

11. Applications of Logic Programming

Oscar Nierstrasz

SEND
+MORE

MONEY

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

2. Symbolic Interpretation

- Definite Clause Grammars
- Interpretation as Proof
- An interpreter for the calculator language

Reference

- > *The Ciao Prolog System Reference Manual*, Technical Report CLIP 3/97.1, ciao-lang.org

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

2. Symbolic Interpretation

- Definite Clause Grammars
- Interpretation as Proof
- An interpreter for the calculator language

1. Solving a puzzle

 *Find values for the letters so the following equation holds:*

$$\begin{array}{r} \text{SEND} \\ +\text{MORE} \\ \hline \text{MONEY} \end{array}$$

This is a classical Diophantine equation, a polynomial in multiple unknowns, but with integer solutions.

A non-solution:

We would like to write:

```
soln0 :-A is 1000*S + 100*E + 10*N + D,  
        B is 1000*M + 100*O + 10*R + E,  
        C is 10000*M + 1000*O + 100*N + 10*E + Y,  
        C is A+B,  
        showAnswer(A,B,C).  
  
showAnswer(A,B,C)      :- writeln([A, ' + ', B, ' = ', C]).  
writeln([])             :- nl.  
writeln([X|L])          :- write(X), writeln(L).
```

Although this is not as elegant as the original formulation, it is highly declarative, and expresses what we want as a result, without saying how to compute the answer. We trust Prolog's backtracking engine to search (depth-first) for a solution.

A non-solution ...

```
?- soln0.
```

```
⇒ » evaluation_error: [goal(_1007 is 1000 * _1008 +  
100 * _1009 + 10 * _1010 + _1011),  
argument_index(2)]  
[Execution aborted]
```

But this doesn't work because “is” can only evaluate expressions over *instantiated variables*.

```
?- 5 is 1 + x.
```

```
⇒ » evaluation_error: [goal(5 is  
1+_64),argument_index(2)]  
[Execution aborted]
```

The problem is that, at the point where we reach the subgoal

`C is A+B`

neither A nor B is fully instantiated, so `is/2` cannot compute an answer.

Recall that `is/2` is an example of a Prolog predicate that requires its second argument to be a *fully instantiated term*.

A first solution

So let's instantiate them first:

```
digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).
digits([]).
digits([D|L]) :- digit(D), digits(L).
% pick arbitrary digits:
soln1 :-    digits([S,E,N,D,M,O,R,E,M,O,N,E,Y]),
            A is 1000*S + 100*E + 10*N + D,
            B is 1000*M + 100*O + 10*R + E,
            C is 10000*M + 1000*O + 100*N + 10*E + Y,
            C is A+B,          % check if solution is found
            showAnswer(A,B,C).
```

The predicate `digit/1` will instantiate its argument to a digit.
`digits/1` will *instantiate a list of digits*.

Now we are sure that both `A` and `B` are fully instantiated when we try to compute `A+B`.

Note that some variables *appear more than once* in the call to `digits/2`. What does this mean?

A first solution ...

This is now correct, but yields a trivial solution!

soln1.

⇒ $0 + 0 = 0$
yes

There is an implicit assumption that the *initial digits* of SEND, MORE and MONEY are *not zero*, but we did not express this constraint in our query.

A second (non-)solution

So let's constrain S and M:

```
soln2 :-  digits([S,M]),
          not(S==0), not(M==0),           % backtrack if 0
          digits([N,D,M,O,R,E,M,O,N,E,Y]),
          A is 1000*S + 100*E + 10*N + D,
          B is 1000*M + 100*O + 10*R + E,
          C is 10000*M + 1000*O + 100*N + 10*E + Y,
          C is A+B,
          showAnswer(A,B,C).
```

A second (non-)solution ...

Maybe it works. We'll never know ...

```
soln2.
```

```
⇒ [Execution aborted]
```

after 8 minutes still running ...

What went wrong?

Before reading ahead, try to figure out what went wrong this time!

