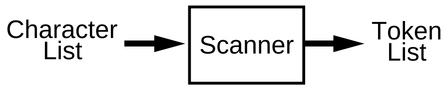
Scanning



Recognizing Kinds of Characters

```
lower(C) :- 97=<C, C=<122.          % a-z
upper(C) :- 65=<C, C=<90.           % A-Z
digit(C)   :- 48 =< C, C=< 57.      % 0-9

space(32).        tabch(9).          period(46).
slash(47).        endline(10).       endfile(26).
endfile(-1).

whitespace(C) :- space(C); tabch(C); endline(C).

idchar(C) :- lower(C) ; digit(C).
```


Reserved Words

```
reswd(program).        reswd(is).
reswd(begin).          reswd(end).
reswd(integer).        reswd(read).
reswd(var).            reswd(if).
reswd(boolean).        reswd(write).
reswd(while).          reswd(do).
reswd(then).           reswd(else).
reswd(skip).           reswd(or).
reswd(and).            reswd(true).
reswd(false).          reswd(not).
```


Reading Tokens

gettoken(C,T,D)
- \+ C is the lookahead character
- – Construct the next token T
- – Find the next lookahead character D

restid(C,[C|Lc],E)
- \+ C is the lookahead character
- – Construct rest of identifier or reserved word in string Lc
- – Find the next lookahead character E

getch(C)
- – Get the next character C and echo that character

Identifiers and Reserved Words:

```
gettoken(C,T,E) :- lower(C), getch(D),
                   restid(D,Lc,E), name(I,[C|Lc]),
                   (reswd(I),T=I ; T=ide(I)).

  restid(C,[C|Lc],E) :-  idchar(C), getch(D),
                         restid(D,Lc,E).

  restid(C,[ ],C).    % end of identifier if C
                                 not an id char
```

Numerals are handled in a similar manner using three clauses:

```
  gettoken(C,num(N),E) :-   digit(C), getch(D),
                            restnum(D,Lc,E),
                            name(N,[C|Lc]).
  restnum(C,[C|Lc],E) :-   digit(C), getch(D),
                           restnum(D,Lc,E).
  restnum(C,[ ],C).
```

Special Symbols:

```
single(40,lparen).      single(41,rparen).
single(42,times).       single(43,plus).
single(44,comma).       single(45,minus).
single(47,divides).     single(59,semicolon).
single(61,equal).

double(58,colon).
double(60,less).
double(62,grtr).

pair(58,61,assign).     % :=
pair(60,61,lteq).       % <=
pair(60,62,neq).        % <>
pair(62,61,gteq).       % >=

gettoken(C,T,D) :- single(C,T), getch(D).

gettoken(C,T,E) :- double(C,U), getch(D),
                   (pair(C,D,T),getch(E) ;
                    T=U,E=D).
```


End of Text

```
gettoken(C,eop,0) :- endfile(C).
```

Whitespace

```
gettoken(C,T,E) :- whitespace(C), getch(D),
                   gettoken(D,T,E).
```

Everything Else

```
gettoken(C,T,E) :- write('Illegal character: '),
                   put(C), nl, abort.
```


Scanning Predicate

```
scan([T|Lt]) :- tab(4), getch(C),
                gettoken(C,T,D), restprog(T,D,Lt).

getch(C) :- get0(C), (endline(C), nl, tab(4) ;
                      endfile(C), nl ;
                      put(C)).

restprog(eop,C,[ ]).% end of file reached with
                      previous character

restprog(T,C,[U|Lt]) :- gettoken(C,U,D),
                        restprog(U,D,Lt).
```


Example

Input Stream: "done := true;"

List of ascii characters =

[100, 111, 110, 101, 32, 58, 61,
32, 116, 114, 117, 101, 59, -1]

getch(C)	C = 100
gettoken(100,T,E)	
lower(100)	
getch(D)	D = 111
restid(111,Lc,E)	
restid(111,[111|Lc1],E)	Lc = [111|Lc1]
idchar(111)	
getch(D1)	D1 = 110
restid(110,Lc1,E)	
restid(110,[110|Lc2],E)	Lc1 = [110|Lc2]
idchar(110)	
getch(D2)	D2 = 101
restid(101,Lc2,E)	

```
        restid(101,[101|Lc3],E)        Lc2 = [101|Lc3]
          idchar(101)
          getch(D3)                    D3 = 32
          restid(32,Lc3,E)             Lc3 = [ ]
                                       E = 32
```

