Top-down Syntax Analysis

– Wilhelm/Maurer: Compiler Design, Chapter 8 –

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de
and
Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il

Subjects

- Functionality and Method
- Recursive Descent Parsing
- Using parsing tables
- Explicit stacks
- Creating the table
- LL(k)–grammars
- Other properties
- Handling Limitations

Top-Down Syntax Analysis

input: A sequence of symbols (tokens)

output: A syntax tree or an error message

method

- Read input from left to right
- Construct the syntax tree in a top-down manner starting with a node labeled with the start symbol
- **until** input accepted (or error) **do**
  - Predict expansion for the actual leftmost nonterminal (maybe using some lookahead into the remaining input) or
  - Verify predicted terminal symbol against next symbol of the remaining input

Finds leftmost derivations.

Grammar for Arithmetic Expressions

Left factored grammar G_2, i.e. left recursion removed.

$S \rightarrow E$	
$E \rightarrow TE'$	E generates T with a continuation E'
$E' \rightarrow +E \vert \epsilon$	E' generates possibly empty sequence of $+T$s
$T \rightarrow FT'$	T generates F with a continuation T'
$T' \rightarrow *T \vert \epsilon$	T' generates possibly empty sequence of $*F$s
$F \rightarrow \mathbf{id} \vert (E)$	

Recursive Descent Parsing

- parser is a program,
- a procedure X for each non-terminal X,
  - parses words for non-terminal X,
  - starts with the first symbol read (into variable *nextsym*),
  - ends with the following symbol read (into variable *nextsym*).
- uses one symbol lookahead into the remaining input.
- uses the **FiFo** sets to make the expansion transitions deterministic

$$\textbf{FiFo}(N \to \alpha) = FIRST_1(\alpha) \oplus_1 FOLLOW_1(N) = \begin{cases} FIRST_1(\alpha) \cup FOLLOW_1(N) & \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ FIRST_1(\alpha) & \text{otherwise} \end{cases}$$

Parser for G_2

```
program parser;
var nextsym: string;
proc scan;
 {reads next input symbol into nextsym}
proc error (message: string);
 {issues error message and stops parser}
proc accept; {terminates successfully}

proc S;
 begin E
 end ;

proc E;
 begin T; E'
 end ;
```

```
proc E';
 begin
   case nextsym in
    {"+"}: if nextsym = "+ "
           then scan
           else error( "+ expected") fi ; E;
    otherwise ;
   endcase
 end ;

proc T;
 begin F; T' end ;
proc T';
 begin
   case nextsym in
    {" * "}: if nextsym = "*"
           then scan
           else error( "* expected") fi ; T;
    otherwise ;
   endcase
 end ;
```

```
proc F;
 begin
   case nextsym in
    {"("}:   if nextsym = "("
             then scan
             else error( "( expected") fi ; E;
             if nextsym = ")"
             then scan else error(" ) expected") fi;
    otherwise if nextsym ="id"
             then scan else error("id expected") fi;
   endcase
 end ;
begin
scan; S;
if nextsym = "#" then accept else error("# expected") fi
end .
```

How to Construct such a Parser Program

Observation: Much redundant code generated. Why this?
Code was automatically generated from the grammar and the **FiFo** sets.
Nice application for a functional programming language!
Let $G = (V_N, V_T, P, S)$ be a context-free grammar and FiFo be the computed lookahead sets.
The functional program generating the parser would have the functions:

N_prog	:	$V_N \rightarrow$ code	nonterminals
C_prog	:	$(V_N \cup V_T)^* \rightarrow$ code	concantenations
S_prog	:	$V_N \cup V_T \rightarrow$ code	symbols

Parser Schema

```
program parser;
    var  nextsym: symbol;
    proc  scan;
        (* reads next input symbol into  nextsym *)
    proc  error (message: string);
        (* issues error message and stops the parser *)
    proc  accept;
        (* terminates parser successfully *)

    N_prog(X0);                        (* X0 start symbol *)
    N_prog(X1);
        ⋮
    N_prog(Xn);
```

```
begin
    scan;
    X_0;
    if nextsym = "#"
        then accept
        else error("...")
    fi
end
```

