Syntax analysis, parsing

A parser for a CFG (*Context-Free Grammar*) is a program which determines whether a string `w` is part of the language L(G).

Function

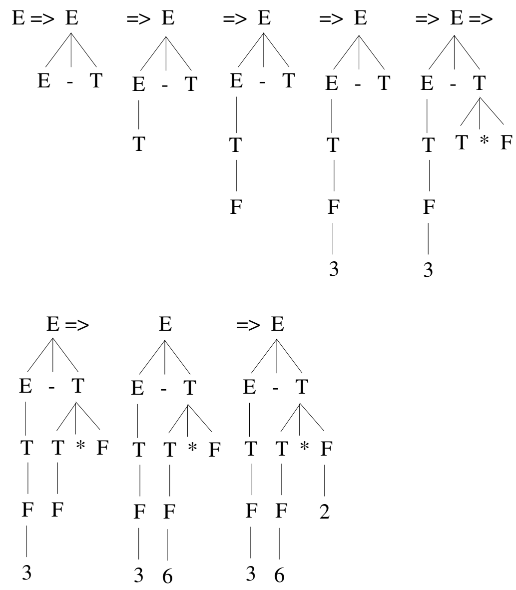
1. Produces a parse tree if $w \in$ L(G).
2. Calls semantic routines.
3. Manages syntax errors, generates error messages.

Input:
String (finite sequence of tokens)
Input is read from left to right.

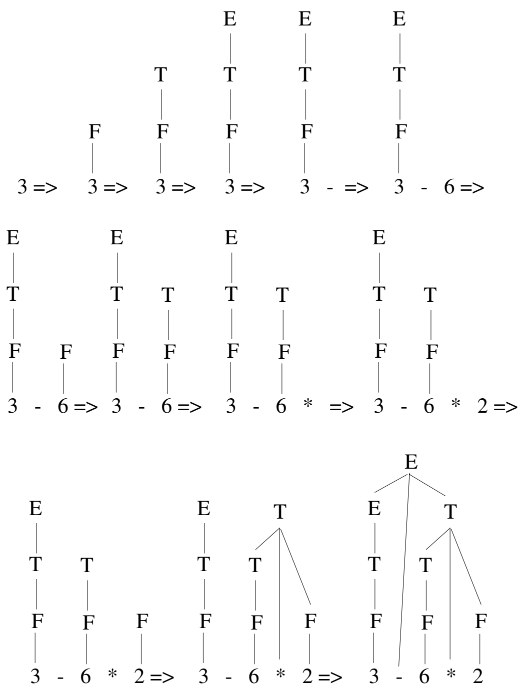
Output:
Parse tree / error messages

Example: **Top-down** parsing with input: `3 - 6 * 2`

E → E - T | T
T → T * F | F
F → Integer | (E)



Example: **Bottom-up** parsing with input: `3 - 6 * 2`
(same CFG as in the example)



Top-down analysis

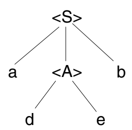
How do we know in which order the string is to be derived?

Use one or more tokens lookahead.

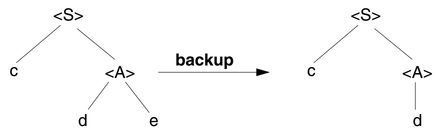
Example: **Top-down analysis with backup**

1. <S> → a <A> b *1 token lookahead works well*
2. | c <A> *1 token lookahead works well*
3. <A> → d e *test right side until something*
4. | d *fits*

a) adeb



b) cd



- Top-down analys with backup is implemented by writing a procedure or a function for each nonterminal whose task is to find one of its right sides:

```
function A:boolean;(* A → d e | d *)
var savep : ptr;
begin
   savep := inpptr;
   if inpptr^ = 'd' then begin
      scan;    (* Get next token,
                  move inpptr a step *)
      if inpptr^ = 'e' then begin
         scan;
         return(true);  (* 'de' found *)
      end;
   end;

inpptr := savep;   (* 'de' not found, back up and
                                          try 'd'*)
   if inpptr^ = 'd' then begin
      scan;
      return (true);    (* 'd' found, OK *)
   end;
   return(false);
end;
```



```
function S:boolean;   (* S → a A b | c A *)
begin
 if inpptr^ = 'a' then begin
    scan;
    if A then begin
       if inpptr^ = 'b' then begin
          scan;
          return(true);
       end
       else return(false)
    else return(false);
 end
 else if inpptr^ = 'c' then begin
         scan;
         if A then  return(true)
         else return(false);
       end
       else return (false);
end;
```


Construction of a top-down parser

Code the program as follows:

- Write a procedure for each nonterminal.
- Call scan directly after each token is consumed.
- Start by calling the procedure for the start symbol.
- At each step check the leftmost non-treated vocabulary symbol.
- If it is a terminal symbol

  Match it with the current token, and read the next token.

- If it is a nonterminal symbol

  Call the routine for this nonterminal.

- In case of error call the error management routine.

