Top-down Syntax Analysis

Sebastian Hack
(based on slides by Reinhard Wilhelm and Mooly Sagiv)

http://compilers.cs.uni-saarland.de

Compiler Construction Core Course 2017
Saarland University

Top-Down Syntax Analysis

input: A sequence of symbols (tokens)

output: A syntax tree or an error message

- 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.

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

G_2 defines the same language as G_0 und G_1.

Grammar for Arithmetic Expressions

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

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

G_2 defines the same language as G_0 und G_1.

But the parse tree is not so suitable as an abstract syntax tree!

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

$$\textbf{FiFo}(N \to \alpha) = \mathit{FIRST}_1(\alpha) \oplus_1 \mathit{FOLLOW}_1(N)$$

The $FIRST_1$ Sets

- A production $N \to \alpha$ is applicable for symbols that "begin" α
- Example: Arithmetic Expressions, Grammar G_2
  - The production $F \to id$ is applied when the current symbol is **id**
  - The production $F \to (E)$ is applied when the current symbol is **(**
  - The production $T \to F$ is applied when the current symbol is **id** or **(**
- Formal definition:

$$FIRST_1(\alpha) = \{1 : w \mid \alpha \overset{*}{\Longrightarrow} w, w \in V_T^*\}$$

The $FOLLOW_1$ Sets

- A production $N \to \epsilon$ is applicable for symbols that "can follow" N in some derivation
- Example: Arithmetic Expressions, Grammar G_2
  - The production $E' \to \epsilon$ is applied for symbols $\#$ and $)$
  - The production $T' \to \epsilon$ is applied for symbols $\#$, $)$ and $+$
- Formal definition:

$$FOLLOW_1(N) = \{a \in V_T \mid \exists \alpha, \gamma : S \stackrel{*}{\Longrightarrow} \alpha N a \gamma\}$$

Definitions

Let $k \geq 1$

- k-prefix of a word $w = a_1 \dots a_n$

$$k : w = \begin{cases} a_1 \dots a_n & \text{if } n \leq k \\ a_1 \dots a_k & \text{otherwise} \end{cases}$$

- k-concatenation

$$\oplus_k : V^* \times V^* \;\rightarrow\; V^{\leq k}, \text{ defined by } u \oplus_k v = k : uv$$

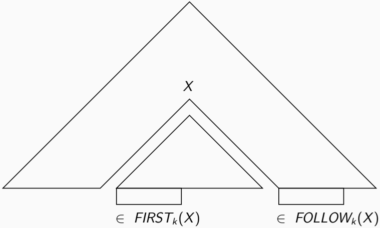
- extended to languages

$$k : L = \{k : w \mid w \in L\}$$

$$L_1 \oplus_k L_2 = \{x \oplus_k y \mid x \in L_1, y \in L_2\}$$

$$V^{\leq k} = \bigcup_{i=1}^{k} V^i \text{ set of words of length at most } k$$

$FIRST_k$ and $FOLLOW_k$



- set of k–prefixes of terminal words for α

$$FIRST_k : (V_N \cup V_T)^* \rightarrow 2^{V_T^{\leq k}}$$

$$FIRST_k(\alpha) = \{k : u \mid \alpha \overset{*}{\Longrightarrow} u\}$$

- set of k–prefixes of terminal words that may immediately follow X

$$FOLLOW_k : V_N \rightarrow 2^{V_{T\#}^{\leq k}}$$

$$FOLLOW_k(X) = \{w \mid S \overset{*}{\Longrightarrow} \beta X \gamma \text{ and } w \in FIRST_k(\gamma)\}$$

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
```

```
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

- Code was automatically generated from the grammar and the **FiFo** sets.
- The program generating the parser has the functions:

N_prog	:	$V_N \rightarrow \text{code}$	nonterminals
C_prog	:	$(V_N \cup V_T)^* \rightarrow \text{code}$	concantenations
S_prog	:	$V_N \cup V_T \rightarrow \text{code}$	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(X_0);                              (* X_0 start symbol *)
    N_prog(X_1);
        ⋮
    N_prog(X_n);
```

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

proc X;
begin
case nextsym **in**
FiFo($X \to \alpha_1$) : C_progr(α_1);
FiFo($X \to \alpha_2$) : C_progr(α_2);
$\vdots$
FiFo($X \to \alpha_{k-1}$) : C_progr(α_{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 have to 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
  $M\colon V_N \times V_T \to P$
  $M[N, a]$ 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

M is constructed as follows:

- For all $X \rightarrow \alpha \in P$ and $a \in FIRST_1(\alpha)$, set $M[X, a] = (X \rightarrow \alpha)$
- If $\varepsilon \in FIRST_1(\alpha)$, for all $b \in FOLLOW_1(X)$, set $M[X, b] = (X \rightarrow \alpha)$
- Set all other entries of M to *error*

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

Example – arithmetic expressions

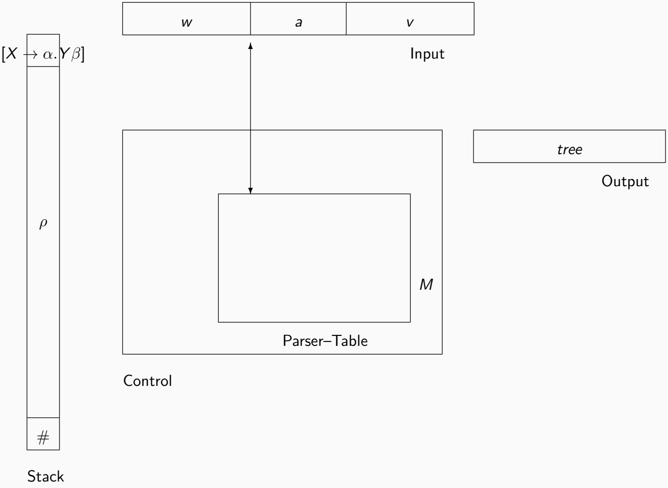
nonterminal	symbol	Production
S	$(, id$	$S \rightarrow E$
S	$+, *,), \#$	*error*
E	$(, id$	$E \rightarrow TE'$
E	$+, *,), \#$	*error*
E'	$+$	$E' \rightarrow +E$
E'	$), \#$	$E' \rightarrow \epsilon$
E'	$(, *, id$	*error*
T	$(, id$	$T \rightarrow FT'$
T	$+, *,), \#$	*error*
T'	$*$	$T' \rightarrow *T$
T'	$+,), \#$	$T' \rightarrow \epsilon$
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' → .S] );
    while  nextsym ≠ "#" do
    case top in
    [X → β.aγ]:      if  nextsym = a
                     then  scan; replaceby([X → βa.γ])
                     else error fi ;
    [X → β.Yγ] :     if  M[Y, nextsym] = (Y → α)
                     then  push([Y → .α])
                     else  error fi ;
    [X → α.]:        reduce;
    [S' → 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

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

  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

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

Example 1

Let G_1 be the cfg with the productions

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

G_1 is an LL(1)-grammar.

$$
\begin{array}{lclclcl}
STAT & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ STAT\,\alpha & \underset{lm}{\Longrightarrow} & w\,\beta\,\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,x \\
STAT & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ STAT\,\alpha & \underset{lm}{\Longrightarrow} & w\,\gamma\,\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,y
\end{array}
$$

From $1 : x = 1 : y$ follows $\beta = \gamma$,
e.g., from $1 : x = 1 : y = \textbf{if}$ follows

Example 2

Let G_2 be the cfg with the productions

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

Example 2 (cont'd)

G_2 is not an LL(1)–grammar.

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

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

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

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

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

Example 3

Let G_3 have the productions

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

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 $\beta = \text{"}VAR := VAR\text{"} \neq \gamma = \text{"}\textbf{id}\,(IDLIST)\text{"}$.

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

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

A non–LL(k)–language

Let

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

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

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

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.

LL(k)–conditions

Theorem

G is LL(1) iff for different productions $A \rightarrow \beta$ *and* $A \rightarrow \gamma$
$FIRST_1(\beta) \oplus_1 FOLLOW_1(A) \cap FIRST_1(\gamma) \oplus_1 FOLLOW_1(A) = \emptyset$

Corollary

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

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:*

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

The Theorem was used in the parser construction!

Further Definitions and Theorems

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

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

- SLL(1) = LL(1)
- 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.*