The Non-terminal Procedures

N = Non-terminal, C = Concatenation, S = Symbol

N_prog(X) = (* $X \rightarrow \alpha_1|\alpha_2|\cdots|\alpha_{k-1}|\alpha_k$ *)

proc X;
begin
case nextsym **in**
FiFo($X \rightarrow \alpha_1$) : C_progr(α_1);
FiFo($X \rightarrow \alpha_2$) : C_progr(α_2);
⋮
FiFo($X \rightarrow \alpha_{k-1}$) : C_progr($\alpha_{k-1}$);
otherwise C_progr(α_k);
endcase
end ;

C_progr($\alpha_1 \alpha_2 \cdots \alpha_k$) =
  S_progr(α_1); S_progr(α_2); ... S_progr(α_k);
S_progr(a) =
  if nextsym = a **then** scan
  else error ("a expected")
  fi
S_progr(Y) = Y

FiFo–sets should be disjoint (LL(1)–grammar)

A Generative Solution

Generate the control of a deterministic PDA from the grammar and the **FiFo** sets.

- At compiler–generation time construct a table M
  $M: V_N \times V_T \rightarrow P$
  $M[N, a]$ is the production used to expand nonterminal N when the current symbol is a
- For some grammars report that the table cannot be constructed
  The compiler writer can then decide to:
  - change the grammar (but not the language)
  - use a more general parser-generator
  - "Patch" the table (manually or using some rules)

Creating the table

Input: cfg G, $FIRST_1$ und $FOLLOW_1$ for G.

Output: The parsing table M or an indication that such a table cannot be constructed

Method: M is constructed as follows:
For all $X \to \alpha \in P$ and $a \in FIRST_1(\alpha)$, set $M[X, a] = (X \to \alpha)$.
If $\varepsilon \in FIRST_1(\alpha)$, for all $b \in FOLLOW_1(X)$, set $M[X, b] = (X \to \alpha)$.
Set all other entries of M to *error* .

Parser table cannot be constructed if at least one entry is set twice.
G is not LL(1)

Example – arithmetic expressions

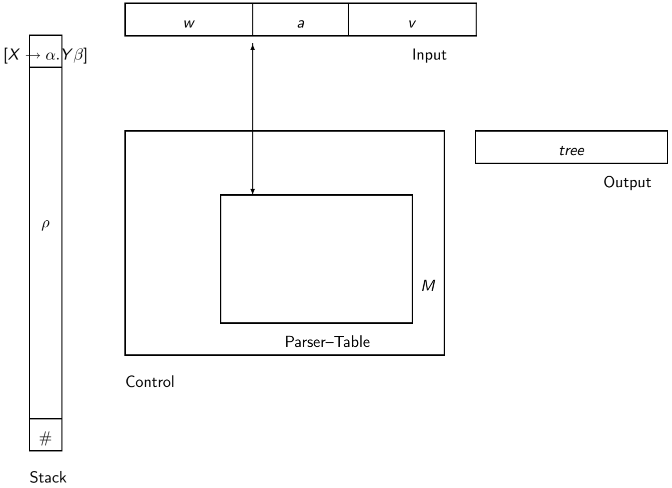
nonterminal	symbol	Production
S	$(, id$	$S \rightarrow E$
S	$+, *,), \#$	*error*
E	$(, id$	$E \rightarrow TE'$
E	$+, *,), \#$	*error*
E'	$+$	$E' \rightarrow +E$
E'	$), \#$	$E' \rightarrow \epsilon$
E'	$(, *, id$	*error*
T	$(, id$	$T \rightarrow FT'$
T	$+, *,), \#$	*error*
T'	$*$	$T' \rightarrow *T$
T'	$+,), \#$	$T' \rightarrow \epsilon$
T'	$(, id$	*error*
F	id	$F \rightarrow id$
F	$($	$F \rightarrow (E)$
F	$+, *,)$	*error*

