HW 4 – Bottom-up parsing
CS 421 – Spring 2013
Revision 1.0

Assigned Feb. 7, 2013
Due Feb. 12, 2013, 9:30 AM
Extension 48 hours (20% penalty)
Total points 40

1 Change Log

1.0 Initial Release.
1.1 Corrected problem 4b; shift-reduce parses have 10 steps.

2 Overview

This homework will give you practice with the concepts we covered in lectures 7 and 8. You will be asked, among other things, to write some programs in OCaml. You do not need to run these (although it would be a good way to check your answers).

3 Handing in the assignment

You must submit this homework electronically, as a pdf. You may type it in TeX or Word or simply in text and print it to a pdf, or you can write it (legibly!) and scan the result. But it must be in the form of a pdf.

The handin process is simply this: put your document on an ews machine, name it hw4.pdf and run /class/cs421/handin -s hw4 (or /class/cs421/handin -s -f hw4 for late submissions).

A few problems below call for you to show parse trees. You can do this in several ways: Use a drawing program (as in Word); draw it by hand, scan it in, and include it in your document; or write it out using indentation to indicate the childhood relationship. As an example of the latter, a tree whose root is A, with two children B and C, with B having children D and E, would be written

A
  B
    D
  C
    E
4 Problems

1. (10 pts) This is grammar $G_E$ from lecture 7:

\[
G_E: \quad E \rightarrow E - T | T \\
T \rightarrow id | T * id
\]

and this is an OCaml data type representing *concrete* syntax trees in this grammar:

```ocaml
type g_e_tree = E1 of g_e_tree * token * g_e_tree | E2 of g_e_tree 
| T1 of token | T2 of g_e_tree * token * token
and token = Ident of string | Star | Minus ;;
```

For example, the parse tree for “x - y * z” would be represented by the term:

\[
E1 (E2(T1(Ident "x")), Minus, T2((T1(Ident "y")), Star, Ident "z"))
\]

Here is an abstract syntax for arithmetic expressions with identifiers and times and minus:

```ocaml
type expr = Id of string | Sub of expr*expr | Times of expr*expr
```

Write an OCaml function to translate concrete syntax trees to abstract syntax trees:

```ocaml
# let rec cst_to_ast t = ... 
val cst_to_ast : g_e_tree -> expr = <fun>
# cst_to_ast (E1 (E2(T1(Ident "x")), Minus, T2((T1(Ident "y")), Star, Ident "z")));;
- : expr = Sub (Id "x", Times (Id "y", Id "z"))
```

Note that some uses of the concrete syntax constructors are impossible; for example, the second argument to $E1$ is *always* the token Minus. In writing `cst_to_ast`, you can rely on this; for example, if you use pattern-matching, the patterns may not be exhaustive.

2. (10 pts) This is grammar $G_A$ from lecture 7:

\[
E \rightarrow id | E - E | E * E
\]

(a) The sentence $a * b * c$ has exactly two parse trees. Show them. Number them clearly as 1 and 2, for reference in later question.

(b) Those two parse trees correspond to distinct shift-reduce parses. Write them, using the format from lecture 8. Here is the first line:
Remember that on each line you should give the action to take on the stack and input on that line; the effect of that action — a different stack and/or input — shows up on the following line. Both these shift-reduce parses have exactly 11 steps. Present them in the same order as the trees in part (a), so we know which parse corresponds to which tree. Point to the first line at which the two parses differ from one another.

(c) Only one of the two parse trees is “correct” in the sense of making multiplication left-associative. (i) Say which one is correct, and (ii) say which of the two parses in part (2) took the correct action when the other took the wrong one.

(d) Give the precedence declarations you would use in ocamlyacc to make sure the correct action is taken for all ambiguous cases for this grammar. (Use names Minus and Star for the two tokens.)

3. (10 pts) In C, there are many operators with different precedences and associativities. The stratified grammar \( G_E \) from problem 1 can be extended to any number of levels. Add operators = and ==. The precedence of the four operators is: * > - > == > =. == is right-associative, while the other three are left-associative.

4. (10 pts) This question concerns a famous example of ambiguity in programming languages: the “dangling else.” The ambiguity arises from this part of the C grammar:

\[
Stmt \rightarrow \text{if ( Expr ) Stmt} \mid \text{if ( Expr ) Stmt else Stmt}
\]

Consider this statement in C:

\[
\text{if (x>0) if (y>0) z=x; else z=y;}
\]

Under what circumstances does the assignment \( z=y \) get executed? The correct answer is: when \( x>0 \) and \( y \leq 0 \). That is, it behaves as if it were:

\[
\text{if (x>0) {if (y>0) z=x; else z=y;}}
\]

But you can only know this if you’ve read the manual. The statement itself is grammatically ambiguous, and could instead be interpreted as if written like this:

\[
\text{if (x>0) {if (y>0) z=x;} else z=y;}
\]
so that \( z = y \) would be executed whenever \( x \leq 0 \).

The rule adopted by all languages is “an else binds with the closest enclosing if.”

Like many ambiguities in programming languages (expressions are an example), this one *could* be resolved by changing the grammar to make it unambiguous, but at the cost of a rather convoluted grammar. An alternative is to keep the original grammar but add precedence rules to make sure the parser “does the right thing” in these circumstances.

To explore this, we use a simple grammar abstracted from this problem by removing the condition (which does not contribute to the ambiguity), and just having one simple statement:

\[
Stmt \rightarrow \text{if } Stmt \mid \text{if } Stmt \text{ else } Stmt \mid \text{null}
\]

(a) The analog of the ambiguous statement above is: if if null else null. Show the two parse trees for this sentence, showing the correct one first.

(b) Show the shift-reduce parses for the two parse trees, showing the correct one first. (Hint: Each parse has 10 steps.)

(c) There is a spot where the two shift-reduce parses differ, where the correct one does a shift while the other does a reduce. Give the ocamlyacc precedence declarations that would cause it to take the correct action in this case.