Lc = [111|Lc1] = [111,110|Lc2]
= [111,110,101|Lc3] = [111,110,101]

```
  name(I,[100,111,110,101])            I = done
  (reswd(I),T=I ; T=ide(I))            T = ide(done)

gettoken(32,T1,E1)
  whitespace(32)
  getch(D4)                            D4 = 58

gettoken(58,T1,E1)
  double(58,colon)
  getch(D5)                            D5 = 61
  pair(58,61,T1)                       T1 = assign
  getch(E1)                            E1 = 32
```



```
gettoken(32,T2,E2)
  whitespace(32)
  getch(D6)                            D6 = 116

gettoken(116,T2,E2)
  lower(116)
  getch(D7)                            D7 = 114
  restid(114,Lc4,E2)
  restid(114,[114|Lc5],E2)             Lc4 = [114|Lc5]
    idchar(114)
    getch(D8)                          D8 = 117
    restid(110,Lc5,E2)
    restid(117,[117|Lc6],E2)           Lc5 = [117|Lc6]
      idchar(117)
      getch(D9)                        D9 = 101
      restid(101,Lc6,E2)
      restid(101,[101|Lc7],E2)         Lc6 = [101|Lc7]
        idchar(101)
        getch(D10)                     D10 = 59
```



```
        restid(59,Lc7,E2)              Lc7 = [ ]
                                       E2 = 59
```

Lc4 = [114|Lc5] = [114,117|Lc6]
= [114,117,101|Lc7] = [114,117,101]

```
  name(I1,[116,114,117,101])           I1 = true
  reswd(true)                          T2 = true

gettoken(59,T3,E3)
  single(59,semicolon)                 T3 = semicolon
  getch(D11)                           D11 = -1
                                       E3 = -1

gettoken(-1,T4,E4)
  endfile(-1)                          T4 = eop
                                       E4 = 0
```

Tokens:

T = ide(done)   T1 = assign   T2 = true
T3 = semicolon   T4 = eop

scan(L) returns

L = [ide(done), assign, true, semicolon, eop]

Controling the System

```
go :- nl, write('>>> Scanning Wren <<<'), nl, nl,
      write('Enter name of source file: '), nl,
      getfilename(File), nl,
      see(File), scan(Tokens), seen,
      write('Scan successful'), nl,
      write(Tokens), nl.
```

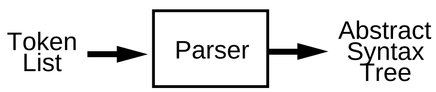
Read a File Name

```
getfilename(W) :- get0(C), restfilename(C,Cs),
                  name(W,Cs).

restfilename(C,[C|Cs]) :- filechar(C), get0(D),
                          restfilename(D,Cs).

restfilename(C,[ ]).

filechar(C) :- lower(C) ; upper(C) ; digit(C) ;
               period(C) ; slash(C).
```

Try It cp ~slonnegr/public/plf/scan .

cp ~slonnegr/public/plf/prime.w .

Parsing



Logic Grammars

Joke: Prolog was invented by Robert Kowalski in 1974 and implemented by Alain Colmerauer in 1973.

Explanation

Prolog originated out of Colmerauer’s interest in using logic to express grammar rules and to formalize the parsing of natural language sentences.

It was Kowalski who saw the power of logic programming as a general purpose programming language.

Concrete Syntax

<sentence> ::= <noun phrase> <verb phrase> **.**

<noun phrase> ::= <determiner> <noun>

<verb phrase> ::= <verb> | <verb><noun phrase>

<determiner> ::= **a** | **the**

<noun> ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**

<verb> ::= **saw** | **touched** | **surprised** | **sang**

Abstract Syntax

Sentence ::= NounPhrase Predicate

NounPhrase ::= Determiner Noun

Predicate ::= Verb | Verb NounPhrase

Determiner ::= **a** | **the**

Noun ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**

Verb ::= **saw** | **touched** | **surprised** | **sang**

Example

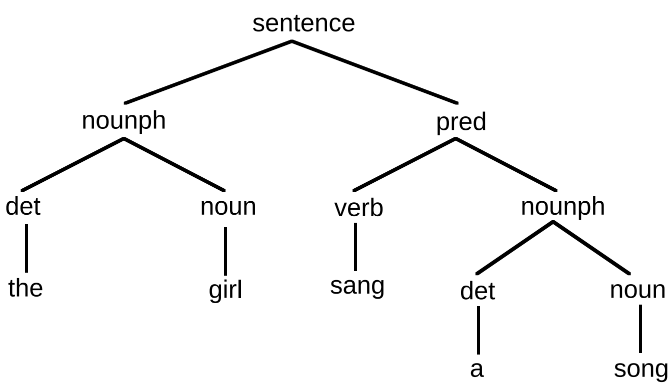
“the girl sang a song.”

