Top-down Parsing ("Recursive Descent")

- "top-down": starting with the start symbol of the grammar, tries to derive the input string. This is done by using the next input symbol(s) and the current state of the parser to properly "guess" the next derivation step.

- Implemented in terms of transitions on a transition diagram:
  - There is a procedure corresponding to each non-terminal: this procedure is responsible for parsing strings derivable from that nonterminal.
  - A transition on a *token* (i.e., terminal) means this transition should be taken if the next input symbol matches it.
  - A transition on a *nonterminal* is a call to the procedure for that nonterminal.

Example:

Grammar fragment:

\[ Stmt \rightarrow \text{if } Expr \text{ then } Stmt \text{ else } Stmt \]

Transition Diagram:

![Transition Diagram](image)

Corresponding Code Fragment:

```java
Stmt()
{
  match(IF);
  Expr();
  match(THEN);
  Stmt();
  match(ELSE);
  Stmt();
}
```
Pitfalls in Top-down Parsing

- **Ambiguity**: more than one production looks applicable in a derivation, don't know which one to use.

- **Left Recursion**: productions of the form
  \[ E \rightarrow E + E \]
  Causes the parser to loop.

- **Backtracking**: might have to "back up", e.g. with
  \[ A \rightarrow aB | aC \]

---

Eliminating Ambiguity: Left Factoring

**Basic Idea**: Delay the decision about which parse tree to use, by "factoring out" any common prefix between the productions for a nonterminal.

**Example**: Given the productions

\[
S \rightarrow \text{if } E \text{ then } S \\
S \rightarrow \text{if } E \text{ then } S \text{ else } S
\]

Left factoring yields

\[
S \rightarrow \text{if } E \text{ then } S \ S' \\
S' \rightarrow \text{else } S \mid \epsilon
\]
Now add the *disambiguating rule*:

```
if (next token == `else`) take `else`-arc;
else take `ε`-arc.
```

**Remark**: This solution effectively associates each `else` with the closest `then`.

---

**Eliminating Left Recursion**

This refers to productions of the form

```
A → Aα
```

e.g., \( E → E + E \).

**Problem**: causes top-down parsers to loop.

**Solution**: Transform left recursion to right recursion.
**Basic Idea**: Given productions

\[ A \rightarrow A\alpha | \beta \]

we derive strings in the set

\[ \{ \beta\alpha^n, \beta\alpha\alpha^n, \beta\alpha\alpha\alpha^n, \ldots \} \]

where \( \beta\alpha^n \) corresponds to \( n \) times around the loop.

In general, after \( n \) times around the loop we get \( \beta\alpha^n \).

So: change the structure of the loop:

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' | \varepsilon \]

<table>
<thead>
<tr>
<th>Orig. Grammar</th>
<th>LL(1) Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + T \mid T )</td>
<td>( E \rightarrow TE' )</td>
</tr>
<tr>
<td>( E' \rightarrow + TE' \mid \varepsilon )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow T^* F \mid F )</td>
<td>( T \rightarrow FT' )</td>
</tr>
<tr>
<td>( T' \rightarrow F T' \mid \varepsilon )</td>
<td></td>
</tr>
<tr>
<td>( F \rightarrow (E) \mid id )</td>
<td>( F \rightarrow (E) \mid id )</td>
</tr>
</tbody>
</table>
**Eliminating Left Recursion**: cont’d

In general: given the productions

\[ A \rightarrow A\alpha_1 | \cdots | A\alpha_m | \beta_1 | \cdots | \beta_n \]

where no \( \beta_i \) begins with \( A \): transform this to

\[ A \rightarrow \beta_1 A' | \cdots | \beta_n A' \]
\[ A' \rightarrow \alpha_1 A' | \cdots | \alpha_m A' | \epsilon \]

**Note**: This does not remove *indirect* left recursion, e.g.:

\[ A \rightarrow Ba \]
\[ B \rightarrow Ab \]

---

**Backtracking**

**Problem**: If the RHS of two productions for the *same* nonterminal have a common prefix, then we don’t know which production to use.

