A parser is a program that, given an input sentence, “recognizes” whether or not that sentence is in the language of a given grammar.

Works by reconstructing a derivation for the sentence.

Parser “constructs” parse tree:

- explicitly – actual data structure is built; or
- implicitly – “semantic actions” are invoked at points corresponding to nodes in the tree, but no tree is actually built.

All parsers read input left-to-right, but they differ in how tree is constructed: top-down vs. bottom-up.

Any context-free grammar can be parsed by a (nondeterministic) pushdown automaton (NFA + stack), but not necessarily by a deterministic one (much less an efficient one).
TOP-DOWN PARSING

Idea: construct parse tree by starting at start symbol and “guessing” each derivation until we reach a string that matches input.

Example Grammar:

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print} \mid \epsilon \\
E \rightarrow \text{true} \mid \text{false} \mid \text{id}
\]

Token string:

if id_b then while true do else print

Tree:

\[
S
\]

Input: if id_b then while true do else print

Action: Guess for \( S \)
Tree:

Input: if idb then while true do else print

Action:

if matches; guess for $E$
Tree:

Input: $id_b$ then while true do else print

Action:

$id_b$ matches; then matches; guess for $S$
Tree:

```
S
  /   \
/     \ 
if     then
|       |
|       E|
|       |
/       \
/         \
/             \
idb  while  E  do  S
```

Input: while true do else print

Action:

while matches; guess for $E$
Tree:

Input: true do else print

Action:

true matches; do matches; guess for $S$
Tree:

```
S
 /\  
/   \ 
S     S
|     /\ 
|    /   
|  /     
| /      
|/       
S       S
```

Input: `else print`

Action:

$\epsilon$ matches; else matches; guess for $S$
Input: print

Action:

print matches; input exhausted; done.
Implementation of top-down parser using a **recursive** procedure for each non-terminal.

For many languages, can make perfect guesses (avoid back-tracking) by using 1-symbol **lookahead**. I.e., if

\[ A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n \]

choose correct \( \alpha_i \) by looking at **first** symbol it derives.

(If \( \epsilon \) is an alternative, choose it last.)

This approach is also called **predictive parsing**

Recursive-descent parsers are easy to write by hand and reasonably efficient.

Often must massage grammar into suitable form (more later).

Not all languages can be parsed this way.
For the same grammar as before:

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S | \text{print } \epsilon \\
E \rightarrow \text{true } | \text{false } | \text{id}
\]

We write one function to parse expressions, and another to parse statements.

Each function returns normally if it successfully parsed; otherwise it calls `error()`, which we assume issues an error message and terminates the parser.

For simplicity in this example, the parser functions don’t return anything.

We assume that `lex()` returns the next token and advances the lexical analyzer’s token stream.

```c
void e() {
    if (tok == TRUE || tok == FALSE || tok == ID) 
        tok = lex();
    else error();
}
```
void s() {
    if (tok == IF) {
        tok = lex();    /* get next input token */
        e();
        if (tok == THEN) {
            tok = lex();
            s();    /* recursive call! */
            if (tok == ELSE) {
                tok = lex();
                s();
            } else error();    /* issue error message */
        } else error();
    } else if (tok == WHILE) {
        tok = lex();
        e();
        if (tok == DO) {
            tok = lex();
            s();
        } else error();
    } else if (tok == PRINT) {
        tok = lex();
    } else /* epsilon case falls out */
        /* issue error message */
}
PROBLEMS FOR RECURSIVE-DESCENT PARSING

- Left recursion: a derivation
  \[ A \Rightarrow^* A\alpha \]
  causes parser to loop!

Solution: **Remove** left recursion from grammar.

- Need to backtrack (inefficient) because one-symbol lookahead can’t “guess” correctly, e.g.:

  \[
  S \rightarrow V := \text{int} \\
  V \rightarrow \text{alpha} '[\text{int}]' | \text{alpha}
  \]

Possible inputs: \( x := 77 \) or \( x[2] := 17 \).

Which alternative should we choose for \( V \) ?

Solution: **Left-factor** the grammar.

- These problems arise naturally in expression grammars. (Can usually prevent them in statement grammars by careful language design.)
Eliminating Immediate Left Recursion

Replace left-recursive productions of the form

\[ A \rightarrow A\alpha \mid \beta \]

which generate sentences of the form

\[ \beta, \beta\alpha, \beta\alpha\alpha, \ldots \]

by the right-recursive productions

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' \mid \epsilon \]

Yields different parse trees but same language:
For a simplified expression grammar:

\[
E \rightarrow E \ + \ T \mid E \ - \ T \mid T
\]
\[
T \rightarrow T \ * \ F \mid T \ / \ F \mid F
\]
\[
F \rightarrow ( \ E ) \mid id
\]

becomes

\[
E \rightarrow T \ E'
\]
\[
E' \rightarrow + \ T \ E' \mid - \ T \ E' \mid \epsilon
\]
\[
T \rightarrow F \ T'
\]
\[
T' \rightarrow * \ F \ T' \mid / \ F \ T' \mid \epsilon
\]
\[
F \rightarrow ( \ E ) \mid id
\]