Scanner

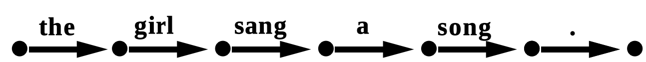
[the, girl, sang, a, song, '.']

Parser

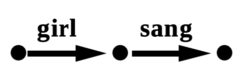
sent(nounph(det(the), noun(girl)),
pred(verb(sang),
nounph(det(a), noun(song))))



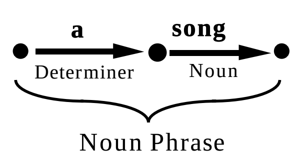
A Graph



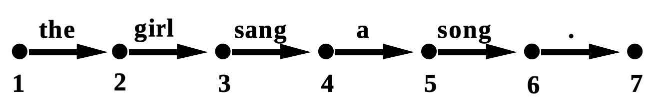
Contiguous Tokens



Forming a Nonterminal



A Labeled Graph



Prolog Rules (Version 1)

sentence(K,L) :-
    nounPhrase(K,M), predicate(M,N), period(N,L).

nounPhrase(K,L) :- determiner(K,M), noun(M,L).

predicate(K,L) :- verb(K,M), nounPhrase(M,L).

predicate(K,L) :- verb(K,L).

determiner(K,L) :- a(K,L).

noun(K,L) :- boy(K,L).

verb(K,L) :- saw(K,L).

Creating the Graph

the(1,2).	girl(2,3).	sang(3,4).
a(4,5).	song(5,6).	period(6,7).

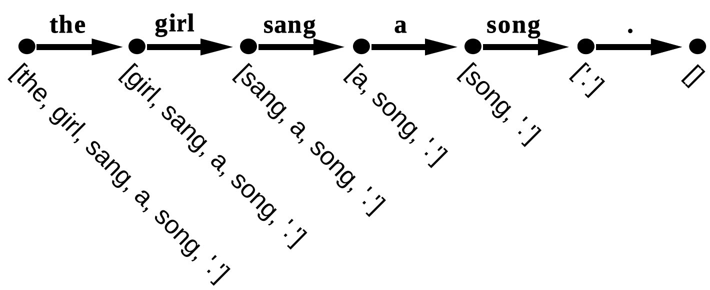
Queries

```
?- sentence(1,7).
yes
?- sentence(X,Y).
X = 1
Y = 7
yes
```


Problems

1. Linkage between scanner and parser.
2. Creating the abstract syntax tree.

New Labels for the Graph



Differences Lists

Connect Predicate: 'C'(Left,Token,Right).

The edge from node Left to node Right is labeled with atom Token.

Connect, 'C', is a predefined predicate.

Definition: 'C'([H|T],H,T).

Prolog Rules (Version 2)

sentence(K,L) :-
    nounPhrase(K,M), predicate(M,R), 'C'(R,'.',L).

nounPhrase(K,L) :- determiner(K,M), noun(M,L).

predicate(K,L) :- verb(K,M), nounPhrase(M,L).

predicate(K,L) :- verb(K,L).

determiner(K,L) :- 'C'(K,a,L).

noun(K,L) :- 'C'(K,boy,L).

verb(K,L) :- 'C'(K,saw,L).

Queries

```
?- sentence(
   [the,girl,sang,a,song,'.'],[]).
yes
?- sentence(S,[]).
```

624 possible answers

Preprocessor

Most Prolog implementations have a preprocessor using "-->" for grammar clauses.

Prolog Rules (Version 3)

sentence --> nounPhrase, predicate, ['**.**'].

nounPhrase --> determiner, noun.

predicate --> verb, nounPhrase.

predicate --> verb.

determiner --> [a].

determiner --> [the].

noun --> [boy] ; [girl] ; [cat] ;
    [telescope] ; [song] ; [feather].

verb --> [saw] ; [touched] ; [surprised] ; [sang].

Version 3 is *automatically* translated into Version 2.

Parameters in Grammars

Logic grammar rules allow additional parameters that are inserted if front of the implicit arguments.

