **id '='** ⟨constant⟩ **';'**
⟨method-decl⟩	→	{ ⟨type⟩ | **void** } **id '('** [{ ⟨type⟩ **id** }+**,**] **')'** ⟨block⟩
⟨block⟩	→	**'{'** ⟨var-decl⟩ * ⟨statement⟩ * **'}'**
⟨var-decl⟩	→	⟨type⟩ { **id** }+**,** **';'**
⟨type⟩	→	**int** | **bool**
⟨statement⟩	→	⟨assign⟩ **';'** | ⟨method-call⟩ **';'** | ⟨block⟩ | **return** [⟨expr⟩] **';'**
⟨assign⟩	→	⟨lvalue⟩ **'='** ⟨expr⟩
⟨method-call⟩	→	⟨method-name⟩ **'('** [{ ⟨expr⟩ }+**,**] **')'**
	|	**callout '('** **stringConstant** [{ **','** { ⟨callout-arg⟩ }+**,** }] **')'**
⟨method-name⟩	→	**id**
⟨callout-arg⟩	→	⟨expr⟩ | **stringConstant**
⟨lvalue⟩	→	**id** | **id '['** ⟨expr⟩ **']'**
⟨expr⟩	→	⟨lvalue⟩ | ⟨method-call⟩ | ⟨constant⟩
	|	⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
	|	**'-'** ⟨expr⟩ | **'!'** ⟨expr⟩ | **'('** ⟨expr⟩ **')'**
⟨bin-op⟩	→	⟨arith-op⟩ | ⟨rel-op⟩ | ⟨eq-op⟩ | ⟨cond-op⟩
⟨arith-op⟩	→	**'+'** | **'-'** | **'*'** | **'/'** | **'<<'** | **'>>'** | **'%'** | **rot**
⟨rel-op⟩	→	**'<'** | **'>'** | **'<='** | **'>='**
⟨eq-op⟩	→	**'=='** | **'!='**
⟨cond-op⟩	→	**'&&'** | **'||'**
⟨constant⟩	→	**intConstant** | **charConstant** | ⟨bool-constant⟩
⟨bool-constant⟩	→	**true** | **false**

Your program should implement register spilling either to the stack or the heap. Your program should allow method definitions and method calls with an arbitrary number of parameters by generating the appropriate stack frame for method calls, especially recursive method calls.

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

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

(2) **Code generation for control-flow & loops**

The following fragment of **Decaf** syntax should be added to the grammar in Q. 1. It adds control flow (**if** statements) and loops (**while** and **for** statements) to **Decaf**.

$$
\begin{aligned}
\langle \text{statement} \rangle \quad \rightarrow \quad & \textbf{if `('} \langle \text{expr} \rangle \textbf{`)'} \langle \text{block} \rangle \left[\textbf{else} \ \langle \text{block} \rangle \right] \\
| \quad & \textbf{while `('} \langle \text{expr} \rangle \textbf{`)'} \langle \text{block} \rangle \\
| \quad & \textbf{for `('} \left\{ \langle \text{assign} \rangle \right\}^{+}\textbf{,`;'} \langle \text{expr} \rangle \textbf{`;'} \left\{ \langle \text{assign} \rangle \right\}^{+}\textbf{,`)'} \langle \text{block} \rangle \\
| \quad & \textbf{break `;'} \\
| \quad & \textbf{continue `;'}
\end{aligned}
$$

Your program must implement short-circuit evaluation for the **if** statement.

As before, save the MIPS assembly program to file `filename.mips` and run the simulator `spim` as follows:

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

(3) **Semantic checks**

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.

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.

Your program should take a syntactically valid **Decaf** program as input and perform all the semantic checks listed above. You can optionally 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.

(4) † **The `Decaf` compiler**

Combine all the **Decaf** fragments you have implemented so far (Q. 5 from Homework #3 and Q. 1 and Q. 2 from this homework). Create a single yacc/lex program that accepts any syntactically valid **Decaf** programs and produces MIPS assembly language output. Your program should also perform the semantic checks defined in Q. 3 above.

Submit your compiler as a self-contained package that can be used to compile **Decaf** programs into MIPS assembly and subsequently execute it using the `spim` simulator for the MIPS processor. Make sure that your compiler can be compiled by running `make`. Create a shell script called `decafcc` (or `decafcc.sh`) that accepts a **Decaf** program as input and runs the MIPS simulator on the code generation output of your compiler (assume `spim` is in the PATH).