Homework #3: 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/hw3 CMPT379-1127-g-your-group-name/.`
- Put your solution programs in the `hw3/answer` directory. Use the `makefile` provided. There are strict filename requirements. Read the file `readme.txt` in the `hw3` directory for details.
- Create a file called `HANDLE` in your `hw3` directory which contains your group handle (no spaces).
- The `hw3/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) Implement a symbol table. A symbol table is a mapping from identifiers to any information that the compiler needs for code generation. A symbol table is easily implemented using hash tables or maps, e.g. here is a declaration of a symbol table using STL in C++:

```
typedef map<string, descriptor* > symbol_table;
```

where a *descriptor* is a structure or class which contains a *type*, a *register destination*, a *memory address* location, and a variable *spilled* indicating if the value is to be found in a register or in memory (note that we will not use memory locations for variables until later).

In **Decaf** we are allowed to *shadow* a variable declaration (see Q. (2) for an example). This means that a new definition for an identifier in a block will cause a new descriptor to be associated with the identifier, but once the block terminates the previous descriptor for the identifier has to be restored. A simple way to implement this notion of local scoping is to specify that each block can create a new symbol table in a list:

```
typedef list<symbol_table > symbol_table_list;
symbol_table_list symtbl;
```

If a variable has a local definition that shadows another definition of the same variable name, we pick up the most recently defined descriptor for that variable by simply scanning the list of symbol tables starting from the most recent one:

```
descriptor* access_symtbl(string ident) {
  for (symbol_table_list::iterator i = symtbl.begin(); i != symtbl.end(); ++i) {
    symbol_table::iterator find_ident;
    if ((find_ident = i->find(ident)) != i->end())
      return find_ident->second;
  }
  return NULL;
}
```

This is just one way to implement a symbol table. You can implement it any way you like, as long as it can handle shadowing of variables.

For this question, you can assume that the identifiers are variables, but in later homeworks, the symbol table will also store information about function names, global variables, etc., and additional information will have to be added to the descriptor.

(2) † Use a symbol table implementation to remember the line number of each variable definition. A variable definition includes the following cases: field or global variables, variables defined in function parameters, and local variables defined in a block. In **Decaf** you can start a new block by just opening curly braces and close a block using closing curly braces.

Your implementation should enter information about each variable definition into a symbol table. For each instance when an identifier is used in a statement, your yacc program should introduce a comment line into the **Decaf** program that specifies the line number of the variable definition for that identifier.

Print out the input source in the lex program. You only need to add actions in your **Decaf** yacc grammar for the variable definition rules (to add information to the symbol table) and the variable usage (to add print statements to print out a comment about where the variable was defined). When entering a new block you should make sure to insert a new symbol table so that local re-definitions of a variable will shadow higher level definitions. A class definition, a method definition or simply a block between { and } defines a new nested symbol table.

For example, for the input on the left column, your program should produce the output in the right column. The line numbers refer to lines in the original source code.

```
class C {                          class C {
    int main() {                       int main() {
        int x, y;                          int x, y;
        {                                  {
            int p, q;                          int p, q;
            {                                  {
                int y;                             int y;
                x = 1;                             x = 1; // using decl on line: 3
                y = 1;                             y = 1; // using decl on line: 7
                {                                  {
                    int x;                             int x;
                    p = 1;                             p = 1; // using decl on line: 5
                    x = 1;                             x = 1; // using decl on line: 11
                    y = 1;                             y = 1; // using decl on line: 7
                }                                  }
            }                                  }
        }                                  }
    }                                  }
}                                  }
```

Type in `nl decaf-file` to print out the line numbers to check if your output refers to the correct line numbers.

(3) **Decaf standard library**

The file `decaf-stdlib.c` contains the standard library for **Decaf** containing the implementation of functions like `print_int`, `read_int`, etc.

Write a C or C++ program that uses the functions defined in `decaf-stdlib.c`.
For example,

```
int i = read_int();
print_string("this is a test:");
print_int(i);
print_string("\n");
```

We will link the **Decaf** standard library with the x86 assembly that will be generated using LLVM.

(4) **LLVM Assembly**

LLVM is both a library for code generation and also a definition of an abstract assembly language. LLVM assembly is converted into x86 machine code. The file `helloworld.ll` contains a simple Hello, World program in LLVM assembly.

```
; Declare the string constant as a global constant.
; run the following command to run this LLVM assembly program:
; sh run-llvm-code.sh helloworld.ll
@LC0 = internal constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8*)

; Definition of main function
define i32 @main() {
  ; Convert [13 x i8]* to i8*
  %cast = getelementptr [13 x i8]* @LC0, i8 0, i8 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast)
  ret i32 0
}
```

`i8` is an 8-bit integer used by LLVM for ASCII characters. The `puts` function takes an ASCII string and returns an integer return value of type `i32`. Except for the `getelementptr` instruction the rest is easy to follow. The next question explains the use of the `getelementptr` to access global constants. The LLVM assembly file can be converted into executable machine code using the following steps.

```
$ make helloworld
using llvm to compile file: helloworld.ll
llvm-as helloworld.ll
llc helloworld.bc
gcc helloworld.s decaf-stdlib.c -o ./helloworld
```

In this case we did not need to link with the **Decaf** standard library since we do not use any of the function in it, but when we implement the **Decaf** compiler it will be easier to use the standard library functions instead of a function like `puts` which take pointers as arguments.

