Parser

- 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**
  - Produces a parse tree if w ∈ L(G).
  - Calls semantic routines.
  - Manages syntax errors, generates error messages.
- **Input:**
  - String (finite sequence of tokens)
  - Input is read from left to right.
- **Output:**
  - Parse tree / error messages

### Top-Down Parsing

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

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

#### Bottom-up Parsing

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

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

### Bottom-Up Parsing cont.

```
3 - 6 * 2
```

### 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 backtracking
  - `<S>` → a `<A>` b 1 token lookahead works well
  - `<A>` → c `<A>` d e test right side until something fits
  - Example: Top-down analysis with backtracking
  - `<S>` → a `<A>` b
  - `<A>` → c `<A>` d e
  - `<A>` → a `<A>` b c d e
  - Backtracking
Top-down Analys with Backtracking, cont.

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

```c
bool A() { /* A → d e | d */
    char* savep;
    savep = inpptr;
    if (*inpptr == 'd') {
        scan(); /* Get next token, move inpptr a step */
        if (*inpptr == 'e') {
            scan();
            return true; /* 'de' found */
        }
    }
    inpptr = savep;
    /* 'de' not found, backtrack and try 'd' */
    if (*inpptr == 'd') {
        scan(); return true; /* 'd' found, OK */
    }
    return false;
}
```


```c
bool S() { /* S -> a A b | c A */
    if (*inpptr == 'a') {
        scan();
        if A() {
            if (*inpptr == 'b') {
                scan();
                return true;
            } else return false;
        } else return false;
    }
    else if (*inpptr == 'c') {
        scan();
        if A() return true; else return false;
    }
    else return false;
}
```


Construction of a top-down parser

- Write a procedure for each nonterminal.
- Call scan directly after each token is consumed.
  - Reason: The look-ahead token should be available
- 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
```

A Top-Down Parser that does not Work, Infinite Recursion:

```c
void TopDown(input,output) {
    /* main program */
    S();
}
```

Grammar:

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

Sketch of a Top-Down Parser (recursive descent)

```c
void TopDown(input,output) {
    /* main program */
    S();
}
```

Grammar:

```
S → BinaryDigit BinaryNumber
BinaryNumber → BinaryDigit BinaryNumber | ε
BinaryDigit → 0 | 1
```
Non-LL(1) Structures in a Grammar:

- Left recursion, example:
  \[ E \rightarrow E \cdot T \]
  \[ T \]

- Productions for a nonterminal with the same prefix in two or more right-hand sides, example:
  \[ \text{arglist} \rightarrow ( ) \]
  \[ | ( \text{args} ) \]
  \[ \text{or} \]
  \[ A \rightarrow a \cdot b \]
  \[ | a \cdot c \]

The problem can be solved in most cases by rewriting the grammar to an LL(1) grammar.

Convert a grammar for top-down parsing?

1. Eliminate left recursion
   a) Transform the grammar to iterative form

EBNF (Extended BNF) Notation:

- \( \{ \beta \} \) same as the regular expression: \( \beta^* \)
- \( [ \beta ] \) same as the regular expression: \( \beta | \epsilon \)
- ( ) left factoring, e.g. \( A \rightarrow a \cdot b | a \cdot c \) in EBNF is rewritten:
  \[ A \rightarrow a (b | c) \]

Transform the grammar to be iterative using EBNF

- \( A \rightarrow A \cdot \beta \) \( (\text{where } \beta \text{ may not be preceded by } A) \)
  in EBNF is rewritten:
  \[ A \rightarrow \beta A' \]

2. Left Factoring Using ( ) or [ ]

- \( \text{Original Grammar: } \)
  \[ \text{<stmt> \ } \rightarrow \text{ if <expr> then <stmt> } \]
  \[ \mid \text{ if <expr> then <stmt> else <stmt> } \]

- \( \text{Solution using rewriting: } \)
  \[ \text{<stmt> \ } \rightarrow \text{ if <expr> then <stmt> } \]
  \[ \text{<rest-if> } \]
  \[ \text{<rest-if> \ } \rightarrow \text{ else <stmt> } \mid \epsilon \]

Summary LL(1) and Recursive Descent

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).

Two main methods for writing a top-down parser:

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

<table>
<thead>
<tr>
<th></th>
<th>Recursive Descent</th>
<th>Table-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fast</td>
<td>Hand-written</td>
</tr>
<tr>
<td></td>
<td>Much coding, fast,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Good error management</td>
<td>Easy to include</td>
</tr>
<tr>
<td></td>
<td>and restart</td>
<td>semantic actions,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>good error mgmt</td>
</tr>
</tbody>
</table>

Small Rewriting Grammar Exercise
Example: A recursive Descent Parser for Pascal Declarations, Orig. Grammar

\[
\begin{align*}
\langle \text{declarations} \rangle & \rightarrow \langle \text{constdecl} \rangle \langle \text{vardecl} \rangle \\
\langle \text{constdecl} \rangle & \rightarrow \text{CONST} \langle \text{consdeflist} \rangle \\
\langle \text{constdeflist} \rangle & \rightarrow \langle \text{constdeflist} \rangle \langle \text{constdef} \rangle \\
\langle \text{constdef} \rangle & \rightarrow \text{id} = \text{number} ; \\
\langle \text{vardecl} \rangle & \rightarrow \text{VAR} \langle \text{idlist} \rangle : \langle \text{type} \rangle ; \\
\langle \text{idlist} \rangle & \rightarrow \langle \text{idlist} \rangle , \text{id} \\
\langle \text{type} \rangle & \rightarrow \text{integer} \mid \text{real} \\
\end{align*}
\]

Rewrite in EBNF so that a Recursive Descent Parser can be Written

\[
\begin{align*}
\langle \text{declarations} \rangle & \rightarrow \langle \text{constdecl} \rangle \langle \text{vardecl} \rangle \\
\langle \text{constdecl} \rangle & \rightarrow \text{CONST} \langle \text{constdef} \rangle \{ \langle \text{constdef} \rangle \} \\
\langle \text{constdef} \rangle & \rightarrow \text{id} = \text{number} ; \\
\langle \text{vardecl} \rangle & \rightarrow \text{VAR} \langle \text{idlist} \rangle : \langle \text{type} \rangle ; \\
\langle \text{idlist} \rangle & \rightarrow \langle \text{idlist} \rangle , \text{id} \\
\langle \text{type} \rangle & \rightarrow \text{integer} \mid \text{real} \\
\end{align*}
\]

A Recursive Descent Parser for the New Pascal Declarations Grammar in EBNF

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

```c
void declarations() /*<declarations> \rightarrow <constdecl> <vardecl> */ {
  constdecl();
  vardecl();
  /* declarations */
}

void constdecl() /* <constdecl> \rightarrow CONST <consdeflist> */ {
  if (token == CONST) {
    scan();
    if (token == id)
      constdef();
    else
      error("Missing id after CONST");
    while (token == id) constdef();
  }
}

void vardecl() /* <vardecl> \rightarrow VAR <vardeflist> */ {
  if (token == VAR) {
    scan();
    if (token == ID)
      vardef();
    else
      error("Missing id after VAR");
    while (token == ID) {
      vardef();
    }
  }
}

void vardef() /* <vardef> \rightarrow id { , id } : ( integer | real ) ; */ {
  scan();
  while (token == ',') {
    scan();
    if (token == ID)
      scan();
    else error("id expected after ','");
  }
  if (token == ':') {
    scan();
    if ((token == INTEGER) || (token == REAL))
      scan();
    else error("Incorrect type of variable");
    if (token == ';')
      scan();
    else error("Missing ';' in variable decl.");
  } else error("Missing ';' in var. decl.");
}
```

Pascal Declarations Parser cont 1

