Top-Down Parsing

Non-terminal as a Function

In a top-down parser a non-terminal may be viewed as a procedure matching a portion of the input. In our expression grammar G with the *E-productions* $E \rightarrow E + T$, $E \rightarrow E - T$, $E \rightarrow T$, the function looks as follows:

Function of E

```
E()
    Select an $E$-production $p$
            // here is the nondeterminism
    **if** $p = E \rightarrow E + T$
        call $E()$
        **if** yylex()='+'
            call $T()$
        **else** *error*
    **else if** $\cdots$
```

Example

Let the input be `ic`$\cdots$. The parser starts with the start symbol E and chooses the production rule $E \rightarrow E + T$. But then there is no change in the input and the leftmost non-terminal E may be expanded again and again *ad infinitum*.
A left recursive grammar may lead to non-termination.

Example

In the previous example let the parser starts with the start symbol E and chooses the following sequence of production rules: $E \rightarrow T$, $T \rightarrow F$ and $F \rightarrow \texttt{ic}$. The first symbol of the input matches, but the choice may be incorrect if the next input symbol is '`+`', as there is no rule with right hand side $F + \cdots$.
It may be necessary to backtrack on the choice of production rules.

Example

Consider the grammar:

$$S \rightarrow aSa \mid aTba \mid c$$

$$T \rightarrow bS$$

If the input is `a`$\cdots$, we cannot decide whether to use the first or the second production rule of S. But if the parser is designed to *look-ahead* another symbol (*2-look-ahead*), the correct choice can be made. If the input is `aa`$\cdots$, the selected rule for derivation is $S \rightarrow aSa$. But if it is `ab`$\cdots$, the choice is $S \rightarrow aTba$.

Note

In case of the expression grammar G, no fixed amount of *look-ahead* can help. We may have *5-look-ahead* and the input is `ic+ic+ic`$\cdots$. The derivation sequence will be $E \rightarrow E + T \rightarrow E + T + T$. But the next step is not known as the operator after the rightmost `ic` is not known. Note that no *token* has been consumed (read) so far.

Example

Consider the ambiguous grammar:

$$\begin{aligned} S &\rightarrow aSa \mid bSb \mid aTba \mid c \\ T &\rightarrow bS \end{aligned}$$

There is no way to decide a rule entirely on the basis of the input, without removing the ambiguity.

$LL(k)$

An unambiguous context-free grammar without left recursion is called an $LL(k)$ grammar[a], if a top-down predictive parser for its language can be constructed with at most k input look-ahead. We shall consider the case of $k = 1$.

[a] The parser scans the input from left-to-right and uses the leftmost derivation.

FIRST(X)

Informally, the FIRST() set of a terminal or a non-terminal X or a string over terminals and non-terminals, is the collection of all terminals (also ε) that can be derive from X, in the grammar, as the first (leftmost) terminal symbol.

FIRST(X)

If $X \in \Sigma \cup N \cup \{\varepsilon\}$, then FIRST$(X) \subseteq \Sigma \cup \{\varepsilon\}$ is defined inductively as follows:

- FIRST$(X) = \{X\}$, if $X \in \Sigma \cup \{\varepsilon\}$,
- FIRST(X) is $\cup_{X \to \alpha \in P}$ FIRST(α), $X \in N$,
- $\varepsilon \in$ FIRST(X), if there is $X \to \alpha$ and $\alpha \to \varepsilon$,
- FIRST$(A \to \alpha)$ is the FIRST(α).

$\boxed{\textbf{FIRST}(X)}$

If $\alpha = X_1X_2\cdots X_k$, then $\text{FIRST}(X_1) \setminus \{\varepsilon\} \subseteq \text{FIRST}(\alpha)$ and $\text{FIRST}(X_i) \setminus \{\varepsilon\} \subseteq \text{FIRST}(\alpha)$ when $\varepsilon \in \bigcap_{j=1}^{i-1} \text{FIRST}(X_j)$, $1 < i \leq k$.
If $\varepsilon \in \bigcap_{j=1}^{k} \text{FIRST}(X_j)$, then $\varepsilon \in \text{FIRST}(\alpha)$.

Example

Consider the classic expression grammar;
FIRST(E) =FIRST(T) =FIRST(F) $= \{$`ic`$, (\}$.
There are two production rules for each of E and T with the identical FIRST() sets:
$E \to E + T,\ E \to T$ and $T \to T * F,\ T \to F$

Example

Consider the grammar obtained after removing the left-recursion from G:

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \varepsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \varepsilon \\ F & \rightarrow & (E) \mid \texttt{ic} \end{array}$$

Example

FIRST(E) =FIRST(T) =FIRST(F) $= \{$`ic`$, (\}$, FIRST(E') $= \{+, \varepsilon\}$, and FIRST(T') $= \{*, \varepsilon\}$.

