Garrett Hall
CSC 173
Prolog, Week 3-4

Overview

I created a compact scanner, parser, and evaluator that can calculate basic arithmetic expressions. First it prompts the user to type an expression

parser.pl

```
-? test.
Input:
```

Then it outputs the result of each step of the evaluation

```
?- test.
Input: 10 * (3 + 4) - 3 / 2.
Scan: [10, *, (, 3, +, 4, ), -, 3, /, 2]
Parse: e(e(10, *, e(3, +, 4)), -, e(3, /, 2))
Eval: 68.5
```

I'll discuss each of these in order.

Scanner

The scanner can be written in 4 rules. The first simply grabs input and begins the scanning.

```
scan(S) :- get(C), scan_token(S, C).
```

The `scan_token` predicate simply accumulates tokens in the first argument and terminates when it finds a period. `name(T,Word)` converts words (list of integers) into tokens.

```
scan_token([T|Rest], C) :- scan_word(Word, C, Next),
      (period(Next), name(T,Word), Rest = [];
       name(T,Word), scan_token(Rest, Next)).
```

Each of the tokens is built using `scan_word`. The only types it cares about are whitespaces, operators, and numbers. Scanning numbers requires a separate predicate (not shown), corresponding to a new DFA state for accepting digits. `Next` is the lookahead character.

```
scan_word(Word, C, Next) :-
      whitespace(C), get(Next), scan_word(Word, Next, _);
      operator(C), get(Next), Word = [C];
      digit(C), scan_number(Word, C, Next).
```

Aside from juggling lookahead characters and lack of decent character and string support, scanning in Prolog is not too difficult. The logic-based nature of Prolog implicitly creates fail states, making DFA construction somewhat easier.

Parser

Thanks to Prolog's built-in `-->` operator, parsing can be implemented quite elegantly. My parser is only 12 one-line rules:

```
parse(In, Out) :- phrase(expr(Out), In).
expr(e(T,A,E)) --> term(T), add_op(A), expr(E).
expr(T) --> term(T).
term(e(F,M,T)) --> factor(F), mult_op(M), term(T).
term(F) --> factor(F).
factor(E) --> ['('], expr(E), [')'].
factor(Num,[Num|X],X) :- number(Num).
factor(e(0,-,F)) --> [-], factor(F).
add_op(+) --> [+].
add_op(-) --> [-].
mult_op(*) --> [*].
mult_op('/') --> ['/'].
```

This is Scott's grammar verbatim, only the left-recursion has been removed so that

$$expr \rightarrow expr\ \ add\_op\ \ term$$
$$expr \rightarrow term$$

becomes

$$expr \rightarrow term\ \ add\_op\ \ expr$$
$$expr \rightarrow term$$

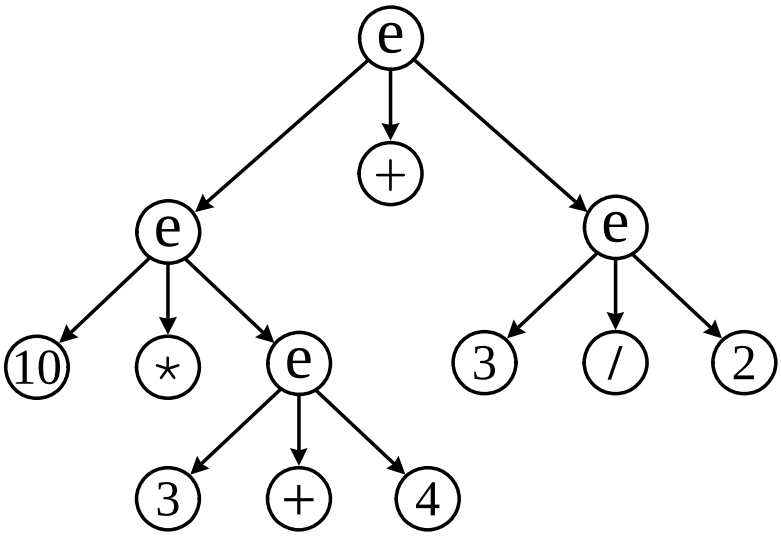
The parse tree being built is quite simple, with only `e()` used to indicate a tree node. So in this parse tree (taken from first example)

```
e(e(10, *, e(3, +, 4)), -, e(3, /, 2))
```

the `(3 + 4)` from the input has become the function `e(3, +, 4)`, indicating a higher evaluation priority. The simplicity pays off when the tree is passed to the evaluator.

Evaluator

The evaluator uses only 3 predicates.



At the very first node the left and right sides are recursively evaluated to get `Xv` and `Yv`, which are themselves evaluated:

```
eval(e(X,Op,Y), V) :-
      eval(X,Xv),
      eval(Y,Yv),
      eval(e(Xv,Op,Yv), V).
```

If the left and right sides are numbers then they can be evaluated immediately:

```
eval(e(X,Op,Y), V) :-
      number(X), number(Y),
       (Op = +, V is X + Y;
        Op = -, V is X - Y;
        Op = *, V is X * Y;
        Op = '/', V is X / Y).
```

Lastly, if asked to evaluate a number, the number is just returned:

```
eval(X, V) :- number(X), X = V.
```