A third solution

Let's try to exercise more control by *instantiating variables bottom-up*:

```
sum([ ], 0).  
sum([N|L], TOTAL) :-    sum(L, SUBTOTAL),  
                        TOTAL is N + SUBTOTAL.  
  
% Find D and C, where  $\sum L$  is  $D + 10 * C$ , digit(D)  
carrysum(L, D, C) :- sum(L, S), C is S/10, D is S - 10*C.
```

```
?- carrysum([5,6,7], D, C).  
⇒ D = 8  
   C = 1
```

The problem is simply that *the search space is too large*.

Luckily there is more information that we can exploit. We know, for example, that either $D+E=Y$ or $D+E=Y+10$.

Similarly, if the first addition yielded a carry digit of $C=0$ or $C=1$, then either $C+N+R=E$, or $C+N+R=E+10$, and so on.

By *instantiating the variables from right to left*, we will reduce our search space dramatically, since variables must be instantiated so they solve the sums.

A third solution ...

We instantiate the final digits first, and use the carrysum to *constrain the search space*:

```
soln3 :- digits([D,E]), carrysum([D,E],Y,C1),
          digits([N,R]), carrysum([C1,N,R],E,C2),
          digit(0), carrysum([C2,E,0],N,C3),
          digits([S,M]), not(S==0), not(M==0),
          carrysum([C3,S,M],O,M),
          A is 1000*S + 100*E + 10*N + D,
          B is 1000*M + 100*O + 10*R + E,
          C is A+B,
          showAnswer(A,B,C).
```

Note that by the time we reach the last carrysum, the wanted result that C is $A+B$ is *guaranteed by construction*, so we should no longer have to backtrack before this point!

A third solution ...

This is also correct, but uninteresting:

soln3.

⇒ $9000 + 1000 = 10000$
yes

Once again, before reading ahead, ask yourself *what went wrong?*

A fourth solution

Let's try to make the variables *unique*:

```
% There are no duplicate elements in the argument list  
unique([X|L]) :- not(in(X,L)), unique(L).  
unique([]).
```

```
in(X, [X|_]).
```

```
in(X, [_|L]) :- in(X, L).
```

```
?- unique([a,b,c]).
```

```
⇒ yes
```

```
?- unique([a,b,a]).
```

```
⇒ no
```


The problem was that we failed to express the implicit assumption that *every variable in the puzzle represents a different digit*.

We therefore introduce a predicate to ensure that each number in a given list is a unique value.

A fourth solution ...

```
soln4 :- L1 = [D,E], digits(L1), unique(L1),
        carrysum([D,E],Y,C1),
        L2 = [N,R,Y|L1], digits([N,R]), unique(L2),
        carrysum([C1,N,R],E,C2),
        L3 = [O|L2], digit(O), unique(L3),
        carrysum([C2,E,O],N,C3),
        L4 = [S,M|L3], digits([S,M]),
            not(S==0), not(M==0), unique(L4),
        carrysum([C3,S,M],O,M),
        A is 1000*S + 100*E + 10*N + D,
        B is 1000*M + 100*O + 10*R + E,
        C is A+B,
        showAnswer(A,B,C).
```

As we consider more and more digits, working from right to left, we ensure that they are all unique.

A fourth solution ...

This works (at last), in about 1 second on a G3 Powerbook.

```
soln4.
```

```
⇒ 9567 + 1085 = 10652  
yes
```

OK, so it was a long time ago when I last timed it.