LL-Parser Driver (interprets the table M)

```
program parser;
    var  nextsym: symbol;
    var  st: stack of  item;
    proc  scan;
        (* reads next input symbol into  nextsym *)
    proc  error (message: string);
        (* issues error message and stops the parser *)
    proc  accept;
        (* terminates parser successfully *)
    proc reduce;
        (* replaces [X → β.Yγ][Y → α.] by [X → βY.γ]  *)
    proc pop;
        (* removes topmost item from st *)
    proc push ( i : item);
        (* pushes i onto st *)
    proc replaceby ( i: item);
        (* replaces topmost item of st by i *)
```

begin
scan; push($[S' \rightarrow .S]$);
while nextsym $\neq$ "#"**do**
case top **in**
$[X \rightarrow \beta.a\gamma]$: **if** nextsym = a
then scan; replaceby($[X \rightarrow \beta a.\gamma]$
else error **fi** ;
$[X \rightarrow \beta.Y\gamma]$: **if** $M[Y, nextsym] = (Y \rightarrow \alpha)$
then push($[Y \rightarrow .\alpha]$)
else error **fi** ;
$[X \rightarrow \alpha.]$: reduce;
$[S' \rightarrow S.]$: **if** nextsym = "#"**then** accept
else error **fi**
endcase
od
end .

Explicit Stack
Deterministic Pushdown Automaton



LL(k)-grammar

Goal: formalizing our intuition when the expand-transitions of the Item-Pushdown-Automaton can be made deterministic.

Means: k-symbol lookahead into the remaining input.

LL(k)-grammar

Let $G = (V_N, V_T, P, S)$ be a cfg and k be a natural number.
G is an **LL(k)-grammar** iff the following holds:
if there exist two leftmost derivations

$$S \underset{lm}{\overset{*}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \underset{lm}{\overset{*}{\Longrightarrow}} ux \text{ and}$$

$$S \underset{lm}{\overset{*}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \underset{lm}{\overset{*}{\Longrightarrow}} uy, \text{ and if } k : x = k : y,$$

then $\beta = \gamma$.

The expansion of the leftmost non-terminal is always uniquely determined by

- the consumed part of the input and
- the next k symbols of the remaining input

LL(k)-grammar

Let $G = (V_N, V_T, P, S)$ be a cfg and k be a natural number.
G is an **LL(k)-grammar** iff the following holds:
if there exist two leftmost derivations

$$S \overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux \text{ and}$$

$$S \overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy, \text{ and if } k : x = k : y,$$

then $\beta = \gamma$.
The expansion of the leftmost non-terminal is always uniquely determined by

- the consumed part of the input and
- the next k symbols of the remaining input

Example 1

Let G_1 be the cfg with the productions

$$
\begin{array}{lll}
STAT & \rightarrow & \textbf{if id then } STAT \textbf{ else } STAT \textbf{ fi} \quad | \\
 & & \textbf{while id do } STAT \textbf{ od} \quad | \\
 & & \textbf{begin } STAT \textbf{ end} \quad | \\
 & & \textbf{id} := \textbf{id}
\end{array}
$$

G_1 is an LL(1)-grammar.

$$
STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\ \beta\ \alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\ x
$$

$$
STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\ \gamma\ \alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\ y
$$

From $1 : x = 1 : y$ follows $\beta = \gamma$,
e.g., from $1 : x = 1 : y =$ **if** follows
$\beta = \gamma =$ "**if id then** $STAT$ **else** $STAT$ **fi**"

Example 1

Let G_1 be the cfg with the productions

$$\begin{array}{lll} STAT & \rightarrow & \textbf{if id then } STAT \textbf{ else } STAT \textbf{ fi} \quad | \\ & & \textbf{while id do } STAT \textbf{ od} \quad | \\ & & \textbf{begin } STAT \textbf{ end} \quad | \\ & & \textbf{id} := \textbf{id} \end{array}$$

G_1 is an LL(1)-grammar.

$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\; STAT\; \alpha \underset{lm}{\Longrightarrow} w\, \beta\, \alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\, x$$

$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\; STAT\; \alpha \underset{lm}{\Longrightarrow} w\, \gamma\, \alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\, y$$

From $1 : x = 1 : y$ follows $\beta = \gamma$,
e.g., from $1 : x = 1 : y = \textbf{if}$ follows
$\beta = \gamma =$ "**if id then** *STAT* **else** *STAT* **fi**"

Example 2

Let G_2 be the cfg with the productions

$$
\begin{array}{lll}
STAT & \rightarrow & \textbf{if id then } STAT \textbf{ else } STAT \textbf{ fi} \quad | \\
 & & \textbf{while id do } STAT \textbf{ od} \quad | \\
 & & \textbf{begin } STAT \textbf{ end} \quad | \\
 & & \textbf{id} := \textbf{id} \quad | \\
 & & \textbf{id}\text{: } STAT \quad | \qquad (* \text{ labeled statem. } *) \\
 & & \textbf{id}\,(\textbf{id}\,) \qquad (* \text{ procedure call } *)
\end{array}
$$

