Homework #4: CMPT-379

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` takes each **Decaf** program in the `../testcases` directory and provides the LLVM output.

(1) **Extern Definitions and Recursive Functions**
Start with your solution to Homework #3 and add support for extern definitions and recursive function calls. The fragment of the **Decaf** grammar for this part is shown below. Using this sub-grammar will clean up the ad-hoc code for `print_int` calls and the top-level `main` function definition in Homework #3.

⟨program⟩	→	⟨extern-defn⟩ * **class id** ‘{’ ⟨method-decl⟩ * ‘}’
⟨extern-defn⟩	→	**extern** ⟨method-type⟩ **id** **‘(’** [{ ⟨extern-type⟩ } +,] **‘)’** **‘;’**
⟨extern-type⟩	→	**string** | ⟨type⟩
⟨method-decl⟩	→	⟨method-type⟩ **id** **‘(’** [{ ⟨type⟩ **id** } +,] **‘)’** ⟨block⟩
⟨block⟩	→	‘{’ ⟨var-decl⟩ * ⟨statement⟩ * ‘}’
⟨var-decl⟩	→	⟨type⟩ { **id** } +, **‘;’**
⟨method-type⟩	→	**void** | ⟨type⟩
⟨type⟩	→	**int** | **bool**
⟨statement⟩	→	⟨assign⟩ **‘;’**
	|	⟨method-call⟩ **‘;’**
	|	⟨block⟩
	|	**return** [**‘(’** [⟨expr⟩] **‘)’**] **‘;’**
⟨assign⟩	→	⟨ℓ-value⟩ **‘=’** ⟨expr⟩
⟨method-call⟩	→	**id** **‘(’** [{ ⟨method-arg⟩ } +,] **‘)’**
⟨method-arg⟩	→	⟨expr⟩ | **stringConstant**
⟨ℓ-value⟩	→	**id**
⟨expr⟩	→	**id**
	|	⟨method-call⟩
	|	⟨constant⟩
	|	⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
	|	**‘-’** ⟨expr⟩
	|	**‘!’** ⟨expr⟩
	|	**‘(’** ⟨expr⟩ **‘)’**
⟨bin-op⟩	→	⟨arith-op⟩ | ⟨rel-op⟩ | ⟨eq-op⟩ | ⟨cond-op⟩
⟨arith-op⟩	→	**‘+’** | **‘-’** | **‘*’** | **‘/’** | **‘<<’** | **‘>>’** | **‘%’**
⟨rel-op⟩	→	**‘<’** | **‘>’** | **‘<=’** | **‘>=’**
⟨eq-op⟩	→	**‘==’** | **‘!=’**
⟨cond-op⟩	→	**‘&&’** | **‘||’**
⟨constant⟩	→	**intConstant** | **charConstant** | ⟨bool-constant⟩
⟨bool-constant⟩	→	**true** | **false**

(2) **Global variables**

Add support for global variables. The modified rules for the fragment of **Decaf** is given below.

$$
\begin{array}{rcl}
\langle\text{program}\rangle & \rightarrow & \langle\text{extern-defn}\rangle^{*}\ \ \textbf{class id}\ \text{`\{'}\ \langle\text{field-decl}\rangle^{*}\ \langle\text{method-decl}\rangle^{*}\ \text{`\}'} \\
\langle\text{field-decl}\rangle & \rightarrow & \langle\text{type}\rangle \left\{ \textbf{id} \ \mid \ \left\{ \textbf{id}\ \text{`}\texttt{[}\text{'}\ \textbf{intConstant}\ \text{`}\texttt{]}\text{'} \right\} \right\}^{+}\!\!,\ \text{`}\texttt{;}\text{'} \\
 & \mid & \langle\text{type}\rangle\ \textbf{id}\ \text{`}\texttt{=}\text{'}\ \langle\text{constant}\rangle\ \text{`}\texttt{;}\text{'} \\
\langle\ell\text{-value}\rangle & \rightarrow & \textbf{id} \\
 & \mid & \textbf{id}\ \text{`}\texttt{[}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{]}\text{'} \\
\langle\text{expr}\rangle & \rightarrow & \textbf{id} \\
 & \mid & \textbf{id}\ \text{`}\texttt{[}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{]}\text{'} \\
 & \mid & \langle\text{method-call}\rangle \\
 & \mid & \langle\text{constant}\rangle \\
 & \mid & \langle\text{expr}\rangle\ \langle\text{bin-op}\rangle\ \langle\text{expr}\rangle \\
 & \mid & \text{`}\texttt{-}\text{'}\ \langle\text{expr}\rangle \\
 & \mid & \text{`}\texttt{!}\text{'}\ \langle\text{expr}\rangle \\
 & \mid & \text{`}\texttt{(}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{)}\text{'}
\end{array}
$$

