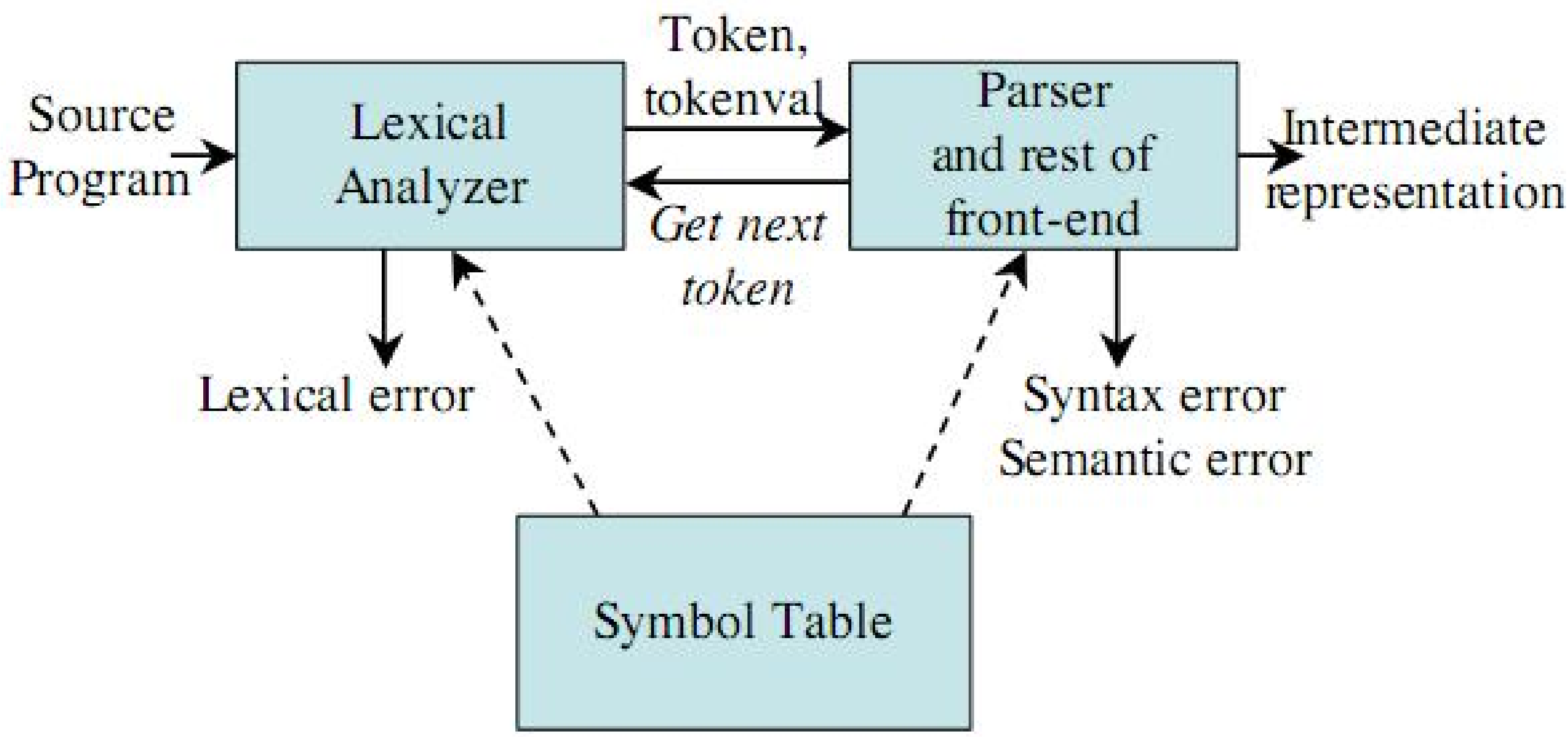
Syntaktická analýza

Ján Šturc

Zima 208

Position of a Parser in the Compiler Model



The parser

- The task of the parser is to check syntax
- The syntax-directed translation stage in the compiler's front-end checks static semantics a produces an intermediate representation (IR) of the source program
  - Abstract syntax trees (ASTs)
  - Control-flow graphs (CFGs) with triples, three-add code, or register transfer lists
  - WHIRL (SGI Pro64 compiler) has 5 IR levels!

Error handling

- A good compiler should assist in identifying and locating errors
  - Lexical errors: important, compiler can easily recover and continue
  - Syntax errors: most important for compiler, can almost always recover
  - Static semantic errors: important, can sometimes recover
  - Dynamic semantic errors: hard or impossible to detect at compile time, runtime checks are required
  - Logical errors: impossible to detect, in the majority of cases

Viable-prefix property

- The viable-prefix property of LL/LR parsers allows early detection of syntax errors
  - Goal: detection of an error as soon as possible without consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

prefix
```
...
for (;)
...
```

Error detected here

prefix
```
...
DO 10 I = 1;0
...
```

Error Recovery Strategies

- Panic mode
  - Discard input until a token in a set of designated synchronizing tokens is found
- Phrase-level recovery
  - Perform local correction on the input to repair the error
- Error productions
  - Augment grammar with productions for erroneous constructs
- Global correction
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Grammar (opakovanie)

- Context-free grammar is a 4-tuple $G=(N,T,R,S)$ where
  - T is a finite set of tokens (terminal symbols),
  - N is a finite set of nonterminals,
  - R is a finite set of productions of the form $\alpha \to \beta$, where $\alpha \in (N \cup T)^* N (N \cup T)^*$ and $\beta \in (N \cup T)^*$ and
  - S is a designated start symbol $S \in N$.