Example 2 (cont'd)

G_2 is not an LL(1)–grammar.

$$STAT \underset{lm}{\overset{*}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\ \overbrace{\textbf{id} := \textbf{id}}^{\beta}\ \alpha \underset{lm}{\overset{*}{\Longrightarrow}} w\ x$$

$$STAT \underset{lm}{\overset{*}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\ \overbrace{\textbf{id} : STAT}^{\gamma}\ \alpha \underset{lm}{\overset{*}{\Longrightarrow}} w\ y$$

$$STAT \underset{lm}{\overset{*}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\ \overbrace{\textbf{id}(\textbf{id})}^{\delta}\ \alpha \underset{lm}{\overset{*}{\Longrightarrow}} w\ z$$

and $1 : x = 1 : y = 1 : z =$ "**id**",
and β, γ, δ are pairwise different.
G_2 is an LL(2)–grammar.
$2 : x =$ "**id** :=", $2 : y =$ "**id** :", $2 : z =$ "**id**(" are pairwise different.

Example 3

Let G_3 have the productions

$$
\begin{array}{lcll}
STAT & \rightarrow & \textbf{if}\ \ \textbf{id}\ \textbf{then}\ \ STAT\ \textbf{else}\ \ STAT\ \textbf{fi} & | \\
 & & \textbf{while}\ \ \textbf{id}\ \textbf{do}\ STAT\ \textbf{od} & | \\
 & & \textbf{begin}\ \ STAT\ \textbf{end} & | \\
 & & VAR := VAR & | \\
 & & \textbf{id}(\ IDLIST\ ) & (*\ \text{procedure call}\ *) \\
VAR & \rightarrow & \textbf{id}\ \ |\ \ \textbf{id}\,(IDLIST\,) & (*\ \text{indexed variable}\ *) \\
IDLIST & \rightarrow & \textbf{id}\ \ |\ \ \textbf{id},\ IDLIST &
\end{array}
$$

G_3 is not an LL(k)–grammar for any k.

Example 3

Let G_3 have the productions

$$
\begin{array}{lcll}
STAT & \rightarrow & \textbf{if}\ \ \textbf{id}\ \textbf{then}\ \ STAT\ \textbf{else}\ \ STAT\ \textbf{fi} & | \\
 & & \textbf{while}\ \ \textbf{id}\ \textbf{do}\ STAT\ \textbf{od} & | \\
 & & \textbf{begin}\ \ STAT\ \textbf{end} & | \\
 & & VAR := VAR & | \\
 & & \textbf{id}(\ IDLIST\ ) & (*\ \text{procedure call}\ *) \\
VAR & \rightarrow & \textbf{id}\ \mid\ \textbf{id}\,(IDLIST) & (*\ \text{indexed variable}\ *) \\
IDLIST & \rightarrow & \textbf{id}\ \mid\ \textbf{id},\ IDLIST &
\end{array}
$$

G_3 is not an LL(k)–grammar for any k.

Proof:

Assume G_3 to be LL(k) for a $k > 0$.

Let $STAT \Rightarrow \beta \overset{*}{\underset{lm}{\Longrightarrow}} x$ and $STAT \Rightarrow \gamma \overset{*}{\underset{lm}{\Longrightarrow}} y$ with

$$x = \textbf{id}\,(\underbrace{\textbf{id}, \textbf{id}, \ldots, \textbf{id}}_{\lceil \frac{k}{2} \rceil \text{ times}}) := \textbf{id} \text{ and } y = \textbf{id}\,(\underbrace{\textbf{id}, \textbf{id}, \ldots, \textbf{id}}_{\lceil \frac{k}{2} \rceil \text{ times}})$$

Then $k : x = k : y$,
but $\beta = "VAR := VAR" \neq \gamma = "\textbf{id}\,(IDLIST)"$.

Transforming to LL(k)

Factorization creates an LL(2)–grammar, equivalent to G_3.