But note that the desired left-associativity has been lost!
ELIMINATING ALL LEFT-RECURSION

Consider

\[ S \rightarrow Aa \mid b \]

\[ A \rightarrow Sc \mid d \]

Non-terminal \( S \) is left-recursive in two steps:

\[ S \Rightarrow Aa \Rightarrow ScA \Rightarrow Aaca \Rightarrow Scaca \Rightarrow \ldots \]

Fairly General Algorithm

(Works unless \( A \xrightarrow{\pm} A \) or \( A \rightarrow \epsilon \). Fully general algorithm exists, but is complicated!)

- arrange non-terminals in some order \( A_1, \ldots, A_n \).
- for \( i = 1 \) to \( n \) do
  - for \( j = 1 \) to \( i-1 \) do
    - for any production \( A_i \rightarrow A_j \alpha \)
      replace it by substituting definition of \( A_j \) into r.h.s.,
      i.e., by changing it to
      \[ A_i \rightarrow \beta_1 \alpha \mid \ldots \mid \beta_m \alpha \]
      where current productions for \( A_j \) are
      \[ A_j \rightarrow \beta_1 \mid \ldots \mid \beta_m \]
    - eliminate any immediate left-recursion in \( A_i \)
**Example:** Consider a grammar with the non-terminals ordered as follows:

(1) \[ S \rightarrow Pa \mid b \]
(2) \[ P \rightarrow Sc \mid Rd \]
(3) \[ R \rightarrow Se \mid f \]

**Step i=2,j=1:** In-line (1) into (2), giving

(2) \[ P \rightarrow Pac \mid bc \mid Rd \]

and then remove immediate left-recursion, giving

(2) \[ P \rightarrow bcP' \mid RdP' \]
(2') \[ P' \rightarrow acP' \mid \epsilon \]

**Step i=3,j=1:** Inline (1) into (3), giving

(3) \[ R \rightarrow Pae \mid be \mid f \]

**Step i=3,j=2:** Inline new (2) into new (3), giving

(3) \[ R \rightarrow bcP'ae \mid RdP'ae \mid be \mid f \]

and then remove immediate left-recursion, giving

(3) \[ R \rightarrow bcP'aeR' \mid beR' \mid fR' \]
(3') \[ R' \rightarrow dP'aeR' \mid \epsilon \]
Easy to remove common prefixes by left-factoring, creating new non-terminals.

Change

\[ V \rightarrow \alpha\beta \mid \alpha\gamma \]

to

\[ V \rightarrow \alpha V' \]
\[ V' \rightarrow \beta \mid \gamma \]

Example:

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha } [\ ' \text{int } '] ]' \mid \text{alpha} \]

becomes

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha } V' \]
\[ V' \rightarrow [\ ' \text{int } '] ]' \mid \epsilon \]
Recall grammar of arithmetic expressions after removal of left recursion:

\[

e \rightarrow \text{T} \text{E}' \\
\text{E}' \rightarrow + \text{T} \text{E}' \mid \epsilon \\
\text{T} \rightarrow \text{F} \text{T}' \\
\text{T}' \rightarrow * \text{F} \text{T}' \mid \epsilon \\
\text{F} \rightarrow (\text{E}) \mid \text{id}
\]

This leads directly to this code for \( E \) and \( E' \) (\( T \) and \( T' \) are similar):

```c
int e() {  
    t();  
    e1();  
    return 0;  
}

int e1() {  
    if (tok == '+') {  
        tok = lex();  
        t();  
        e1();  
    }  
    /* epsilon case falls through */  
}
```


We can simplify this code (and improve its performance) by turning the recursions into iterations, e.g.:

```c
e1() {
    if (tok == '+') {
        tok = lex(); t(); e1();
    } /* epsilon case falls through */
}
```

becomes:

```c
e1() {
    while (tok == '+') {
        tok = lex(); t();
    }
}
```
We then inline functions that are (now) called only once, e.g.:

```c
e() { t(); e1(); }
```

becomes:

```c
e() {
    t();
    while (tok == '+') {
        tok = lex(); t();
    }
}
```

Performing the same transformation on \( t \) and \( t1 \), and adding the usual recursive descent code for \( f \), we get...
expression parsing using recursive-descent (iv)

e() {
    t();
    while (tok == ’+’) { lex(); t(); } 
}

t() {
    f();
    while (tok == ’*’) { lex(); f(); } 
}

f() {
    if (tok == ID)
        lex();
    else if (tok == ’(‘) {
        lex();
        e();
        if (tok == ’)’)
            lex();
        else error();
    else error();
}