Homework #5: CMPT-379

Distributed on Mon, Oct 23; Due on Fri, Nov 3

Anoop Sarkar – anoop@cs.sfu.ca

(1) **LR Parser Table Construction**:

Your task for this question is to build the action/goto table needed for LR parsing. We will be using LR(0) items to build the table. A program called `makeLRsets` has been provided to you which computes the LR(0) configuration sets for the augmented context-free grammar and the finite-state machine (FSM) that is the result of the *Set-of-Items* construction. Use this FSM to build the action/goto table, which you can then use with the LR parser you built in your previous homework. `makeLRsets` can be run as follows:

```
./makeLRsets -f grammar.txt -o lrfsm.txt > lritems.txt
```

This invocation takes a grammar file called `grammar.txt` in the usual format:

```
t f
t t TIMES f
f ID
f LPAREN t RPAREN
```

In the above invocation, `makeLRsets` writes to `lrfsm.txt` the automaton based on the *Set-of-Items* construction in the following format:

```
0 1 shift LPAREN
0 2 shift ID
0 3 goto f
0 4 goto t
1 1 shift LPAREN
1 2 shift ID
1 3 goto f
1 5 goto t
4 6 shift TIMES
5 6 shift TIMES
5 7 shift RPAREN
6 1 shift LPAREN
6 2 shift ID
6 8 goto f
% 2 reduce 3
% 3 reduce 1
% 4 accept 0
% 7 reduce 4
% 8 reduce 2
```

The format for shift/goto actions is: ⟨`fromstate`⟩ ⟨`tostate`⟩ ⟨`action`⟩ ⟨`symbol`⟩. and for reduce/accept actions: `%` ⟨`state`⟩ ⟨`action`⟩ ⟨`rule-number`⟩, where rule number `0` is assigned to the new rule `top` → `t` added by the construction to create an augmented CFG.

The standard output for `makeLRsets`, which is saved to the file `lritems.txt` in the above invocation, contains all the LR(0) items, the configuration sets for each of the states in the automaton, and the augmented CFG with the rule numbers used to index the reduce actions.

The notation for an item (or dotted rule) is:

`number A` α `\.` β

where `number` is the CFG rule number and `\.` represents the dot.

Your task it to build an action/goto table for your **`Decaf`** context-free grammar using the `makeLRsets` program. It is your job to resolve any conflicts in the action/goto table using the following alternative methods.

Note that for each type of conflict one of these methods will be the most appropriate, e.g. favor shift in shift/reduce conflicts for dangling *else* types of ambiguity, and implement precedence and associativity for arithmetic expressions, etc.

1. Rewrite the **`Decaf`** CFG that you have written for an earlier homework and remove all conflicts.
2. Implement and use the FOLLOW set for the CFG non-terminals to remove shift/reduce and reduce/reduce conflicts using the SLR(1) construction.
3. Exploit the precedence and associativity definition in the **`Decaf`** language definition to handle ambiguity in the use of binary operators.
4. If none of the above resolve the conflicts in the action/goto table then:
   - Resolve shift/reduce conflicts in favor of shifts, and
   - Resolve reduce/reduce conflicts by picking the production rule that was listed first in the grammar file

Submit the CFG for **`Decaf`** that you used for this assignment and the action/goto table produced and any precedence/associativity information you have used as part of the parser definition.

(2) **Error Reporting**

You should include some minimal error reporting in your LR parser, so that syntax errors report the line number and character position in the program file (you can stop at the first error). The particular syntax errors you report is up to you. Optionally, you can exploit the current state of the LR parser to provide more detailed error reporting.

You will need to submit the entire pipeline: from passing the input **`Decaf`** program through the lexical analysis phase, and based on the LR parser, you should produce the parse tree for valid **`Decaf`** programs. Your LR parser should have at least minimal error reporting implemented. Provide a readme file with any special instructions.

(3) **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. Use the idea of using stacked temporary registers explained in Section 8.3 (page 480) of the Dragon book. 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
```