The productions

$$STAT \rightarrow VAR := VAR \mid \mathbf{id}(IDLIST)$$

are replaced by

$$\begin{array}{lll} STAT & \rightarrow & ASSPROC \mid \mathbf{id} := VAR \\ ASSPROC & \rightarrow & \mathbf{id}(IDLIST)\ APREST \\ APREST & \rightarrow & := VAR \mid \varepsilon \end{array}$$

A non–LL(k)–language

Let $G_4 = (\{S, A, B\}, \{0, 1, a, b\}, P_4, S)$

$$P_4 = \left\{ \begin{array}{lll} S & \rightarrow & A \mid B \\ A & \rightarrow & aAb \mid 0 \\ B & \rightarrow & aBbb \mid 1 \end{array} \right\}$$

$L(G_4) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$.

G_4 is not LL(k) for any k.

Consider the two leftmost derivations

$$S \underset{lm}{\overset{0}{\Longrightarrow}} S \underset{lm}{\Longrightarrow} A \underset{lm}{\overset{*}{\Longrightarrow}} a^k 0 b^k$$

$$S \underset{lm}{\overset{0}{\Longrightarrow}} S \underset{lm}{\Longrightarrow} B \underset{lm}{\overset{*}{\Longrightarrow}} a^k 1 b^{2k}$$

With $u = \alpha = \varepsilon$, $\beta = A$, $\gamma = B$, $x =$ "$a^k 0 b^k$", $y =$ "$a^k 1 b^{2k}$" it holds $k : x = k : y$, but $\beta \neq \gamma$.

Since k can be chosen arbitrarily, we have G_4 is not LL(k) for any k.

There even is no LL(k)-grammar for $L(G_4)$ for any k.

Towards Checkable LL(k)–conditions

Theorem

G is LL(k)–grammar iff the following condition holds:
Are $A \to \beta$ and $A \to \gamma$ different productions in P, then

$$FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha) = \emptyset \quad \text{for all } \alpha \quad \text{with } S \overset{*}{\underset{lm}{\Longrightarrow}} wA\alpha$$

Theorem

Let G be a cfg without productions of the form $X \to \varepsilon$.
G is an LL(1)–grammar iff
for each non-terminal X with the alternatives $X \to \alpha_1 | \ldots | \alpha_n$
the sets $FIRST_1(\alpha_1), \ldots, FIRST_1(\alpha_n)$ are pairwise disjoint.

Theorem

G is LL(1) iff
For different productions $A \to \beta$ and $A \to \gamma$
$FIRST_1(\beta) \oplus_1 FOLLOW_1(A) \cap FIRST_1(\gamma) \oplus_1 FOLLOW_1(A) = \emptyset$.

Corollary:

G is LL(1) iff for all alternatives $A \to \alpha_1 | \ldots | \alpha_n$:

1. $FIRST_1(\alpha_1), \ldots, FIRST_1(\alpha_n)$ are pairwise disjoint; in particular, at most one of them may contain ε
2. $\alpha_i \stackrel{*}{\Longrightarrow} \varepsilon$ implies:

$$FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \emptyset \text{ for } 1 \leq j \leq n,\ j \neq i.$$

The condition of the Theorem was used in the parser construction!

Further Definitions and Theorems

- G is called a **strong LL(k)-grammar** if for each two different productions $A \rightarrow \beta$ and $A \rightarrow \gamma$

$$FIRST_k(\beta) \oplus_k FOLLOW_k(A) \cap FIRST_k(\gamma) \oplus_k FOLLOW_k(A) = \emptyset,$$

- A production is called **directly left recursive**, if it has the form $A \rightarrow A\alpha$
- A non-terminal A is called **left recursive** if it has a derivation $A \stackrel{+}{\Longrightarrow} A\alpha$.
- A cfg G is called **left recursive**, if G contains at least one left recursive non-terminal

Theorem

(a) *G is not $LL(k)$ for any k if G is left recursive.*

(b) *G is not ambiguous if G is $LL(k)$-grammar.*

Regular Right Sides

Left recursion

- prevents LL parsing,
- is used for lists and sequences,
- can be replaced by iteration, i.e., the Kleene star

Needs new definitions for derivation, First, and Follow!

Right-regular Context-free Grammar

$P : V_N \rightarrow RA$ is now a function from V_N into the set RA of regular expressions over $V_N \cup V_T$.
A pair (X, r) with $p(X) = r$ is written as $X \rightarrow r$.
New causes for non-determinism! Which?