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

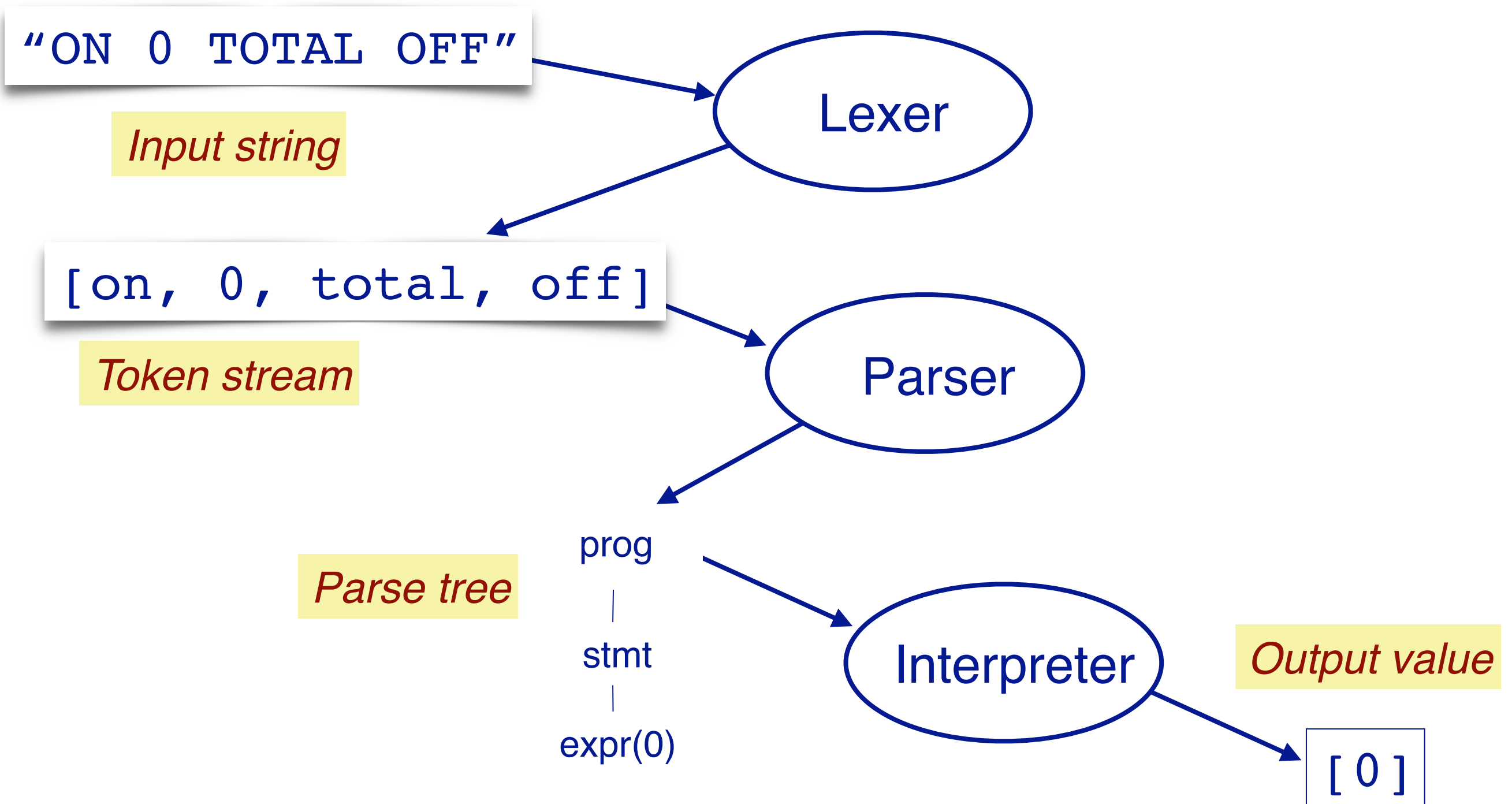
2. Symbolic Interpretation

- Definite Clause Grammars
- Interpretation as Proof
- An interpreter for the calculator language

2. Symbolic Interpretation

- > *Prolog is an ideal language for implementing small languages:*
 - Implement BNF using *Definite Clause Grammars*
 - Implement semantic rules directly as *Prolog rules*

Goal-directed interpretation



We will use Definite Clause Grammars to implement both the lexer and parser as a set of rules that first transform input strings to streams of tokens, and then transform these token streams to structured parse trees.

Then we will implement the interpreter as well as a set of rules that interpret parse trees and transform them to the final output.

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

2. Symbolic Interpretation

- **Definite Clause Grammars**
- Interpretation as Proof
- An interpreter for the calculator language

Definite Clause Grammars

Definite clause grammars are an extension of context-free grammars.