```c
void constdef() /* <constdef> \rightarrow <vardecl> */ {
  if (token == CONST) {
    scan();
    if (token == id)
      constdef();
    else
      error("Missing id after const decl");
  }
}
```

Pascal Declarations Parser cont 2

```c
LL Parsing Issues
Beyond Recursive Descent

LL(k)
LL items
Finite pushdown automaton
FIRST and FOLLOW
Table-driven Predictive Parser
```
Automaton Model for Parsing Context-Free Languages

Finite pushdown automaton (FPA)
- A finite automaton with a stack of states
- Context-free grammar G = (N, \Sigma, P, S)
- Integer k \geq 0
- G is (in) LL(k) if:
  - For any two leftmost derivations:
    - S \Rightarrow^*_x u\alpha \Rightarrow^*_x u\gamma \Rightarrow^*_y \delta
    - S \Rightarrow^*_x u\alpha \Rightarrow^*_x u\gamma \Rightarrow^*_y \epsilon
    - with \alpha[1:k] = \delta[1:k]
  - It holds: \beta = \gamma.
- That is, for fixed left context \alpha, the choice for the "right" production to apply to \delta is uniquely determined by the next k input tokens.

Example
- The following grammar is LL(1) (terminals are bold face):
  \[ S \rightarrow \text{id} \text{then} S \text{else} S \text{fi} \]
  \[ | \text{id} \text{do} \text{S} \text{od} \]
  \[ | \text{begin} \text{S} \text{end} \]
  \[ \text{id} \rightarrow \text{id} \]

Grammar with productions
- S \rightarrow aSb | c
- Add new start symbol S': S \rightarrow S
- Transition diagram (showing stack actions below arrows):
  - First tokens of context-free items (states)
  - Transitions in \delta are tuples:
    - (current state, input symbol, stack nonempty, new state, read action, stack action)
  - Grammar G is LL(1) if there exists a finite pushdown automaton recognizing L(G) where \delta is a function (i.e., a deterministic pushdown automaton) and start from [S'\rightarrow S | \#] with empty stack (#).

FIRST and FOLLOW
- For a sentential form \alpha in (N \cup S)^*.
  - FIRST(\alpha) denotes the set of all terminals with can be first in a string derived from \alpha.
  - For a nonterminal A in N,
    - FOLLOW(A) denotes the set of all terminals (e.g., a) that could appear immediately after A in a sentential form i.e., there exists S \Rightarrow^* aAa\beta for arbitrary \alpha, \beta.
Computing FIRST = FIRST₁
For all grammar symbols X:
- If X is a terminal, then FIRST(X) = {X}.
- If X = ε is a production, then add ε to FIRST(X).
- If X is a nonterminal and X \rightarrow Y₁ Y₂ ... Yᵦ is a production,
  then place all those a of S in FIRST(X) where for some i, a is in FIRST(Yᵢ).
  and ε is in all of FIRST(Yᵢ), ... FIRST(Yᵦ).
- If ε is in FIRST(Yᵢ) for all i, 1 ≤ i ≤担负，则 add ε to FIRST(X).

For the example grammar

\[ S' \rightarrow S; \ S \rightarrow aSb; \ S \rightarrow c \]
\[ \text{FIRST}(a) = \{a\}, \text{FIRST}(b) = \{b\}, \text{FIRST}(c) = \{c\} \]
\[ \text{FIRST}(S') = \text{FIRST}(S) \]
\[ \text{FIRST}(S) = \{a, c\} \]

Computing FOLLOW
Compute FOLLOW(B) for each nonterminal B:
- Add \( \rightarrow \) to FOLLOW(S).
- If there is a production A \( \rightarrow \alpha \) B \( \beta \) for arbitrary \( \alpha, \beta \) then add all of FIRST(\( \beta \)) except ε to FOLLOW(B).
