Top-Down Parsing

The parse tree is constructed
- From the top
- From left to right.
- The terminals are seen in order of appearance in the token stream (t2, t5, t6, t8, t9):

```
1
7
4
3

t2

3

t5

4

t6

5

7

6

8

9
```

Building The LL(1) Parser

1. Remove left recursion.
2. Left factor the grammar.
3. Construct transition diagrams for each production:
   - IF Expr() THEN Stat() ELSE Stat() -> ...
   - ELSE Stat() -> ...
   - IF Expr() -> ...

1. Compute FIRST(A) for each grammar symbol A.
2. Compute FOLLOW(A) for each nonterminal A.
3. Simplify the transition diagrams.
4. Construct the recursive procedures.

FIRST Sets

- FIRST(α) is the set of terminals that begin strings derived from α.
- FIRST-sets help us solve problem 3.

prog → decl | stat
stat → if ... | id() | while ...
decl → int id | real id

FIRST(stat) = \{if, id, while\}
FIRST(decl) = \{int, real\}
FIRST Sets...

PROCEDURE prog();
    IF curr_tok \in \{if, id, while\} THEN stat();
    ELSIF curr_tok \in \{int, real\} THEN decl();
    ELSE syntax error ENDIF;
END;
PROCEDURE stat();
    · · ·
END;
PROCEDURE decl();
    · · ·
END;
FIRST(stat) = \{if, id, while\}
FIRST(decl) = \{int, real\}

FIRST(α) is the set of terminals that begin strings derived from α, i.e. FIRST(α) = \{a | a a terminal, α \to aβ\}.

Example Grammar

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>\to</td>
<td>T E'</td>
</tr>
<tr>
<td>E'</td>
<td>\to</td>
<td>\pm T E'</td>
</tr>
<tr>
<td>T</td>
<td>\to</td>
<td>F T'</td>
</tr>
<tr>
<td>T'</td>
<td>\to</td>
<td>\ast F T'</td>
</tr>
<tr>
<td>F</td>
<td>\to</td>
<td>(E)</td>
</tr>
</tbody>
</table>

FIRST Sets:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST(E) = {\epsilon, id}</td>
<td>FIRST(E') = {\pm, \epsilon}</td>
<td></td>
</tr>
<tr>
<td>FIRST(T) = {\epsilon, id}</td>
<td>FIRST(T') = {\ast, \epsilon}</td>
<td></td>
</tr>
<tr>
<td>FIRST(F) = {\epsilon, id}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Computing FIRST(A)

REPEAT until no more changes:
1. IF A is a terminal THEN FIRST(A) = \{A\}.
2. IF A is a nonterminal, and there is a production \(A \to \epsilon\) THEN \(\epsilon\) is in FIRST(A).
3. IF A is a nonterminal, and there is a production \(A \to Y_1 \cdots Y_k\) THEN
   FOR \(i := 1\) to \(k - 1\) DO
     IF \(\epsilon \in FIRST(Y_i) \land \cdots \land \epsilon \in FIRST(Y_i)\)
       and \(a (\neq \epsilon) \in FIRST(Y_{i+1})\) THEN \(a\) is in FIRST(A);
     IF \(\epsilon \in FIRST(Y_1) \land \cdots \land \epsilon \in FIRST(Y_k)\) THEN \(\epsilon\) is in FIRST(A);
FOLLOW Sets

- We let $ symbolize end-of-input.
- FOLLOW(A) is the set of terminals that can follow right after the nonterminal A in some sentential form. 
  FOLLOW(A) = \{a | a a terminal, S \Rightarrow Aaβ\}.
- $ ∈ FOLLOW(A) if A is the rightmost symbol in a sentential form, i.e. S \Rightarrow αA.

\[
\begin{array}{|c|c|c|}
\hline
E & T E' & F \\
\hline
E' & \pm T E' | \epsilon & \star F T' | \epsilon \\
\hline
F & \langle E \rangle | \text{id} & \\
\hline
\end{array}
\]

- $ is in FOLLOW(E), because

  \[E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \langle E \rangle T'E'.\]