A DCG rule in Prolog takes the general form:

```
head --> body.
```

meaning “a possible form for head is body”.

The head specifies a non-terminal symbol, and the body specifies a sequence of terminals and non-terminals.

DCG rules can express context free grammars, and are written in a form that resembles BNF (Backus-Naur Form). Each rule has as its head a non-terminal, and as its body a sequences of terminals and non-terminals. Alternatives are expressed using multiple rules.

Definite Clause Grammars ...

- > *Non-terminals* may be *any Prolog term* (other than a variable or number).
- > A sequence of zero or more *terminal* symbols is written as *a Prolog list*. A sequence of ASCII characters can be written as a string.
- > *Side conditions* containing Prolog goals may be written in *{ } braces* in the right-hand side of a grammar rule.

Side conditions are used to produce side effects when the DCG parses actual input.

DCG translation

Grammar rules are just syntactic sugar for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis.

```
p(X) --> q(X).
```

translates to

```
p(X, S0, S) :- q(X, S0, S).
```

Stuff to parse

What's left after parsing

```
p(X, Y) -->  
  q(X),  
  r(X, Y),  
  s(Y).
```

translates to

```
p(X, Y, S0, S) :-  
  q(X, S0, S1),  
  r(X, Y, S1, S2),  
  s(Y, S2, S).
```

As we just saw, a non-terminal can be an arbitrary Prolog term, taking any number of arguments to represent any useful information about that non-terminal (i.e., to represent its value).

In order to parse an input stream, the DCG needs to keep track of where it is in the stream, so we need additional arguments to keep track of this information. These additional arguments clutter up the rules, but they are purely boilerplate (i.e., they can be automatically generated as syntactic sugar).

Here we see how a rule of the form `head --> body` expands to an ordinary Prolog rule with the missing arguments being generated.

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

2. Symbolic Interpretation

- Definite Clause Grammars

- **Interpretation as Proof**

- An interpreter for the calculator language

Example

This grammar parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.  
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.  
expr(X) --> term(X).
```

```
term(Z) --> number(X), "*", term(Y), {Z is X * Y}.  
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.  
term(Z) --> number(Z).
```

```
number(C) --> "+", number(C).  
number(C) --> "-", number(X), {C is -X}.  
number(X) --> [C], {0'0=<C, C=<0'9, X is C - 0'0}.
```

This is the canonical example of a DCG grammar for simple arithmetic expressions. Non-terminals are `expr`, `term`, and `number`, and terminals are arithmetic operators and digits.

Parameters to the non-terminals represent the values of those non-terminals, which are computed during the parse. For example, `expr (Z)` represents a parsed expression the value of which is the number `Z`. `number (X)` represents a parsed number `X`, consisting of a number of recognised digits.

Side conditions are used to compute the values of the non-terminals.

The last line says: `X` is a number if we consume `[C]`, where `C` is a single character between ascii `' 0 '` (`0 ' 0` in Prolog) and ascii `' 9 '`. The value of `X` is the difference between `C` and ascii `' 0 '`.

Note that this is a *scannerless parser*: there is no separate lexical analysis phase, but the grammar is directly expressed in terms of individual characters.

How does it work?

DCG rules are just syntactic sugar for normal Prolog rules.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
```

translates to:

```
expr(Z, S0, S) :-                               % input and goal
    term(X, S0, S1),                             % pass along state
    'C'(S1, 43, S2),                             % "+" = [43]
    expr(Y, S2, S),
    Z is X + Y .
```

'C' is a built-in predicate to recognize terminals.

Here we see a concrete example of how DCG rules are translated to normal Prolog rules.

To use such a rule, we set `S0` to the input to be parsed, and `S` to the desired remaining input (i.e., the empty input).

Note how terminal strings are matched using a special built-in DCG predicate called `'C'` that recognizes a single ASCII character (`'+'` is ASCII 43).

How to use this?

The query

```
?- expr(Z, "-2+3*5+1", []).
```

will compute $Z=14$.

Here we explicitly bind the generated parameters S_0 to an input string and the target S to the empty (fully consumed) input.