Example: An LL(1) grammar which describes binary numbers:

```
S → BinaryDigit BinaryNumber
BinaryNumber → BinaryDigit  BinaryNumber
             | ε
BinaryDigit → 0 | 1
```

<u>Sketch of a top-down parser (*recursive descent*):</u>

```
program TopDown(input,output);
   procedure BinaryDigit;
   begin
    if token in [0, 1]
    then scan
    else error(...)
   end; (* BinaryDigit *)

   procedure BinaryNumber;
   begin
    if token in [0, 1]
    then begin
           BinaryDigit;
           BinaryNumber
        end; (* OK for the case with ε *)
   end; (* B' *)

   procedure S;
   begin
     BinaryDigit;
     BinaryNumber;
   end; (* S *)
begin (*  main program *)
   scan;
   S;
   if not eof then error(...)
```

Non-LL(1) structures in a grammar:

- Left recursion

  Example:

```
E →  E - T
   |  T
```

- Productions for a nonterminal with the same prefix in two or more right sides

  Example:

```
arglist → ( )
        |  ( args )
```

or

```
A →  a b
  |  a c
```

The problem can be solved in most cases by rewriting the grammar to an LL(1), i.e. to a grammar that can be analysed using top-down methods.

How do you convert a grammar so that it can be analysed top-down?

1. Eliminate left recursion

   a) Transform the grammar to iterative form by using EBNF (*Extended BNF*):

   {β} same as the regular expression: β^*

   [β] same as the regular expression: $\beta \mid \varepsilon$

   ( ) left factoring, t ex

   $A \to ab \mid ac$ in EBNF is rewritten:

   $A \to a\ (b \mid c)$

   $A \to A\ \alpha \mid \beta$
   (where β may not be preceded by A)

   in EBNF is rewritten:

   $A \to \beta\ \{\alpha\}$

b) Transform the grammar to right recursive form using the rewrite rule:

$A \to A\ \alpha \mid \beta$ (where β may not be preceded by A)

is rewritten to

$A \to \beta\ A'$

$A' \to \alpha\ A' \mid \varepsilon$

Generally:

$A \to A\ \alpha_1 \mid A\ \alpha_2 \mid \dots \mid A\ \alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

(where β_1, β_2, ... may not be preceded by A)

is rewritten to:

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

2. Left factoring

```
<stmt> → if <expr> then <stmt>
       | if <expr> then <stmt> else <stmt>
```

Solution using EBNF:

```
<stmt>→ if <expr> then <stmt>
          [ else <stmt> ]
```

Solution using rewriting:

```
<stmt> → if <expr> then <stmt> <rest-if>
<rest_if> → else <stmt> | ε
```

Summary of the LL(1) grammar:

- Many CFGs are not LL(1)
- Some can be rewritten to LL(1)

\- The underlying structure is lost (because of rewriting).

Methods for writing a top-down parser

- Table-driven, LL(1)
- Recursive descent

LL(1)	Recursive Descent
Table-driven	Hand-written
+ fast	- much coding
+ good error management and restart	+ easy to include semantic actions

Example: **A recursive descent parser for Pascal-declarations**

<declarations> → <constdecl> <vardecl>
<constdecl> → CONST <consdeflist>
| ε
<consdeflist> → <consdeflist> <constdef>
| <constdef>

<constdef> → id = number ;
<vardecl> → VAR <vardeflist>
| ε
<vardeflist> → <vardeflist> <idlist> : <type> ;
| <idlist> : <type> ;
<idlist> → <idlist> , id
| id
<type> → integer
| real

Rewrite in EBNF so that a recursive descent parser can be written

<declarations> → <constdecl> <vardecl>
<constdecl> → CONST <consdef> { <consdef> }
| ε
<constdef> → id = number ;
<vardecl> → VAR <vardef> { <vardef> }
| ε
<vardef> → id { , id } : (integer | real) ;

A recursive descent parser for the new grammar in EBNF

- We have one character lookahead.
- `scan` should be called when we have consumed a character.

```
procedure declarations;
  (* <declarations> → <constdecl> <vardecl> *)
begin
  constdecl;
  vardecl;
end (* declarations *);

procedure constdecl;
  (* <constdecl> → CONST <consdef> { <consdef> }
                 | ε *)
begin
  if (token = 'CONST') then begin
    scan;
    if (token = 'id') then
       constdef
    else
       error('Missing id after CONST');
    while token = 'id' do
       constdef;
  end;
end (* constdecl *);
```



```
procedure constdef;
   (* <constdef> → id = number ; *)
  begin
    scan; (* consume 'id', get next token *)
    if (token = '=') then
      scan
    else
      error('Missing '=' after id');
    if (token = 'number') then
      scan
    else
      error('Missing number');
    if (token = ';') then
      scan (* consume ';', get next token *)
    else
      error('Missing ';' after const decl')
  end (* constdef *);

  procedure vardecl;
   (* <vardecl> → VAR <vardef> { <vardef> }
                | ε    *)
  begin
    if (token = 'VAR') then begin
      scan;
      if (token = 'id') then
        vardef
      else
        error('Missing id after VAR');
      while (token = 'id') do
        vardef;
    end;
  end (* vardecl *);
```

```
procedure vardef;
  (* <vardef> → id { , id } : ( integer | real ) ; *)
begin
   scan;
   while (token = ',') do begin
      scan;
      if (token = 'id') then
         scan
      else
         error('id expected after `,'');
   end (* while *);
   if (token = ':') then begin
      scan;
      if (token = 'integer') or
         (token = 'real')
      then scan
      else error('Incorrect type of
                                    variable');
      if (token = ';') then
         scan
      else
         error('Missing ';' in variable
                                    decl.');
   end else
     error('Missing `:` in var. decl.')
end (* vardef *);

begin (* main *)
   scan; (* lookahead token *)
   declarations;
   if token<>eof_token then error(...);
end (* main *).
```