Example: Arithmetic Expressions

$$S \rightarrow E$$
$$E \rightarrow T\{\{+\,|\,-\}\; T\}^*$$
$$T \rightarrow F\{\{*\,|\,/\}\; F\}^*$$
$$F \rightarrow (E)\,|\,\mathsf{Id}$$

Regular Derivations

A derivation step

$$
\begin{array}{llcll}
\text{(a)} & w\,X\,\beta & \underset{R,lm}{\Longrightarrow} & w\,\alpha\,\beta & \text{mit } \alpha = p(X) \\
\text{(b)} & w\,(r_1 \mid \ldots \mid r_n)\,\beta & \underset{R,lm}{\Longrightarrow} & w\,r_i\,\beta & \text{für } 1 \leq i \leq n \\
\text{(c)} & w\,(r)^*\,\beta & \underset{R,lm}{\Longrightarrow} & w\,\beta & \\
\text{(d)} & w\,(r)^*\,\beta & \underset{R,lm}{\Longrightarrow} & w\,r\,(r)^*\,\beta &
\end{array}
$$

Regular leftmost derivation for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

$$S \underset{R,lm}{\Longrightarrow} E \underset{R,lm}{\Longrightarrow} T\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{+|-\}T\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + T\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id}\{*|/\}F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$
$$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$$

Computation of First

Compute ε-productivity first.

$$
\begin{aligned}
eps(a) &= false, \quad \text{for } a \in V_T \\
eps(\varepsilon) &= true \\
eps(r^*) &= true \\
eps(X) &= eps(r), \text{ if } p(X) = r \text{ for } X \in V_N \\
eps((r_1| \ldots |r_n)) &= \bigvee_{i=1}^{n} eps(r_i) \\
eps((r_1 \ldots r_n)) &= \bigwedge_{i=1}^{n} eps(r_i)
\end{aligned}
$$

then ε-free First

$$
\begin{aligned}
&\varepsilon\text{-}\mathit{ffi}(\varepsilon) = \emptyset \\
&\varepsilon\text{-}\mathit{ffi}(a) = \{a\} \\
&\varepsilon\text{-}\mathit{ffi}(r^*) = \varepsilon\text{-}\mathit{ffi}(r) \\
&\varepsilon\text{-}\mathit{ffi}(X) = \varepsilon\text{-}\mathit{ffi}(r),\ \text{if } p(X) = r \\
&\varepsilon\text{-}\mathit{ffi}((r_1|\ldots|r_n)) = \bigcup_{1 \le i \le n} \varepsilon\text{-}\mathit{ffi}(r_i) \\
&\varepsilon\text{-}\mathit{ffi}((r_1 \ldots r_n)) = \bigcup_{1 \le j \le n} \{\varepsilon\text{-}\mathit{ffi}(r_j) \mid \bigwedge_{1 \le i < j} \mathit{eps}(r_i)\}
\end{aligned}
$$

Computation of Follow

Follow depends on the right context of a subexpression:
Unusual "bottom-up" recursion!

(1) $FOLLOW_1([S' \rightarrow .S]) = \{\#\}$ The eof symbol '#' follows after each input word.

(2) $FOLLOW_1([X \rightarrow \cdots(r_1|\cdots|.r_i|\cdots|r_n)\cdots]) =$
$FOLLOW_1([X \rightarrow \cdots .(r_1|\cdots|r_i|\cdots|r_n)\cdots])$ for $1 \leq i \leq n$

(3) $FOLLOW_1([X \rightarrow \cdots(\cdots .r_i r_{i+1}\cdots)\cdots]) =$

$$\varepsilon\text{-}ffi(r_{i+1}) \cup \begin{cases} FOLLOW_1([X \rightarrow \cdots(\cdots r_i.r_{i+1}\cdots)\cdots]), \\ \quad \text{if } eps(r_{i+1}) = true \\ \emptyset \quad \text{otherwise} \end{cases}$$

(4) $FOLLOW_1([X \rightarrow \cdots(r_1\cdots r_{n-1}.r_n)\cdots]) =$ $(FOLLOW_1)$
$FOLLOW_1([X \rightarrow \cdots .(r_1\cdots r_{n-1}r_n)\cdots])$

(5) $FOLLOW_1([X \rightarrow \cdots(.r)^*\cdots]) =$
$\varepsilon\text{-}ffi(r) \cup FOLLOW_1([X \rightarrow \cdots .(r)^*\cdots])$