Roadmap



1. Search problems

- $\text{SEND} + \text{MORE} = \text{MONEY}$

2. Symbolic Interpretation

- Definite Clause Grammars

- Interpretation as Proof

- **An interpreter for the calculator language**

Recall our “calculator language”

A Calculator Language

Abstract Syntax:

```
Prog    ::= 'ON' Stmt
Stmt    ::= Expr 'TOTAL' Stmt
        | Expr 'TOTAL' 'OFF'
Expr    ::= Expr1 '+' Expr2
        | Expr1 '*' Expr2
        | 'IF' Expr1 ',' Expr2 ',' Expr3
        | 'LASTANSWER'
        | '(' Expr ')'
        | Num
```

The program “ON 4 * (3 + 2) TOTAL OFF” should print out 20 and stop.

Lexical analysis

We can use DCGs for both scanning and parsing.

Our lexer will convert an input atom into a list of tokens:

```
lex(Atom, Tokens) :-  
    name(Atom, String),  
    scan(Tokens, String, []), !.  
  
scan([T|Tokens]) --> whitespace0, token(T), scan(Tokens).  
scan([]) --> whitespace0.
```

Instead of implementing a scannerless parser, we will have a separate lexical analysis phase.

Following the orthodox approach, our scanner will recognize sequences of characters in the input and convert them into a stream of tokens, while our parser will recognize structure in sequences of tokens and organize them into parse trees.

A “normal” scanner would use regular expressions to recognize efficiently recognize tokens. We will use DCGs for both the scanner and the parser. Since DCGs express context-free grammars, which are strictly more powerful than regular expressions, we can use them for this task, even if they are more expressive than what we need.

Our lexer takes as input a Prolog atom (a single-quoted string literal), converts it to a regular Prolog string, and then uses the DCG `scan` non-terminal to recognise this as a list of tokens. The scanner uses a built-in recognizer to skip whitespace.

Recognizing Tokens

We will represent simple tokens by Prolog atoms:

```
token(on)      --> "ON" .
token(total)   --> "TOTAL" .
token(off)     --> "OFF" .
token(if)      --> "IF" .
token(last)    --> "LASTANSWER" .
token(' , ' )  --> " , " .
token(' + ' )  --> " + " .
token(' * ' )  --> " * " .
token(' ( ' )  --> " ( " .
token(' ) ' )  --> " ) " .
```

Our scanner turns the recognized strings into Prolog “atoms” (terms without any body).

As before, note how the result of parsing is expressed as a parameter to the rule. This may seem backwards from a functional perspective (a function or procedure should “return” something), but is perfectly normal from a logic programming perspective. The rule expresses a logical fact, for example:

```
token(on) --> "ON" .
```

expresses the logical conclusion that parsing "ON" yields the token on. It does not “return” anything in the usual sense.

Recognizing Numbers

and a number N by the term $\text{num}(N)$:

```
token(num(N))      --> digits(DL), { asnum(DL, N, 0) }.
```

```
digits([D|L])      --> digit(D), digits(L).
```

```
digits([D])        --> digit(D).
```

```
digit(D)           --> [D], { "0" =< D, D =< "9" }.
```

How would you implement $\text{asnum}/3$?

This would be easier to express with a regular expression: all we are saying is that a number consists of one or more digits.

The `asnum/3` predicate must take a list of characters representing digits and convert them into the decimal number they represent. How would you implement it?

Concrete Grammar

To parse a language, we need an unambiguous (i.e. concrete) grammar!

```
p      ::= 'ON' s
s      ::= e 'TOTAL' s
        | e ' TOTAL' 'OFF'
e      ::= e0
e0     ::= 'IF' e1 ',' e1 ',' e1
        | e1
e1     ::= e2 '+' e1
        | e2
e2     ::= e3 '*' e2
        | e3
e3     ::= 'LASTANSWER'
        | num
        | '(' e0 ')'
```

The abstract grammar we saw at the beginning of this section is ambiguous. In particular it does not know whether to parse " $1+2*3$ " as $(1+2)*3$ or $1+(2*3)$. The concrete grammar we see now forces the second parse.