(5) **LLVM Assembly with `Decaf` library functions**

The following LLVM assembly code defines a function `@add1` that adds two integers and prints out the value followed by a newline.

```
declare void @print_int(i32)
declare void @print_string(i8*)
declare i32 @read_int()

; store the newline as a string constant
; more specifically as a constant array containing i8 integers
@.nl = constant [2 x i8] c"\0A\00"

define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @main() {
entry:
  %tmp5 = call i32 @add1(i32 3, i32 4)
  call void @print_int(i32 %tmp5)
  ; convert the constant newline array into a pointer to i8 values
  ; using getelementptr, arg1 = @.nl,
  ; arg2 = first element stored in @.nl which is of type [2 x i8]
  ; arg3 = the first element of the constant array
  ; getelementptr will return the pointer to the first element
  %cast.nl = getelementptr [2 x i8]* @.nl, i8 0, i8 0
  call void @print_string(i8* %cast.nl)
  ret i32 0
}
```

Write down a recursive version of the addition function in LLVM assembly. The following Python program illustrates the algorithm.

```
def rec_add(a, b):
    if a == 0:
        return b
    else:
        return rec_add(a-1, b+1)
```

The following template illustrates the use of a conditional expression for branching and the use of a recursive function call. It checks whether the first argument to the `add2` function is equal to zero and sets a boolean location `%tmp1` to a boolean value (booleans in LLVM are of type `i1` or integer of bit width 1). The `br` call then branches either to `%done` or `%recurse` based on the value in the boolean condition variable `%tmp1`. Extend this template to write the LLVM assembly for recursive addition function.

```
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
recurse:
  ; insert LLVM assembly here
done:
  ; insert LLVM assembly here
}
```

(6) Implement the factorial function in LLVM assembly and print out the value of 11! using `print_int`.

(7) LLVM Code Generation API

The following yacc program does code generation using the LLVM API. The full source code is available in the file `sexpr-codegen.y`. It takes simple expressions like `2+3-4` and produces LLVM assembly as output.

```
%union{
  class ExprAST *ast;
  int number;
}

%token <number> NUMBER
%type <ast> expression
%%
statement: expression
           {
             Value *RetVal = $1->Codegen();
             Function *print_int = gen_print_int_def();
             Function *TheFunction = gen_main_def(RetVal, print_int);
             verifyFunction(*TheFunction);
           }
           ;
expression: expression '+' NUMBER
            {
              $$ = new BinaryExprAST('+', $1, new NumberExprAST($3));
            }
          | expression '-' NUMBER
            {
              $$ = new BinaryExprAST('-', $1, new NumberExprAST($3));
            }
          | NUMBER
            {
              $$ = new NumberExprAST($1);
            }
          ;
%%
```

For the input `2+3-4` it produces the output LLVM assembly:

```
declare i32 @print_int(i32)
define i32 @main() {
entry:
  %calltmp = call i32 @print_int(i32 1)
  ret i32 0
}
```

It does this by extending your code for building an abstract syntax tree (AST). For example, binary expressions are represented as the following AST data structure:

```
/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};
```

The code is then generated from the AST by calling functions defined in the LLVM API. Two main data structures contain the LLVM assembly code:

```
static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());

int main() {
  // initialize LLVM
  LLVMContext &Context = getGlobalContext();
  // Make the module, which holds all the code.
  TheModule = new Module("module for very simple expressions", Context);
  // parse the input and create the abstract syntax tree
  yyparse();
  // Print out all of the generated code to stderr
  TheModule->dump();
  exit(0);
}
```

The generated code is produced as a pointer to a data structure called `Value`. For example the following function is used to generate code for binary expressions.

```
Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateAdd(L, R, "addtmp");
  case '-': return Builder.CreateSub(L, R, "subtmp");
  }
}
```

Extend the code provided to you in `sexpr-codegen.y` in order to handle LLVM code generation for the following grammar:

```
statement_list: statement ';' statement_list
              | /* empty */
              ;
statement: NAME '=' expression
         | NAME '(' expression ')'
         ;
expression: expression '+' NUMBER
          | expression '-' NUMBER
          | expression '+' NAME
          | expression '-' NAME
          | NUMBER
          | NAME
          ;
```

It should accept input like the following:

```
a=2+3;
b=5-2;
c=a+b;
print_int(c+2);
```

And produce LLVM assembly:

```
define i32 @main() {
entry:
  %a = alloca i32
  store i32 5, i32* %a
  %b = alloca i32
  store i32 3, i32* %b
  %a1 = load i32* %a
  %b2 = load i32* %b
  %addtmp = add i32 %a1, %b2
  %c = alloca i32
  store i32 %addtmp, i32* %c
  %c3 = load i32* %c
  %addtmp4 = add i32 %c3, 2
  %calltmp = call i32 @print_int(i32 %addtmp4)
  ret i32 0
}
```

You will need to use your symbol table implementation to store the location of the variables. Also, use the LLVM `alloca` instruction to create storage on the stack for the variables in our simple programming language.

