Homework #6: CMPT-379

Distributed on Wed, Mar 15; Due on Mon, Mar 29

Anoop Sarkar – anoop@cs.sfu.ca

(1) **(Intermediate) Code Generation**:

This homework takes the LR parser you have built so far for **Decaf** and augments the parser with the backend step, implementing the code generation portion of the compiler.

The target of the code generation step will be MIPS R2000 assembly language, for a reduced instruction set (RISC) machine. 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.

You will have to read the documentation provided on the course web page that introduces you to MIPS assembly and on how to use `spim`. It assumes some familiarity with assembly language. Please ask for background reading if you are not familiar with any of the terms used in the MIPS documentation.

Chapter 8 in the Dragon Book provides a case-by-case treatment of code generation issues for each kind of statement in **Decaf**. In addition, you will need to pay attention to the MIPS documentation mentioned above which includes a detailed tutorial on passing parameters on the stack frame for procedure calls in MIPS. You can assume that function calls will have no more than four arguments. In MIPS assembly, upto four arguments can be passed directly to a subroutine in the registers `$a0-$a3`.

You have a couple of options in how to implement code-generation on top of your LR parser:

- Implement synthesized and/or inherited attribute passing on top of your LR parser. A reduce passes MIPS assembly code fragments on the LR parser stack for synthesized attributes. For inherited L-attributes, goto actions also pass information on the stack. Store source code that can access the stack and is executed at runtime for each CFG rule for each reduce/goto action.
- Implement code generation by reading in the parse tree and producing code via tree rewriting using the techniques and algorithms given in Chapter 9 of the Dragon book. In fact, you can use your LR parser implementation to implement code generation via tree rewriting (this algorithm is described on page 578 of the Dragon book).

In addition to stack/tree manipulation, you have to manage the register names used in the output assembly code. We will ignore some of the complexities of code generation by assuming that we 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`. Your program should use the algorithm for stacked temporary registers explained in Section 8.3 (page 480) of the Dragon book. However, if despite using this algorithm if your code generation step runs out of registers to use, your program can exit with an error message.

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		

Here 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. When testing your code generation from **`Decaf`** source code, you should start off with simple **`Decaf`** programs like the following:

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

The following MIPS assembly is a valid translation of the above **`Decaf`** code:

```
        .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
```

Note the use of the `.data` section for storing global data. Save the above MIPS assembly program to file `Expr.mips` and run the simulator `spim` as follows:

```
spim -file Expr.mips
```

Once you have implemented procedure calls, if/while/for-statements, array lvalues & rvalues and all arithmetic and boolean operators then you can test with **`Decaf`** programs like `catalan.decaf` and `gcd.decaf` into MIPS and run it using `spim`. Submit the entire compiler pipeline which accepts **`Decaf`** code and produces MIPS assembly that can then be run using `spim` as shown above.