For more on this topic, please consult the [“Compiler Construction”](#) course.

Parsing with DCGs

The concrete grammar is easily written as a DCG:

```
prog(S)                --> [on], stmt(S).
stmt([E|S])             --> expr(E), [total], stmt(S).
stmt([E])               --> expr(E), [total, off].
expr(E)                --> e0(E).
e0(if(Bool, Then, Else)) --> [if], e1(Bool), [' , '],
                             e1(Then), [' , '], e1(Else).
e0(E)                  --> e1(E).
e1(plus(E1,E2))         --> e2(E1), [' + '], e1(E2).
e1(E)                  --> e2(E).
e2(times(E1,E2))        --> e3(E1), [' * '], e2(E2).
e2(E)                  --> e3(E).
e3(last)                --> [last].
e3(num(N))              --> [num(N)].
e3(E)                  --> [' ( '], e0(E), [' ) '].
```

Note how the BNF is easily rewritten as a DCG. The thing to watch for is that the head of each rule should be *parameterized* with the result of the parse, i.e., *a Prolog term representing the parse tree*.

A program is represented by the statement it contains. A statement is a list of expressions. An expression is represented by an `if / 3` term, a `plus / 2` term, a `times / 2` term, a `last / 0`, or a `num / 1` term.

Representing Programs as Parse Trees

We have chosen to represent *expressions as Prolog terms*, and *programs and statements as lists of terms*:

```
parse(Atom, Tree) :-  
    lex(Atom, Tokens),  
    prog(Tree, Tokens, []).  
  
parse(  
    'ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF',  
    [ times(plus(num(1), num(2)),  
             plus(num(3), num(4))),  
      plus(last, num(10))  
    ]).
```

Our parser first invokes the lexer to convert the input atom into a list of tokens, and then it produces a parse tree consisting of a list of expression trees.

We can invoke it as follows:

```
?- parse('ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL  
OFF',T) .
```

T =

```
[times(plus(num(1),num(2)),plus(num(3),num(4))),plus  
(last,num(10))]
```

yes

Testing

We exercise our parser with various test cases:

```
check(Goal) :- Goal, !.  
check(Goal) :-  
    write('TEST FAILED: '),  
    write(Goal), nl.  
  
parseTests :-  
    check(parse('ON 0 TOTAL OFF', [num(0)])),  
    ...
```


Interpretation as Proof

- > One can view the execution of a program as a step-by-step “proof” that the program reaches some terminating state, while producing output along the way.
 - The program and its intermediate states are represented as structures (typically, as syntax trees)
 - Inference rules express how one program state can be transformed to the next

Building a Simple Interpreter

We define semantic predicates over the syntactic elements of our calculator language.

```
eval(Expr, Val) :- parse(Expr, Tree), peval(Tree, Val).
peval(S,L)      :-      seval(S, 0, L).
seval([E], Prev, [Val]) :-      xeval(E, Prev, Val).
seval([E|S], Prev, [Val|L]) :-  xeval(E, Prev, Val),
                                seval(S, Val, L).

xeval(num(N), _, N).
xeval(plus(E1,E2), Prev, V) :-  xeval(E1, Prev, V1),
                                xeval(E2, Prev, V2),
                                V is V1+V2.

...
```

```
eval('ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF', X).
⇒ X = [21, 31]
```

Compare these interpretation rules to those we defined in Haskell. They are very similar, except that the “return values” are now encoded as explicit *parameters* to the Prolog predicates. For example, instead of:

```
pp (On s) = ss s 0
```

we now write:

```
peval(S,L) :- seval(S, 0, L).
```

Testing the interpreter

We similarly define tests for the interpreter.

```
evalTests :-  
    check(eval('ON 0 TOTAL OFF', [0])),  
    check(eval('ON 5 + 7 TOTAL OFF', [12])),  
    ...
```