(3) **Control-flow and loops**

The following fragment of **Decaf** syntax should be added to the grammar in Q. 2. It adds control flow (**if** statements) and loops (**while** and **for** statements) to **Decaf**.

$$
\begin{array}{rcl}
\langle\text{statement}\rangle & \rightarrow & \langle\text{assign}\rangle\ \text{`}\texttt{;}\text{'} \\
 & \mid & \langle\text{method-call}\rangle\ \text{`}\texttt{;}\text{'} \\
 & \mid & \textbf{if}\ \text{`}\texttt{(}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{)}\text{'}\ \langle\text{block}\rangle \left[\textbf{else}\ \ \langle\text{block}\rangle \right] \\
 & \mid & \textbf{while}\ \text{`}\texttt{(}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{)}\text{'}\ \langle\text{block}\rangle \\
 & \mid & \textbf{for}\ \text{`}\texttt{(}\text{'} \left\{ \langle\text{assign}\rangle \right\}^{+}\!\!,\ \text{`}\texttt{;}\text{'}\ \langle\text{expr}\rangle\ \text{`}\texttt{;}\text{'} \left\{ \langle\text{assign}\rangle \right\}^{+}\!\!,\ \text{`}\texttt{)}\text{'}\ \langle\text{block}\rangle \\
 & \mid & \textbf{return} \left[\text{`}\texttt{(}\text{'}\ \left[ \langle\text{expr}\rangle \right]\ \text{`}\texttt{)}\text{'} \right] \text{`}\texttt{;}\text{'} \\
 & \mid & \textbf{break}\ \text{`}\texttt{;}\text{'} \\
 & \mid & \textbf{continue}\ \text{`}\texttt{;}\text{'} \\
 & \mid & \langle\text{block}\rangle
\end{array}
$$

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

(4) **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.

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.

(5) **Code Optimization using LLVM**

Implement at least the following optimization passes:

1. Convert stack allocation usage (alloca) into register usage (mem2reg)
2. Simple "peephole" optimization (instruction combining pass)
3. Re-associate expresssions
4. Eliminate common sub-expressions (GVN)
5. Simplify the control flow graph (CFG simplification)

You can either modify the source code in your yacc program using the `FunctionPassManager` or use the command-line `opt` utility provided by LLVM.

You can even write your own LLVM pass using the documentation in `http://llvm.org/docs/WritingAnLLVMPass.html`. This can be used to add new options to the `opt` command line utility.

(6) **† The `Decaf` compiler**

Combine all the `Decaf` fragments you have implemented to create a compiler for `Decaf` programs. Create a single yacc/lex program that accepts any syntactically valid `Decaf` programs and produces LLVM assembly language output. Your program should reject any syntactically invalid `Decaf` program and provide a helpful error message (the quality of the error reporting is up to you – at least report the line and character number where the syntax error is thrown). Your program should also perform the semantic checks defined in Q. 4 and the code optimizations defined in Q. 5 above. You can either add the optimization passes as part of the source code or as post-processing calls to `opt`. Make sure that `make test` will compile, optimize and run any files in the `../testcases` directory.