Homework #3: CMPT-379

Distributed on Oct 25; due on Nov 12

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) † **Introduction to MIPS assembly**: The target of the code generation step for the compiler will be MIPS R2000 assembly language. MIPS is a reduced instruction set (RISC) machine. We will treat MIPS assembly code as a *virtual machine* and use a simulator for MIPS assembly called spim that takes MIPS assembly and simulates (runs) it on x86 Linux. spim is available in the location mentioned on the course web page.

Your task for this homework is to convert the following **Decaf** program called catalan.decaf *by hand* into MIPS assembly.

```
class Catalan {

    void main() {
       int i, j;
       i = callout("read_int");
       j = cat(i);
       callout("print_int", j);
       callout("print_str", "\n");
    }

    // catalan number of n
    int cat(int n) {
       int t;
       t = fact(n);
       return( fact(2*n) / (t * t * (n+1)) );
    }

    // factorial of n
    int fact(int n) {
       if (n == 1) { return(1); }
       else { return(n*fact(n-1)); }
    }
}
```

Provide the MIPS assembly program in a file called catalan.mips which should run on the simulator spim as follows: spim -file catalan.mips

When the MIPS program is run on the spim simulator, it should wait for an integer input *n* from the user, and then print out the result of the catalan function for *n* followed by a newline character. *The MIPS program must be a direct translation of the* **Decaf** *program.* Comment your MIPS code heavily. Compare your generated code to the parse tree and reflect on automating code generation (topic of a future homework). This exercise will familiarize you with MIPS assembly. Read the documentation provided on the course web page that introduces you to MIPS assembly, including a detailed tutorial on passing parameters on the stack frame for procedure calls in MIPS, and a tutorial on how to use spim. It assumes some familiarity with assembly language. Ask for background reading if you are not familiar with any of the terms used in the MIPS documentation.

You have to manage the register names used in the output assembly code. For this homework, you can ignore some of the complexities of code generation by assuming that we have a sufficient number of temporary registers at hand. A few facts that might be useful: in MIPS assembly, upto four arguments can be passed directly to a subroutine in the registers $a0-$a3, and $s0-$s7 are temporary registers that retain values during a function call, while temporary registers `$t0-$t7` do not retain their values. 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		

Below is an example in MIPS that uses the `syscall` interface above to read an integer from standard input using `read_int`, and then prints it out to standard output using `print_int`, and then prints out a newline using `print_string`:

```
        .data
nl:
        .asciiz "\n"
        .text
main:
        li $v0, 5
        syscall
        move $a0, $v0
        li $v0, 1
        syscall
        li $v0, 4
        la $a0, nl
        syscall
```

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. Below is a simple **Decaf** program that computes a simple expression and the corresponding MIPS translation (it shows how to use temporary registers and how to store and use a global string constant).

```
class Expr {
  void main() {
    int x;
    x = 2*3+5;
    callout("print_int", x);
    callout("print_string", "\n");
  }
}
```

```
        .data
str0:
        .asciiz "\n"
#-------------------------
        .text
        .globl main
main:
        li $t0, 2
        li $t1, 3
        mul $t2, $t0, $t1
        li $t0, 5
        addu $t1, $t0, $t2
        move $a0, $t1
        li $v0, 1
        syscall
        li $v0, 4
        la $a0, str0
        syscall
```

(2) 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. (4) 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.

(3) For the following context-free grammar:

$$
\begin{array}{lcl}
\text{block} & \rightarrow & \text{`\{'}\ \text{var-decl-list}\ \text{`\}'} \\
\text{var-decl-list} & \rightarrow & \text{var-decl var-decl-list} \mid \epsilon \\
\text{var-decl} & \rightarrow & \text{type id-comma-list}\ \text{`}\mathtt{;}\text{'} \\
\text{id-comma-list} & \rightarrow & \textbf{id}\ \text{`}\mathtt{,}\text{'}\ \text{id-comma-list} \mid \textbf{id} \\
\text{type} & \rightarrow & \textbf{int} \mid \textbf{bool}
\end{array}
$$

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.