A top-level script

Finally, we can package the interpreter as a ciao module, and invoke it from a script:

```
#!/bin/sh
exec ciao-shell $0 "$@" # -*- mode: ciao; -*-
:- use_module(calc, [eval/2, test/0]).
main([])                :- test.
main(Argv)              :- doForEach(Argv).
doForEach([]).
doForEach([Arg|Args]) :-
    write(Arg), nl,
    eval(Arg, Val),
    write(Val), nl,
    doForEach(Args).
```

This script uses a common hack. The host O/S sees this is a script and passes it to the Bourne shell (`/bin/sh`). The first executable line replaces the shell by executing instead the `ciao-shell` interpreter, while passing the script file (`$0`) as the first argument, and the remaining arguments as `$@`. The `ciao` shell ignores the first two lines of the script and executes the rest as regular Prolog code.

We can now run the script as follows:

```
./crun.sh 'ON 0 TOTAL OFF'
```

(See `runCalcExamples.sh` in the repo.)

Roadmap



Epilogue: the lambda calculus interpreter

The Lambda interpreter (excerpt)

$$(\lambda x . e_1) e_2 \rightarrow [e_2/x] e_1$$
$$\lambda x . e x \rightarrow e \text{ } x \text{ not free in } e$$

```
% beta reduce
red(apply(lambda(X,Body), E), SBody) :-
    subst(X, E, Body, SBody).

% eta reduce
red(lambda(X, apply(F, name(X))), F) :-
    fv(F, Free),
    not(in(X, Free)).

% reduce LHS (lazy)
red(apply(LHS, RHS), apply(F,RHS)) :-
    red(LHS, F).

redAll(Tree, End) :-
    red(Tree, Next), !,
    redAll(Next, End).
redAll(End, End).

% Reduce to normal form
nf(Expr, NF) :-
    parse(Expr, Tree), !,
    redAll(Tree, End),
    unParse(End, NF).
```


The lambda calculus interpreter uses the same approach as we have just seen. (See `lambda.pl`, `lrun.sh` and `runLambdaExamples.sh` in the examples repo.)

Instead of implementing a denotational semantics, however, we implement *reduction semantics* over lambda expressions, that is, we implement beta and eta reduction. (Alpha conversion is not a reduction, so we do not need it to evaluate expressions.)

Here we have *lazy semantics*, but we can easily change the reduction rule to be strict.






Running the interpreter

```
(\True.\False.(\Not.Not True t f) (\b.b False True)) (\x.\y.x) (\x.\y.y)
-> (\False.(\Not.Not (\x.\y.x) t f) (\b.b False (\x.\y.x))) (\x.\y.y)
-> (\Not.Not (\x.\y.x) t f) (\b.b (\x.\y.y) (\x.\y.x))
-> (\b.b (\x.\y.y) (\x.\y.x)) (\x.\y.x) t f
-> (\x.\y.x) (\x.\y.y) (\x.\y.x) t f
-> (\y.\x.\y.y) (\x.\y.x) t f
-> (\x.\y.y) t f
-> (\y.y) f
-> f
```









Here we bind `True` to `(\x.\y.x)`, `False` to `(\x.\y.y)` and `Not` to `(\b.b False True)`, and then we evaluate `Not True` to `f`. The result, reassuringly, is `f`.

See `runLambdaExamples.sh` for examples of lambda encodings of numbers, and the use of `Y` to implement a recursive addition function.

What you should know!

-  *What are definite clause grammars?*
-  *How are DCG specifications translated to Prolog?*
-  *Why are abstract grammars inappropriate for parsing?*
-  *Why are left-associative grammar rules problematic?*
-  *How can we represent syntax trees in Prolog?*

Can you answer these questions?

-  *What happens when we ask `digits([A,B,A])`?*
-  *How many times will `soln2` backtrack before finding a solution?*
-  *How would you check if the solution to the puzzle is unique?*
-  *How would you generalize the puzzle solution to solve arbitrary additions?*
-  *Why must DCG side conditions be put in { curly brackets }?*
-  *What exactly does the 'C' predicate do?*
-  *Why do we need a separate lexer?*
-  *How would you implement an interpreter for the assignment language we defined earlier?*



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>