(6) $FOLLOW_1([X \rightarrow .r]) = \bigcup FOLLOW_1([Y \rightarrow \cdots .X\cdots])$

then the FiFo-Sets

$$\textbf{FiFo}(N \rightarrow \alpha) = \mathit{FIRST}_1(\alpha) \oplus_1 \mathit{FOLLOW}_1(N) = \begin{cases} \mathit{FIRST}_1(\alpha) \cup \mathit{FOLLOW}_1(N) & \alpha \overset{*}{\Longrightarrow} \epsilon \\ \mathit{FIRST}_1(\alpha) & \text{otherwise} \end{cases}$$

This formulation allows efficient computation, see *Pure Union Problems* in the Book!

Recursive Descent Parsing

```
struct symbol nextsym;

/* Returns next input symbol */
void scan();

/* Prints the error message and
   stops the run of the parser */
void error(String errorMessage);

/* Announces the end of the analysis and
   stops the run of the parser */
void accept();

/* Translating the input grammar */
p_progr(X_0 → α_0);
p_progr(X_1 → α_1);
          ⋮
p_progr(X_n → α_n);
```

```
void parser() {
  scan();
  X0();

  if(nextsym == "\#")
    accept();
  else
    error("...");
}
```

```
p_progr(X → .α)

/* ...we create an according method like this.*/
void X() {
  progr([X → .α]);
}

void progr([X → ··· .(α1|α2|···|αk−1|αk)···]) {
  switch () {
    case (nextsym ∈ FiFo([X → ···(.α1|α2|···|αk−1|αk)···])):
      progr([X → ···(.α1|α2|···|αk−1|αk)···]);
    break;
    case (nextsym ∈ FiFo([X → ···(α1|.α2|···|αk−1|αk)···])):
      progr([X → ···(α1|.α2|···|αk−1|αk)···]);
    break;
        ⋮
    case (nextsym ∈ FiFo([X → ···(α1|α2|···|.αk−1|αk)···])):
      progr([X → ···(α1|α2|···|.αk−1|αk)···]);
    break;
    default:
      progr([X → ···(α1|α2|···|αk−1|.αk)···]);
  }
}
```

```
void progr([X → ··· .(α)* ···]) {
  while(nextsym ∈ FIRST₁(α))) {
    progr([X → ··· .α ···]);
  }
}

void progr([X → ··· .(α)⁺ ···]) {
  do {
    progr([X → ··· .α ···]);
  } while(nextsym ∈ FIRST₁(α));
}

void progr([X → ··· .ϵ ···]) {}
```

For $a \in V_T$ is

```
void progr([X → ··· .a···]) {
  if(nextsym == a)
    scan();
  else
    error("...");
}
```

For $Y \in V_N$ is

```
void progr([X → ··· .Y···]) = void Y()
```

RLL Parser for the Expression Grammar

```
symbol nextsym;

/* Returns next input symbol */
symbol scan();

/* Prints the error message and
   stops the run of the parser */
void error(String errorMessage);

/* Announces the end of the analysis and
   stops the run of the parser */
void accept();
```

RLL Parser for the Expression Grammar

```
void S() {
  E();
}
void E() {
  T();
  while(nextsym == "+" || nextsym == "-") {
    switch (nextsym) {
      case "+":
        if(nextsym == "+")
          scan();
        else
          error("+␣expected");
      break;
      default:
        if(nextsym == "-")
          scan();
        else
          error("-␣expected");
    }
    T();
  }
}
```

RLL Parser for the Expression Grammar

```
void T() {
  F();
  while(nextsym == "*" || nextsym == "/") {
    switch (nextsym) {
      case "*":
        if(nextsym == "*")
          scan();
        else
          error("*␣expected");
      break;
      default:
        if(nextsym == "/")
          scan();
        else
          error("/␣expected");
    }
    F();
  }
}
```

RLL Parser for the Expression Grammar

```
void F() {
  switch (nextsym) {
    case "(":
      E();
      if(nextsym == ")")
        scan();
      else
        error(") expected");
    default:
      if(nextsym == "id")
        scan();
      else
        error("id expected");
  }
}
```

RLL Parser for the Expression Grammar

```
void parser() {
  scan();
  S();
  if(nextsym == "#")
    accept();
  else
    error("#␣expected");
}
```