The LLVM APIs are explained on the LLVM website, but a easier way to learn the API is to read the source code tutorial called `kscope.cc` that implements a simple programming language called Kaleidoscope and re-using code snippets from that code in your own implementation. A simpler version of `kscope.cc` is provided as `kscope1.cc` which is smaller and easier to read as a result.

(8) † **Code generation for expressions in `Decaf`**

Implement code generation for the following fragment of **`Decaf`**:

$$
\begin{array}{rcl}
\langle\text{program}\rangle & \rightarrow & \langle\text{extern-defn}\rangle^{*}\ \textbf{class id}\ \text{‘\{’}\ \langle\text{method-decl}\rangle^{*}\ \text{‘\}’} \\
\langle\text{extern-defn}\rangle & \rightarrow & \textbf{extern}\ \langle\text{method-type}\rangle\ \textbf{id}\ \text{‘(’}\ \left[\left\{\ \langle\text{extern-type}\rangle\ \right\}^{+}\!,\ \right]\ \text{‘)’}\ \text{‘;’} \\
\langle\text{extern-type}\rangle & \rightarrow & \textbf{string}\ |\ \langle\text{type}\rangle \\
\langle\text{method-decl}\rangle & \rightarrow & \langle\text{method-type}\rangle\ \textbf{id}\ \text{‘(’}\ \left[\left\{\ \langle\text{type}\rangle\ \textbf{id}\ \right\}^{+}\!,\ \right]\ \text{‘)’}\ \langle\text{block}\rangle \\
\langle\text{block}\rangle & \rightarrow & \text{‘\{’}\ \langle\text{var-decl}\rangle^{*}\ \langle\text{statement}\rangle^{*}\ \text{‘\}’} \\
\langle\text{var-decl}\rangle & \rightarrow & \langle\text{type}\rangle\ \left\{\ \textbf{id}\ \right\}^{+}\!,\ \text{‘;’} \\
\langle\text{method-type}\rangle & \rightarrow & \textbf{void}\ |\ \langle\text{type}\rangle \\
\langle\text{type}\rangle & \rightarrow & \textbf{int}\ |\ \textbf{bool} \\
\langle\text{statement}\rangle & \rightarrow & \langle\text{assign}\rangle\ \text{‘;’} \\
 & | & \langle\text{method-call}\rangle\ \text{‘;’} \\
 & | & \langle\text{block}\rangle \\
 & | & \textbf{return}\ \left[\text{‘(’}\ \left[\langle\text{expr}\rangle\right]\ \text{‘)’}\right]\ \text{‘;’} \\
\langle\text{assign}\rangle & \rightarrow & \langle\ell\text{-value}\rangle\ \text{‘=’}\ \langle\text{expr}\rangle \\
\langle\text{method-call}\rangle & \rightarrow & \textbf{id}\ \text{‘(’}\ \left[\left\{\ \langle\text{method-arg}\rangle\ \right\}^{+}\!,\ \right]\ \text{‘)’} \\
\langle\text{method-arg}\rangle & \rightarrow & \langle\text{expr}\rangle\ |\ \textbf{stringConstant} \\
\langle\ell\text{-value}\rangle & \rightarrow & \textbf{id} \\
\langle\text{expr}\rangle & \rightarrow & \textbf{id} \\
 & | & \langle\text{method-call}\rangle \\
 & | & \langle\text{constant}\rangle
\end{array}
$$

	|	⟨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**

It includes the following main components for code generation:

- Arithmetic and Boolean expressions (**Warning**: do not attempt short-circuit of boolean expressions).
- Function calls.
- Function definitions (including recursive functions).
- Declaration of extern functions (defined in `decaf-stdlib.c`).

Most of these are trivial to implement using the LLVM API. Only the first one, arithmetic and boolean expressions is non-trivial. The **`Decaf`** language specification has a section on Semantics which lays out the rules of type coercion and other gray areas in the implementation of **`Decaf`**.

Note that boolean constants are of type `i1` in LLVM assembly. Char constants can be either `i8` or `i32`. You will need to refer to `http://llvm.org/releases/3.1/docs/LangRef.html` and `http://llvm.org/releases/3.1/docs/tutorial/`.

(9) For the following context-free grammar:

block	→	‘{’ var-decl-list ‘}’
var-decl-list	→	var-decl var-decl-list | ϵ
var-decl	→	type id-comma-list **‘;’**
id-comma-list	→	**id** **‘,’** id-comma-list | **id**
type	→	**int** | **bool**

Provide a yacc program that passes the type information for each variable by using *inherited attributes*. The program should enter each variable name into a symbol table along with its type information.