**Example**: \( A \rightarrow cdA | ceB \)

**Immediate Solution**: left factoring.

Unfortunately, left factoring has no “look-ahead”, and so may not always work, e.g.:

\[ A \rightarrow aB | C \]
\[ C \rightarrow a \]
Alternative Possibility: Try one production to see if it works. "Remember" the alternative, so that the parser can backtrack and try the other production if the first one gets "stuck".

While possible, this is inefficient, so we do not consider it further.

Solution: Disallow the possibility of backtracking.

Note: This means that it should be possible to process a grammar beforehand and detect any possibility of backtracking. This can be done by computing FIRST and FOLLOW sets.

LL(1) Grammars

Intuitively: a grammar that can be handled by a recursive descent parser without backtracking.

LL(1):

- Input scanned Left-to-right.
- Produces a Leftmost derivation.
- Uses 1-token lookahead.
**Definition**: A grammar is LL(1) if and only if for any two distinct productions

\[ A \rightarrow \alpha | \beta \]

the following hold:

1. \( \text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset. \)
2. If \( \beta \Rightarrow^* \varepsilon \) then \( \text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset. \)

*This includes both terminals and \( \varepsilon \).

---

**What if the grammar is not LL(1)?**

1. Eliminate left recursion, then left factor if necessary.

2. Try a different parsing algorithm, e.g. LALR or LR(1).
Implementing a Recursive Descent Parser

1. Implement a function `gettoken()` that returns the next token, and associated attributes (if any).

2. Using the productions of the grammar, construct a transition diagram for each non-terminal:
   - create initial and final (i.e., return) states;
   - for each production $A \rightarrow X_1 \ldots X_n$ create a path from the initial to the final state with edges $X_1, \ldots, X_n$.

3. Simplify these transition diagrams if possible.

4. For each transition diagram, write a procedure that “traverses” the diagram guided by the input.

---

LL(1) Parsing: Traversing Transition Diagrams:

Suppose the code to “traverse” an edge labelled $s$ is $T(s)$, then:

**case 1:** If $s$ is a terminal then $T(s)$ matches the next input token with $s$ and advances the input (i.e., gets next token).
   - If the input token does not match $s$ then syntax error.

**case 2:** if $s$ is a non-terminal, then $T(s)$ is a procedure call to $s()$. 

---
case 3:

\[
\begin{align*}
\text{begin} \\
T(s_1); \\
\vdots \\
T(s_n); \\
\text{end}
\end{align*}
\]

\[
\begin{array}{c}
\rightarrow \quad s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n \rightarrow \circ
\end{array}
\]

case 4:

\[
\begin{align*}
\text{if curr_tok} \in \text{FIRST}(s_1) \text{ then } T(s_1); \\
\text{else if curr_tok} \in \text{FIRST}(s_2) \text{ then } T(s_2); \\
\vdots \\
\text{else if curr_tok} \in \text{FIRST}(s_n) \text{ then } T(s_n); \\
\text{else syntax error;}
\end{align*}
\]

[or: use equivalent case/switch statement.]
case 5:

\[ \textbf{while} \ \text{curr}\_\text{tok} \in \text{FIRST}(s) \ \textbf{do} \]
\[ T(s); \]

---

**Recursive Descent Parsing: Example**

**Production:** \[ E \rightarrow T \ E' \] : 
\[ E() \]
\[ \{ \]
\[ T(); \]
\[ E'(); \]
\[ \} \]
**Production:** \( E' \rightarrow + T E' \mid \varepsilon \)

\[
E'()
\{
\text{currtok = nexttok();}
\text{if (currtok == '+' \{ /* E' \rightarrow + T E' */}
\text{match('+');}
\text{T();}
\text{E'();}
\}
\text{else if (currtok \in FOLLOW(E')) /* E' \rightarrow \varepsilon */}
\text{return;}
\text{else}
\text{error;}
\}
\]

---

**Table Driven LL(1) Parser**

1. Avoid the cost of function calls by maintaining an explicit stack.

2. Use a parse table to drive prediction of production to be applied.
Constructing the Parse Table

Parse Table:
\[ M[N, T] \rightarrow Production \ Rule, \]
where \( N \) is non-terminal and \( T \) is a terminal.