(4) † Implement the following modified fragment of **Decaf** syntax. The syntax has been changed a little to allow statements like `x;` which are not allowed in **Decaf**.

$$
\begin{array}{rcl}
\langle\text{block}\rangle & \rightarrow & \text{`\{'}\ \langle\text{var-decl}\rangle^{*}\ \langle\text{statement}\rangle^{*}\ \text{`\}'} \\
\langle\text{var-decl}\rangle & \rightarrow & \langle\text{type}\rangle \left\{ \textbf{id} \right\}^{+},\ \text{`}\mathtt{;}\text{'} \\
\langle\text{type}\rangle & \rightarrow & \textbf{int} \mid \textbf{bool} \\
\langle\text{statement}\rangle & \rightarrow & \textbf{id}\ \text{`}\mathtt{;}\text{'} \mid \langle\text{block}\rangle
\end{array}
$$

The 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.

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.

```
{ int x, y;                        { int x, y;
  { int p, q;                        { int p, q;
    { int y;                           { int y;
      x; y;                              x; // using decl on line: 1
      { int x;                      y; // using decl on line: 3
          p; x; y;
      } } } }                            { int x;
                                             p; // using decl on line: 2
                                    x; // using decl on line: 5
                                    y; // using decl on line: 3

                                         } } } }
```

(5) † **Code generation for expressions**

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:

⟨block⟩	→	‘{’ ⟨var-decl⟩ * ⟨statement⟩ * ‘}’
⟨var-decl⟩	→	⟨type⟩ $\{$ **id** $\}^{+}_{,}$ ‘**;**’
⟨type⟩	→	**int** | **bool**
⟨statement⟩	→	⟨assign⟩ ‘**;**’ | ⟨method-call⟩ ‘**;**’ | ⟨block⟩
⟨assign⟩	→	⟨lvalue⟩ ‘**=**’ ⟨expr⟩
⟨method-call⟩	→	**callout** ‘**(**’ **stringConstant** $\Big[\Big\{$ ‘**,**’ $\Big\{$ ⟨callout-arg⟩ $\Big\}^{+}_{,}$ $\Big\}\Big]$ ‘**)**’
⟨callout-arg⟩	→	⟨expr⟩ | **stringConstant**
⟨lvalue⟩	→	**id**
⟨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**

This fragment of **Decaf** represents the expression-level sub-grammar. Note that you should augment your yacc implementation of this sub-grammar from your previous homework and extend it to handle code generation.

For example, the input program (from the sub-grammar) in the left column should produce the MIPS code in the right column. As you can see, each component of the expression parse tree produces a value that is stored in a MIPS register, making sure that you use as many registers as necessary. Other examples of input and output are provided in the homework directory.

```
{ callout("print_int", 2+2*3+5); }
```

```
.text
.globl main
main:
 li $t0, 2
 li $t1, 2
 li $t2, 3
 mul $t1, $t1, $t2
 addu $t0, $t0, $t1
 li $t1, 5
 addu $t0, $t0, $t1
 li $v0, 1
 move $a0, $t0
 syscall
```

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.

Read Section 8.6 from the Dragon book for an algorithm that specifies how to allocate register names. You must reuse registers as far as possible. By exploiting the parse tree, only a few registers are typically used even for very complex expressions. For this question, once your program is reusing registers as much as possible, you can ignore some of the complexities of code generation by assuming that you 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`. If your program, while using this simple code generation algorithm, runs out of registers to use, your program can exit with an error message.

You should print a boolean as an integer: `0`=false and `1`=true.

You will have to implement a *symbol table* which will store information about the variables in the **`Decaf`** program.

Submit a program 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>
```

(6) After finishing Q. (5), change your program so that it can deal with the actual set of registers available in MIPS assembly. Use the heap to store values associated with identifiers. Store the value of an identifier to a memory location on the heap. When the value is needed it should be loaded into a register. The symbol table should store the memory location of the identifier. This will allow the re-use of registers (using register *spilling* to the heap) when the registers run out during code generation. See the MIPS file `using-heap.mips` in the homework directory for an example of how to spill a register to the heap and then load the value back into a register.