No non-terminal has more than one production rule with the identical FIRST() set.

FOLLOW(X)

For every non-terminal X, the FOLLOW() set is the collection of all terminals that can follow X in a *sentential form*. The set can be defined inductively as follows.

- The special symbol *eof* or \$ is in FOLLOW(S), where S is the start symbol.
- If $A \rightarrow \alpha B \beta$ be a production rule, FIRST$(\beta) \setminus \{\varepsilon\} \subseteq$ FOLLOW(B).

FOLLOW(X)

- If $A \to \alpha B\beta$, where $\beta = \varepsilon$ or $\beta \to \varepsilon$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$.

The reason is simple:
$S \to uAv \to u\alpha B\beta v \to u\alpha Bv$, naturally
$\text{FIRST}(v) \subseteq \text{FOLLOW}(A), \text{FOLLOW}(B)$.

Computation of FOLLOW() Sets

```
for each $A \in N$
    FOLLOW$(A) \leftarrow \emptyset$
FOLLOW$(S) \leftarrow \{\$\}$
while (FOLLOW sets are not fixed points)
    for each $A \rightarrow \beta_1\beta_2 \cdots \beta_k \in P$
    if $(\beta_k \in N)$
        FOLLOW$(\beta_k) \leftarrow$ FOLLOW$(\beta_k) \cup$ FOLLOW$(A)$
    $FA \leftarrow$ FOLLOW$(A)$
```

Computation of FOLLOW() Sets

```
for $i \leftarrow k$ downto $2$
    if $(\beta_i \in N \ \& \ \varepsilon \in \text{FOLLOW}(\beta_i))$
        $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1})$
            $\cup \text{ FIRST}(\beta_i) \setminus \{\varepsilon\} \cup FA$
    else
        $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1})$
            $\cup \text{ FIRST}(\beta_i) \setminus \{\varepsilon\}$
        $FA \leftarrow \emptyset$
```

Example

In the expression grammar G:
$\text{FOLLOW}(E) = \{\$, +,)\}$, $\text{FOLLOW}(T) = \text{FOLLOW}(E) \cup \{*\} = \{\$, +,), *\}$ and $\text{FOLLOW}(F) = \{\$, +,), *\}$.

In the transformed grammar:
$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\$,)\}$,
$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\$,), +\}$ and
$\text{FOLLOW}(F) = \{\$,), +, *\}$.

$LL(1)$ Grammar

A context-free grammar G is $LL(1)$ *iff* for any pair of distinct productions $A \rightarrow \alpha$, $A \rightarrow \beta$, the following conditions are satisfied.

- FIRST(α)$\cap$ FIRST(β) $= \emptyset$ i.e. no $a \in \Sigma \cup \{\varepsilon\}$ can belong to both.
- If $\alpha \rightarrow \varepsilon$ or $\alpha = \varepsilon$, then FIRST(β)$\cap$ FOLLOW(A) $= \emptyset$.

Example

Consider the following grammar with the set of *terminals*,
$\Sigma =$ {`id ; := int float main do else end if print scan then while`} $\cup${E BE}[a];
the set of *non-terminals*,
$N =$ {`P DL D VL T SL S ES IS WS IOS`};
the start symbol is `P` and the set of production rules are:

[a] `E` and `BE`, corresponds to expression and boolean expressions, are actually *non-terminals*. But here we treat them as terminals.

Production Rules

```
1  P   →  main DL SL end
2  DL  →  D DL | D
4  D   →  T VL ;
5  VL  →  id VL | id
7  T   →  int | float
9  SL  →  S SL | ε
11 S   →  ES | IS | WS | IOS
```

Production Rules

```
15 ES  → id := E ;
16 IS  → if BE then SL end |
          if BE then SL else SL end
18 WS  → while BE do SL end
19 IOS → scan id ; | print E ;
```

Note

There is no production rule with *left-recursion*. But the rules 2,3, 5,6, and 16,17 needs *left-factoring* as the FIRST() sets are not disjoint. The transformed grammar after factoring is:

New Production Rules

1	`P`	$\rightarrow$	`main DL SL end`
2	`DL`	$\rightarrow$	`D DO`
3	`DO`	$\rightarrow$	`DL` | ε
4	`D`	$\rightarrow$	`T VL ;`
5	`VL`	$\rightarrow$	`id VO`
6	`VO`	$\rightarrow$	`VL` | ε
7	`T`	$\rightarrow$	`int` | `float`

Production Rules

9	`SL`	$\rightarrow$	`S SL` | ε
11	`S`	$\rightarrow$	`ES` | `IS` | `WS` | `IOS`
15	`ES`	$\rightarrow$	`id := E ;`
16	`IS`	$\rightarrow$	`if BE then SL EO`
17	`EO`	$\rightarrow$	`end` | `else SL end`
18	`WS`	$\rightarrow$	`while BE do SL end`
19	`IOS`	$\rightarrow$	`scan id ;` | `print E ;`

FIRST()

The next step is to calculate the FIRST() sets of different rules.