- If there is a production A \( \rightarrow \alpha B \) or a production \( A \rightarrow \epsilon B \) where ε is in FIRST(\( \beta \)), i.e. \( \beta \rightarrow \epsilon \), then add all of FOLLOW(A) to FOLLOW(B).

For the example grammar

\[ S' \rightarrow S, \ S \rightarrow aSb, \ S \rightarrow c \]
\[ \text{FOLLOW}(S) = \{\epsilon, b\} \]

Example Cont.: Finite Pushdown Automaton (FPA) Made Deterministic
- Grammar with productions \( S \rightarrow aSb \text{ | c} \)
- Added new start symbol \( S' \: \{S' \rightarrow S \rightarrow aSb, \ S \rightarrow c\} \)

Disambiguated:
FIRST(\( a \text{ | b} \)) = \{a\}
FIRST(c) = \{c\}

Example (cont.): Transition table (k=1)
<table>
<thead>
<tr>
<th>state</th>
<th>final</th>
<th>lookahead</th>
<th>lookahead</th>
<th>lookahead</th>
<th>lookahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' \rightarrow S \rightarrow \text{</td>
<td>a} )</td>
<td>no</td>
<td>push ( S' \rightarrow S \rightarrow \text{</td>
<td>b} )</td>
<td>[Error]</td>
</tr>
<tr>
<td>( S \rightarrow S \rightarrow \text{</td>
<td>a} )</td>
<td>no</td>
<td>Error</td>
<td>Error</td>
<td>[Error]</td>
</tr>
<tr>
<td>( S \rightarrow S \rightarrow \text{</td>
<td>b} )</td>
<td>no</td>
<td>[Error]</td>
<td>[Error]</td>
<td>[Error]</td>
</tr>
<tr>
<td>( S \rightarrow S \rightarrow \text{</td>
<td>a} )</td>
<td>no</td>
<td>pop state</td>
<td>Error</td>
<td>pop state</td>
</tr>
<tr>
<td>( S \rightarrow S \rightarrow \text{</td>
<td>b} )</td>
<td>no</td>
<td>pop state</td>
<td>Error</td>
<td>pop state</td>
</tr>
<tr>
<td>( S \rightarrow S \rightarrow \text{</td>
<td>c} )</td>
<td>no</td>
<td>pop state</td>
<td>Error</td>
<td>pop state</td>
</tr>
</tbody>
</table>
**General Approach: Predictive Parsing**

At any production $A \rightarrow \alpha$
- If $\epsilon$ is not in $\text{FIRST}(\alpha)$:
  - Parser expands by production $A \rightarrow \alpha$
    if current lookahead input symbol is in $\text{FIRST}(\alpha)$.
- Otherwise (i.e., $\epsilon$ in $\text{FIRST}(\alpha)$):
  - Expand by production $A \rightarrow \alpha$
    if current lookahead symbol is in $\text{FOLLOW}(A)$
    or if it is $\rightarrow$ and $\rightarrow$ is in $\text{FOLLOW}(A)$.

Use these rules to fill the transition table.
(pseudocode: see [ASU86] p. 190, [ALSU06] p. 224)

**Summary: Parsing LL(k) Languages**

- **Predictive LL parser**
  - Iterative, based on finite pushdown automaton
  - Transition-table-driven
  - Can be generated automatically
- **Recursive-descent parser**
  - Recursive
  - Manually coded
  - Easier to fix intermediate code generation, error handling
- **Both require lookahead** (or backtracking)
  - To predict the next production to apply
  - Removes nondeterminism
  - Necessary checks derived from FIRST and FOLLOW sets
  - FIRST and FOLLOW are also useful for syntax error recovery

**Homework**

- Now, read again the part on recursive descent parsers and find the equivalent of
  - Context-free items (Pushdown automaton (PDA) states)
  - The stack of states
  - Pushing a state to stack
  - Popping a state from stack
  - Start state, final state
  - In a recursive descent parser.