For each production \( A \rightarrow \alpha \) of the grammar

1. for each terminal \( a \in FIRST(\alpha) \)
   add \( A \rightarrow a \) to \( M[A, a] \).

2. if \( \varepsilon \in FIRST(\alpha) \) then
   for each terminal \( b \in FOLLOW(A) \)
   add \( A \rightarrow \alpha \) to \( M[A, b] \).

   if \( \varepsilon \in FIRST(\alpha) \) and \( \$ \in FOLLOW(A) \) then
   add \( A \rightarrow \alpha \) to \( M[A, \$] \).

Parse Driver

Initialize stack to \($S$, where \( S \) is the start symbol and \( $ \) marks the end of input.

The parse driver takes actions based upon \( X \) (the symbol at the top of the syntax stack) and \( a \) (the next input token) as follows:

1. if \( X = a = \$ \) then SUCCESS.

2. if \( X = a \neq \$ \) then
   Pop \( X \) from the syntax stack;
   Advance input pointer;
3. if \( X \neq a \) then
   1. if \( X \) is a terminal then ERROR
   2. else /* \( X \) is a non-terminal */
      1. if \( M[X, a] = \text{Null} \) then ERROR
      2. else let \( M[X, a] = X \rightarrow ABC \)
         1. Pop \( X \) from the syntax stack;
         2. Push \( ABC \) on the syntax stack in reverse order;

---

**Example**

**LL(1) Grammar**

<table>
<thead>
<tr>
<th>Production</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow T E' )</td>
<td>( E \rightarrow + T E' \mid \varepsilon )</td>
<td>( E \rightarrow ( E ) \mid \text{id} )</td>
</tr>
<tr>
<td>( T \rightarrow F T' )</td>
<td>( T \rightarrow * F T' \mid \varepsilon )</td>
<td>( F \rightarrow ( E ) \mid \text{id} )</td>
</tr>
<tr>
<td>( E' \rightarrow + T E' \mid \varepsilon )</td>
<td>( E' \rightarrow +, \varepsilon )</td>
<td>( E' \rightarrow (, \text{id} } )</td>
</tr>
<tr>
<td>( F \rightarrow ( E ) \mid \text{id} )</td>
<td>( *, \varepsilon )</td>
<td>( F \rightarrow *, +, $ )</td>
</tr>
</tbody>
</table>
### Grammar Table

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>E' → + TE'</td>
<td>E' → ε</td>
<td>E' → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → FT'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T' → ε</td>
<td>T' → FT'</td>
<td>T' → ε</td>
<td>T' → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F → id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack, Input, Action

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id+id$</td>
<td>E → T E'</td>
</tr>
<tr>
<td>$E'T$</td>
<td>id+id+id$</td>
<td>T → F T'</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>id+id+id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id+id+id$</td>
<td>pop</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>+id+id$</td>
<td>T' → ε</td>
</tr>
<tr>
<td>$E'$</td>
<td>+id+id$</td>
<td>E' → + T E'</td>
</tr>
<tr>
<td>$E'T+$</td>
<td>+id+id$</td>
<td>pop</td>
</tr>
<tr>
<td>$E'T$</td>
<td>id+id$</td>
<td>T → F T'</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>id+id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id+id$</td>
<td>pop</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>*id$</td>
<td>T' → * F T'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'TF'$</td>
<td>*id$</td>
<td>pop</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T$id</td>
<td>id$</td>
<td>pop</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>T' → ε</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>E' → ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>