NT/Rule	FIRST()
`P` (1)	`main`
`DL` (2)	`int float`
`DO` (3)	`int float`
`DO` (3a)	ε
`D` (4)	`int float`

FIRST()

NT/Rule	FIRST()
`VL` (5)	`id`
`VO` (6)	`id`
`VO` (6a)	ε
`T` (7)	`int`
`T` (8)	`float`
`SL` (9)	`id if while scan print`

FIRST()

NT/Rule	FIRST()
`SL` (10)	ε
`S` (11)	`id`
`S` (12)	`if`
`S` (13)	`while`
`S` (14)	`scan print`

FIRST()

NT/Rule	FIRST()
`ES` (15)	`id`
`IS` (16)	`if`
`EO` (17)	`end`
`EO` (17a)	`else`

FIRST()

NT/Rule	FIRST()
`WS` (18)	`while`
`IOS` (19)	`scan`
`IOS` (20)	`print`

Note

Three rules have ε-productions. Their applications in a predictive parser depends on what can follow the corresponding non-terminals. So it is necessary to compute the FOLLOW() sets corresponding to these non-terminals. The rules are:
`DO` $\rightarrow \varepsilon(3a)$, `VO` $\rightarrow \varepsilon(6a)$, `SL` $\rightarrow \varepsilon(10)$.

FOLLOW()

NT	FOLLOW()
`DO`	`id if while scan print end`
`VO`	`;`
`SL`	`end else`

Note

$FOLLOW(DO) = FOLLOW(DL)$ (rule 2). The $FOLLOW(DL) = FIRST(SL) \setminus \{\varepsilon\} \cup FOLLOW(P)$ (rule 1) as SL is *nullable* (rule 10). Now $FOLLOW(P) = \{\texttt{end}\}$.

Note

It is clear from the previous computation that no two production rules of the form $A \rightarrow \alpha_1 \mid \alpha_2$ have common elements in their FIRST() sets. There is also no common element in the FIRST() set of the production rule $A \rightarrow \alpha$ and the FOLLOW() set of A in cases where $A \rightarrow \varepsilon$. So the grammar is $LL(1)$ and a predictive parser can be constructed.

Recursive-Descent Parser

We write a function (may be recursive) for every non-terminal. The function corresponding to a non-terminal A returns $ACCEPT$ if the corresponding portion of the input can be generated by A. Otherwise it returns a $REJECT$ with proper error message.

Example

Consider the production rule

$$\boxed{\texttt{P} \rightarrow \texttt{main DL SL end}}$$

The function corresponding to the non-terminal `P` is as follows:

int P()

```
int P(){
   if(yylex() == MAIN){ // MAIN for "main"
      nextToken = NOTOK;
      if(DL() == ACCEPT)
         if(SL() == ACCEPT) {
            if(nextToken == NOTOK)
               nextToken = yylex();
            if(nextToken == END) // END is the token
               return ACCEPT;        // for "END"
            else {
                 printf("end missing (1)\n");
                 return REJECT;
```

```
                    }
                }
                else {
                        printf("SL mismatch (1)\n");
                        return REJECT;
                }
            else {
                    printf("DL mismatch (1)\n");
                    return REJECT;
            }
        }
        else {
            printf("main missing (1)\n");
            return REJECT;
        }
    }
```

Note

The *global variable* `nextToken` stores the *look-ahead* input (token). If there is a valid `nextToken`, it is to be consumed before calling `yylex()`.
The stack of the push-down automaton is the stack of the recursive call. The body of the function corresponding to a non-terminal corresponds to all its production rules.

