Homework #5: CMPT-379

Distributed on Wed, Mar 3; Due on Mon, Mar 15

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 methods (listed here in order of which option you should exhaust before trying others):

1. Rewrite the **Decaf** CFG that you have written before to remove 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) **Semantic Checking**

Use the parse trees produced by your parser to perform the following semantic checks in an input **Decaf** program:

a. A function 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 function matches the return type in the function definition. e.g. `bool foo() { return(10); }` is an example of a mismatch.

To do these semantic checks you will need to traverse the parse tree. In addition, you will need to implement a symbol table which stores each identifier, the type of the identifier or return type, and the node in the parse tree which indicates the scoping for the identifier.
Raise a semantic error if the input **Decaf** program does not pass any of the above semantic checks.
Submit a program that takes parse trees for **Decaf** as input and performs all the semantic checks listed above. You can 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.