CS 140 Logic Programming
Fall 2006
Machine Assignment #4
Due Wednesday October 18

Using Definite Clause Grammars (DCG's) to write a mini-calculator (and possibly a mini compiler).

As a prelude to this homework please read carefully David Warren's notes at

http://www.cs.sunysb.edu/~warren/xsbbook/node10.html

The DCG parser in the above URL is right recursive implying that an expression like **3 + 4 + 7 + 2** is interpreted as

(3 + (4 + (7 + 2)))
(This is done to avoid left recursion which results in infinite looping.)

The given grammar rules may be fine for commutative operations like addition and multiplication but is wrong in the case of subtraction and division. For example:

8 – 4 – 2 should be interpreted left recursively as (**(8 – 4) – 2) = 2**
and not **right recursively** as **(8 – (4 – 2)) = 6**

First, make sure that the parser at the end of the above URL is operational. Then add actions to the given DCG's so that the parser not only accepts a list representing an arithmetic expression, but also produces the results of the operations and operands being parsed. As shown in class this is achieved by interspersing appropriate actions within curly brackets, e.g., {Z **is** X+Y} or {Z **is** X*Y}

Again, test your program of DCG's with these actions before proceeding.

Once these additions are operational you will have to develop fancier DCG's that handle correctly the operations +, - , *, / and also the processing arithmetic expressions from left to right with the right priority as explained in class. (see **Hints** at the bottom of this URL)

Your next step is to modify the previous programs so that instead of evaluating an arithmetic expression you will construct the appropriate syntax trees. For example, instead of printing **7** for evaluating the list **[3, *, 2, +, 1],** your parser-action program should produce the term ***add (mult(3,2), 1)***

Once this step is implemented you should extend your program by using a scanner that reads an actual arithmetic expression like **3*2 + 1** or those containing embedded parenthesis like **(3+4*2) * (6 +2)** and so on. Your result will still be the term representing the given arithmetic expression.

For this purpose you may use already developed scanners that are available in the Web (try Google with the words ***scanner Prolog***). The scanner should translate an expression like

(3+4*2)*(6 +2) into

[(,3,+,4,*,2,),*, (,6, +,2,)]

I strongly advise you to test the scanner separately before incorporating into your final machine assignment.

Your final part of this assignment is to write an ***evaluator*** that evaluates numerically the term produced by the parser and produces the final numerical results.

Please comment your programs and follow the rules for turning in machine assignments presented in the course's web page.

After doing this assignment you should be in the position of prototyping a mini-compiler for a mini-language (***not part of assignment #4***). This would imply constructing terms for assignments, conditionals and while loops and modifying the evaluator to handle those kinds of terms.

Good luck!

PS As a curiosity you may want to look at:
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/8_1.html

Hints

DCG's using the correct left association for processing arithmetic expressions

Grammar rules

E → T R
R → + T R | ε

Where E stands for <expression> and R for <rest of expression> or simply <rest>. The actual DCG's recognizing the above grammar are:

expression (X) --> term (X0) , rest (X0, X).
rest (X1, X) --> [+], term (X2) , rest (add (X1,X2), X).
rest (X,X) -- > [].
Term (a) -- > [a].
Term (b) -- > [b].
Term (c) -- > [c].

Input [a, +, b, +, c] generates add(add(a,b), c) , that is, the result of the query:

expression (Result, [a, +, b, +, c], [])

yields *yes* with

Result = add(add(a,b), c)

Notice that the query of expression contains 3 parameters: the first is the desired results; the second and the third amount to the representation of the input viewed as a difference list.