Example

We now consider a non-terminal with ε-production.

$$\boxed{\texttt{DO} \rightarrow \texttt{DL} \mid \varepsilon}$$

The members of FIRST(`DL`) are `{int float}` and the elements of FOLLOW(`DO`) are `{id if while scan print end}`.

int DO()

```
int FDO(){
    if(nextToken == NOTOK)
       nextToken = yylex();
    if(nextToken == INT ||
       nextToken == FLOAT)
       if(DL() == ACCEPT) return ACCEPT;
       else {
           printf("DL mismatch (3)\n");
           return REJECT;
       }
    else
       if(nextToken == IDNTIFIER ||
          nextToken == IF ||
          nextToken == WHILE ||
          nextToken == SCAN ||
          nextToken == PRINT ||
          nextToken == END)
          return ACCEPT;
        else {
          printf("DO follow mismatch (3)\n");
          return REJECT;
        }
    }
}
```

Note

The global variable `nextToken` is used to store the *look-ahead* token. This helps to report an error earlier.

Table Driven Predictive Parser

A non-recursive predictive parser can be constructed that maintains a stack (explicitly) and a table to select the appropriate production rule.

Parsing Table

The rows of the predictive parser table are indexed by the non-terminals and the columns are indexed by the terminals including the end-of-input marker ($). The content of the table are production rules or error situations. The table cannot have multiple entries.

Parsing Stack

The parsing stack can hold both terminals and non-terminals. At the beginning, the stack contains the end-of-stack marker (\$) and the start symbol on top of it.

Parsing Table Construction

- If $A \rightarrow \alpha$ is a production rule and $a \in \mathrm{FIRST}(\alpha)$, then $P[A][a] = A \rightarrow \alpha$.
- If $A \rightarrow \varepsilon$ is a production rule and $a \in \mathrm{FOLLOW}(A)$, then $P[A][a] = A \rightarrow \varepsilon$.

Actions

- If the top-of-stack is a terminal symbol (token) and matches with input token, both are consumed. A mismatch is an error.
- If the top-of-stack is a non-terminal A, the input token is a, $P[A][a]$ has the entry $A \rightarrow \alpha$, then A is to be replaced by α, with the head of α on the top of the stack.

Example

Consider the production rules of the non-terminal `SL`.

$$\boxed{\texttt{SL} \rightarrow \texttt{S SL} \mid \varepsilon}$$

The FIRST($\texttt{SL} \rightarrow \texttt{S SL}$) = `{id if while scan print}` and FOLLOW($\texttt{SL}$) =`{end else}`. So, $P[\texttt{SL}][\texttt{IDNTIFIER}] = P[\texttt{SL}][\texttt{IF}] = P[\texttt{SL}][\texttt{WHILE}] = P[\texttt{SL}][\texttt{SCAN}] = P[\texttt{SL}][\texttt{PRINT}] = \texttt{SL} \rightarrow \texttt{S SL}$ and $P[\texttt{SL}][\texttt{END}] = P[\texttt{SL}][\texttt{ELSE}] = \texttt{SL} \rightarrow \varepsilon$.

Note

Multiple entries in a table indicates that the grammar is not $LL(1)$. But it is interesting to note that in some cases we can drop (with proper consideration) some of these entries and construct a parser.

Example

Consider the ambiguous grammar G_1 for expressions.

$$E \;\rightarrow\; E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid ic$$

After the removal of left-recursion we get the following ambiguous, no-left-recursive grammar:

Example

$$E \rightarrow (E)E' \mid icE'$$

$$E' \rightarrow +EE' \mid -EE' \mid *EE' \mid /EE' \mid \varepsilon$$

We calculate FIRST(E') = {+ - * / ε } and the FOLLOW(E') = FOLLOW(E) = {$) + - * /}.

Example

Naturally,
$P[E'][\pm] = \{E' \rightarrow +EE', E' \rightarrow \varepsilon\}$ and
$P[E'][*/] = \{E' \rightarrow *EE', E' \rightarrow \varepsilon\}$.
We may drop the ε-productions from these four places and get a nice parsing table[a].

[a]But it does not work for all grammars. Consider $S \rightarrow aSa \mid bSb \mid \varepsilon$.

Note

It seems that the removal of two ε-production disambiguates the grammar. The corresponding unambiguoes grammar G_2 is as follows:

$$
\begin{array}{rcl}
E & \rightarrow & (E)E' \mid icE' \mid (E) \mid ic \\
E' & \rightarrow & +E \mid -E \mid *E \mid /E \mid \varepsilon
\end{array}
$$

We have $L(G_1) = L(G_2)$ and FOLLOW$(E') =$ {$)}, so there is no multiple entries in the table[a].

[a]How to maintain operator precedence?

Error Recovery

- The token on the top of stack does not match with the token in the input stream.
- The entry in the parsing table corresponding to nonterminal on the top of stack and the current input token is an error.

Panic Mode