- $ is in FOLLOW(T), because

  \[E \Rightarrow TE' \Rightarrow \pm TE' \Rightarrow \langle E \rangle T'E'.\]

\[
\begin{array}{|c|c|c|}
\hline
E & T E' & F T' \\
\hline
E' & \pm T E' | \epsilon & \star F T' | \epsilon \\
\hline
F & \langle E \rangle | \text{id} & \\
\hline
\end{array}
\]

- $ is in FOLLOW(F), because

  \[E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \langle E \rangle FT'E'.\]

- $ is in FOLLOW(E'), because

  \[E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \langle E \rangle T'E' \Rightarrow
  \langle TE' \rangle T'E' \Rightarrow \langle FT'E' \rangle T'E' \Rightarrow
  \langle FE' \rangle T'E' \Rightarrow \langle F \rangle T'E'.\]
Computing FOLLOW Sets

Let $S$ be the start symbol and $\$\$ the end-of-file marker.

REPEAT until no more changes:
1. Add $\$\$ to FOLLOW($S$).

2. IF there is a production $A \rightarrow \alpha B \beta$ THEN
   Add everything in FIRST($\beta$) (except $\epsilon$) to FOLLOW($B$).

3. IF there is a production $A \rightarrow \alpha B$ OR
   a production $A \rightarrow \alpha B \beta$ where $\epsilon \in$ FIRST($\beta$) THEN
   Add everything in FOLLOW($A$) to FOLLOW($B$).

LL(1) Grammars

A grammar is LL(1) if we can construct a recursive descent parser that handles it (without using backtracking).

LL(1) stands for
- The input is scanned from left-to-right.
- The parse produces a leftmost derivation.
- We have 1-token lookahead.

Formal Definition:

A grammar is LL(1) iff for any two productions

$$ A \rightarrow \alpha | \beta $$

the following conditions hold

1. $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
2. If $\beta \Rightarrow \epsilon$ then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$
Recursive Descent Parsers

FOR each non-terminal A DO
create initial and final states;

FOR each production A → X1 · · · Xn DO
create a path A's initial to A's final node with edges labeled X1 · · · Xn:
Simplify the diagrams;

FOR each transition diagram P DO
Create a procedure P that 'traverses' the diagram guided by the input.

PROCEDURE P();
IF curr_tok = a THEN curr_tok := next_token()
ELSE syntax_error; ENDIF
IF curr_tok ∈ FIRST(s1) THEN code for parsing s1
ELSIF curr_tok ∈ FIRST(s2) THEN code for parsing s2
ELSE syntax_error; ENDIF
B();
WHILE curr_tok ∈ FIRST(r ) DO code for parsing r ENDDO
END P;

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' | \epsilon \\
T & \rightarrow F T' \\
T' & \rightarrow * F T' | \epsilon \\
F & \rightarrow ( E ) | id \\
\end{align*}
\]
PROCEDURE E();
LOOP
T();
IF cur_tok = + THEN
cur_tok := next_token()
ELSE EXIT ENDIF
ELSE EXIT ENDIF
ENDLOOP;
PROCEDURE T();
LOOP
F();
IF cur_tok = * THEN
    cur_tok := next token()
ELSE EXIT ENDIF
ENDLOOP;

PROCEDURE F();
IF cur_tok = ( THEN
    cur_tok := next_token();
    E();
    IF cur_tok = ) THEN
        cur_tok := next_token()
    ELSE syntax error ENDIF
ELSIF cur_tok = id THEN
    cur_tok := next_token();
ELSE syntax error ENDIF

static void E() throws Exception {
    T(); while(lookahead("+")) {match("+"); T();}
}
static void T() throws Exception {
    F(); while(lookahead("*")) {match("*"), F();}
}
static void F() throws Exception {
    if (lookahead("(")) {match("("); E(); match(")");
    } else match("id");
}

public static void main(String[] args) {
    try {E(); System.out.println("true");
    } catch (Exception e)
    {System.out.println("false");}
}
• Read Louden, pp. 143–196.

• Or, the Dragon Book:
  - Top-Down Parsing 181–190
  - Error Recovery 192–195
  - Recursive Descent Parsing 40–55, 75–76