Prolog Rules (Version 4)

```
sentence(sent(N,P)) -->
        nounPhrase(N), predicate(P), ['.'].
nounPhrase(nounph(D,N)) -->
        determiner(D), noun(N).
predicate(pred(V,N)) --> verb(V), nounPhrase(N).
predicate(pred(V)) --> verb(V).
determiner(det(a)) --> [a].
noun(noun(boy)) --> [boy].
verb(verb(saw)) --> [saw].
```

Query

```
?- sentence(Tree,
      [the,girl,sang,a,song,'.'],[]).
Tree=sent(nounph(det(the),noun(girl)),
   pred(verb(sang),nounph(det(a),noun(song))))
yes
```


Prolog Goals in a Logic Grammar

Terms within braces are not translated by the preprocessor.

Recognize the Language { $a^n b^n c^n \mid n \geq 0$ }

```
string --> getAs(M1), getBs(M2), getCs(M3),
           { M1=:=M2, M2=:=M3 }.

getAs(M) --> [a], getAs(N), { M is N+1 }.
getAs(0) -->  [ ].

getBs(M) --> [b], getBs(N), { M is N+1 }.
getBs(0) -->  [ ].

getCs(M) --> [c], getCs(N), { M is N+1 }.
getCs(0) -->  [ ].
```

Queries

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

?- string([a,a,b,b,b,c,c,c], [ ]).
no
```


Parsing Wren

```
program(AST) -->
        [program], [ide(I)], [is], block(AST).

block(prog(Decs,Cmds)) -->
      decs(Decs), [begin], cmds(Cmds), [end].

cmds(Cmds) --> command(Cmd),
               restcmds(Cmd,Cmds).
  restcmds(Cmd,[Cmd|Cmds]) -->
               [semicolon], cmds(Cmds).
  restcmds(Cmd,[Cmd]) --> [ ].

command(while(Test,Body)) -->
        [while], boolexpr(Test), [do],
               cmds(Body), [end, while].

command(assign(V,E)) -->
               [ide(V)], [assign], expr(E).
```


Handling Left Recursion

Recall “ancestor3” on pages 574-575.

Expression Example

<expr> ::= <expr> <opr> <numeral>

<expr> ::= <numeral>

<opr> ::= **+ | –**

<numeral> ::= …   % as before

Definite clause grammar

```
expr(plus(E1,E2)) --> expr(E1), ['+'], [num(E2)].
expr(minus(E1,E2)) --> expr(E1), ['–'], [num(E2)].
expr(E) --> [num(E)].
```

Query

```
?- expr(E, [num(5), '–', num(2)], [ ]).
```

➡ ➡ ➡ nontermination

Removing Left Recursion

<expr> ::= <numeral> <rest of expr>

<rest of expr> ::=
<opr> <numeral> <rest of expr>

<rest of expr> ::= ε

Logic Grammar

```
expr(E) --> [num(E1)], restexpr(E1,E).
restexpr(E1,E) -->
        ['+'], [num(E2)], restexpr(plus(E1,E2),E).
restexpr(E1,E) -->
     ['–'], [num(E2)], restexpr(minus(E1,E2),E).
restexpr(E,E)   --> [ ].
```


Parsing Wren Integer Expressions

```
expr(E) --> intexpr(E).
expr(E) --> boolexpr(E).

intexpr(E) --> term(T), restintexpr(T,E).
restintexpr(T,E) --> weakop(Op), term(T1),
                     restintexpr(exp(Op,T,T1),E).
restintexpr(E,E) --> [ ].

term(T) --> element(P), restterm(P,T).
restterm(P,T) --> strongop(Op), element(P1),
                  restterm(exp(Op,P,P1),T).
restterm(T,T) --> [ ].

element(num(N)) --> [num(N)].
element(ide(I)) --> [ide(I)].
element(E) --> [lparen], intexpr(E), [rparen].
element(minus(E)) --> [minus], element(E).
```


Left Factoring

<command> ::= …

| **if** <bool expr> **then** <cmd seq> **end if**

| **if** <bool expr> **then** <cmd seq>
else <cmd seq> **end if**

```
command(Cmd) -->
    [if], boolexpr(Test), [then], cmds(Then),
                restif(Test,Then,Cmd).

    restif(Test,Then,if(Test,Then,Else) -->
                [else], cmds(Else), [end], [if].

    restif(Test,Then,if(Test,Then)) -->
                [end], [if].
```

Try It cp ~slonnegr/public/plf/scanp **.**

cp ~slonnegr/public/plf/nodecs.w **.**