

Homework #4: CMPT-379

Anoop Sarkar – anoop@cs.sfu.ca

- Only submit answers for questions marked with †.
- Checkout your group svn repository:
`svn co https://punch.cs.sfu.ca/svn/CMPT379-1127-g-your-group-name`
- Copy the files for this homework (and then add and commit the files using svn):
`scp -r fraser.sfu.ca:/home/anoop/cmpt379/hw4 CMPT379-1127-g-your-group-name/.`
- Put your solution programs in the `hw4/answer` directory. Use the `makefile` provided. There are strict filename requirements. Read the file `readme.txt` in the `hw4` directory for details.
- Create a file called `HANDLE` in your `hw4` directory which contains your group handle (no spaces).
- The `hw4/testcases` directory contains useful test cases; you will need to consult `readme.txt` for the mapping between the homework questions and test cases and instructions on how to run the auto check program.
- Reading for this homework includes Chp 5 of the Dragon book. We will be using the LLVM Compiler Infrastructure for code generation and code optimization: <http://llvm.org>.

(1) Global variables

Add support for global variables. The following rules in **Decaf** are the ones that involve the use of global variables.

```

<program>  → <extern-defn> * class id ‘{’ <field-decl> * <method-decl> * ‘}’
<field-decl> → <type> { id | { id ‘[’ intConstant ‘]’ } } + ‘;’
              | <type> id ‘=’ <constant> ‘;’
<ℓ-value>  → id
              | id ‘[’ <expr> ‘]’
<expr>     → id
              | id ‘[’ <expr> ‘]’
```

(2) Control-flow and loops

The following fragment of **Decaf** syntax adds control flow (**if** statements) and loops (**while** and **for** statements) to **Decaf**.

```

<statement> → if ‘(’ <expr> ‘)’ <block> [ else <block> ]
              | while ‘(’ <expr> ‘)’ <block>
              | for ‘(’ { <assign> } + ‘;’ <expr> ‘;’ { <assign> } + ‘;’ <block>
              | return [ ‘(’ [ <expr> ] ‘)’ ] ‘;’
              | break ‘;’
              | continue ‘;’
```

Your program must implement short-circuit evaluation for boolean expressions.

(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.

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) 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 expressions
4. Eliminate common sub-expressions (GVN)
5. Simplify the control flow graph (CFG simplification)

You should modify the source code in your yacc program using the `FunctionPassManager` LLVM API call.

You can even write your own LLVM pass using the documentation in <http://llvm.org/docs/WritingAnLLVMPass.html>.

(5) † The **Decaf** compiler

Create a yacc/lex program that accepts any syntactically valid **Decaf** program as defined in the **Decaf** specification 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 the **Decaf** specification and the code optimizations defined in Q. 4 above. Make sure that `make` will compile your program.