CEng 242 Homework 6

Due 3^{rd} June 2003

Prolog is commonly used in area of language processing. Its symbolic processing capabilities and backtracking feature provide fast means of parser implementations. It is very easy to write a recursive descent parser in Prolog. With help of backtracking, you do not have to convert your grammar into some deterministic form. Any context free grammar without left recursion can be implemented as a Prolog program.
Each non-terminal symbol like S in the grammar is translated into a clause of the form:
`s(`*`Input`*`,`*`Output`*`)`

Which can be read as "*`Input`* list becomes *`Output`* after parsing-reducing the symbol `s`". When *`Output`* is empty list, the parser accepts *`Input`* as `s` successfully.
The rewrite rule $S \rightarrow T, U, V$ similarly converted into:

```
s(Input,Output) :- t(Input,Chain1), u(Chain1,Chain2), v(Chain2,Output).
```

So parsing S turns into parsing T with same input, taking the output of T parse as input to U, take its output as input to V and take the output of V as the output of S. As a result S will parse concatenation of the symbols at right hand side. So the *`Input`* and *`Output`* are chained with intermediate Prolog variables.
Terminal symbols on the other hand matches and reduces the input list directly. So $Zero \rightarrow 0$ can be written as:

```
zero([0|Remaining],Remaining).
```

For the following example grammar:

$$\begin{array}{lcl} Exp & \rightarrow & Digit \mid Digit, '+', Exp \\ Digit & \rightarrow & 0 \mid 1 \end{array}$$

Corresponding Prolog program can be:

```
exp(Input,Output) :- digit(Input,Output).
exp(Input,Output) :- digit(Input,A), plus(A,B), exp(B,Output).
plus([+|R],R).
digit([0|R],R).
digit([1|R],R).

accept(Inp) :- exp(Inp,[]).
```

Goal clause '`accept([0,+,1,+,0])`' will reply `yes` where '`accept([0,+,1,2])`' will reply `no`.
In this homework you will write such a parser for a simplified ML like expression syntax. You are given the following grammar where left recursion is removed for convenience:

$Exp \rightarrow LetExp \mid Value \mid IfExp$
$Exp \rightarrow LetExp, '+', Exp \mid IfExp, '+', Exp \mid Value, '+', Exp$

$LetExp \rightarrow$ let, $DecList$, in, Exp, end

$IfExp \rightarrow$ if, Exp, then, Exp, else, $Exp, ';'$

$DecList \rightarrow Var, ' =', Exp \mid Var, ' =', Exp, ';', DecList$

$Value \rightarrow Variable \mid Number$
$Variable \rightarrow$ Any string with digits, letters, '_' , starting with small letters
$Number \rightarrow$ any Prolog number

Terminal symbols for this grammar are:
`let, if, then, else, in, end, '+' , '=', ';'` ∪ all numbers ∪ all prolog atoms without punctuation.
Your parser will get a list of terminals, parse an Exp, and return a parse structure in the form:

```
let([X = Exp1, Y = Exp2,...],Exp)
if([CondExp],Exp1,Exp2)
Exp1 + Exp2
Number
Variable
```

This structures nest as the parser descends through deeper expression levels. In order to provide this parse structure, you will add another argument in the clauses which will collect partial parts from the RHS of a rewrite rule and combine in the LHS like:

```
exp(+(D1,D2),Input,Output) :- Digit(D1,Input,A),plus(A,B),Digit(D2,B,Output).
```

Sample run:

```
?- parseexp(X,[abc,+,1]).
X = abc+1 ;
no
?- parseexp(X,[if,0,then,1,else,4,+,2,;]).
X = if(0,1,4+2) ;
no
?- parseexp(X,[if,0,then,1,else,4,+,2]).
no
?- parseexp(X,[let,y,=,if,0,then,1,else,4,;,+,2,;,x,=,1,in,w,+,w,end]).
X = let([y=if(0,1,4)+2,x=1],w+w);
no
?- parseexp(X,[let,x,=,let,y,=,let,z,=,2,in,
                                        5,+,4,+,z,
                                    end,in,
                                x,+,y,
                            end,in,
                        x,end]).
X = let([x=let([y=let([z=2],5+(4+z))],x+y)],x) ;
```

no

You can use following clauses to identify numbers an identifiers (*Variable*):

```
number(X)   Built-in, succeeds if X is instantiated to a number

identifier(X) :- name(X,Str),  firstsmallletter(Str).

firstsmallletter([X|R]) :- X>=97 , X=<122 ,restothers(R).
restothers([]).
restothers([X|R]) :- X>=48,X=<57,restothers(R).
restothers([X|R]) :- X>=65,X=<90,restothers(R).
restothers([X|R]) :- X>=97,X=<122,restothers(R).
restothers([95|R]) :- restothers(R).
```