Notational Conventions
(nie veľmi dodržované)

- Terminals
  - $a,b,c,\ldots \in T$
  - specific terminals: **0, 1, id, +**
- Nonterminals
  - $A,B,C,\ldots \in N$
  - specific nonterminals: expr, term, stmt
- Grammar symbols
  - $X,Y,Z \in (N \cup T)$
- Strings of terminals
  - $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  - $\alpha,\beta,\gamma, \ldots \in (N \cup T)^*$

Derivations (opakovanie)

- The one-step derivation is defined by
  $\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$
  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is leftmost $\Rightarrow$lm if α does not contain a nonterminal
  - $\Rightarrow$ is rightmost $\Rightarrow$rm if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- The language generated by G is defined by
  $L\ (G) = \{w \mid S \Rightarrow^+ w\}$

Derivation (Example)

$E \rightarrow E \mathbf{+} E$

$E \rightarrow E \mathbf{*} E$

$E \rightarrow \mathbf{(} E \mathbf{)}$

$E \rightarrow \mathbf{-} E$

$E \rightarrow \mathbf{id}$

1. $E \Rightarrow - E \Rightarrow \mathbf{-\ id}$
2. $E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+\ id} \Rightarrow_{rm} \mathbf{id + id}$
3. $E \Rightarrow_{lm} E \mathbf{*} E \Rightarrow_{lm} E * E \mathbf{+} E \overset{*}{\Rightarrow}_{lm} \mathbf{id*id + id}$

Grammar Classification

- A grammar G is said to be
  - Regular if it is right linear where each production is of the form $A \rightarrow w\,B$ or $A \rightarrow w$
  - or left linear where each production is of the form $A \rightarrow B\,w$ or $A \rightarrow w$
  - Context free if each production is of the form $A \rightarrow \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$
  - Context sensitive if each production is of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $A \in N$, $\alpha, \gamma, \beta \in (N \cup T)^*$, $|\gamma| > 0$.
  - Unrestricted (Recursively enumerable).

Chomsky hierarchy

L(regular) ⊆ L(context free) ⊆

L(context sensitive) ⊆ L(unrestricted)

where L(T) = { L(G) | G is of type T }

That is, the set of all languages generated by grammars G of type T

Examples:

Every finite language is regular

$L_1 = \{ a^n b^n \mid n \geq 1 \}$ is context free

$L_2 = \{ a^n b^n c^n \mid n \geq 1 \}$ is context sensitive

Parsing

- Universal (any CF grammar)
  – Cocke-Younger-Kasimi
  – Earley
- Top-down (CF|CS grammar with restrictions)
  – Recursive descent (predictive parsing)
  – LL (Left-to-right, Leftmost derivation) methods
- Bottom-up (CF|CS grammar with restrictions)
  – Operator precedence parsing
  – LR (Left-to-right, Rightmost derivation)
    - methods SLR, canonical LR, LALR

Top-Down Parsing

- Recursive-descent parsing and LL methods (Left-to-right, Leftmost derivation)

Grammar:

$E \rightarrow T + T$

$T \rightarrow (E)$

$T \rightarrow - E$

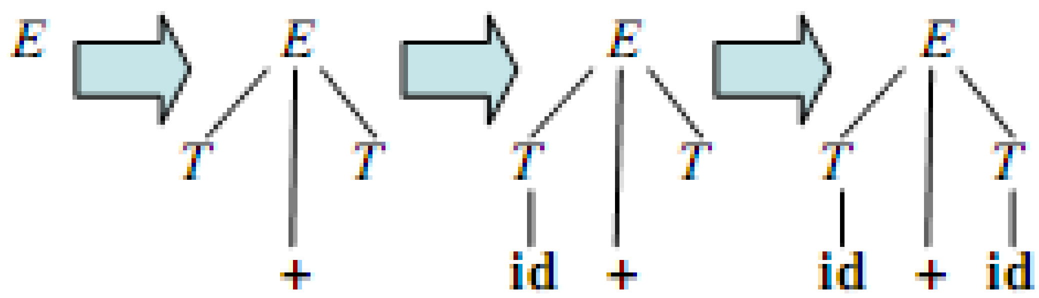
$T \rightarrow id$

Leftmost derivation:

$E \Rightarrow_{lm} T + T$

$\Rightarrow_{lm} id + T$

$\Rightarrow_{lm} id + id$



Left Recursion

- Productions of the form

$$A \rightarrow A\ \alpha$$
$$\mid\ \beta$$
$$\mid\ \gamma$$

are left recursive

- When one of the productions in a grammar is left recursive then a predictive parser may loop forever

Left Recursion Elimination

Arrange the nonterminals in some order $A_1, A_2, ..., A_n$

for i = 1, ..., n **do**

for j = 1, ..., **i**-1 **do**

replace each $A_i \rightarrow A_j \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$

where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$

end

eliminate the immediate left recursion in A_i

enddo

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

$A \rightarrow A\,\alpha \mid \beta \mid \gamma \mid A\,\delta$

$A \rightarrow A\,(\alpha \mid \delta)$

$A \rightarrow (\beta \mid \gamma)$

into a right-recursive production:

$A \rightarrow \beta\,A_R \mid \gamma\,A_R$	$A \rightarrow (\beta \mid \gamma)A_R$
$A_R \rightarrow \alpha\,A_R \mid \delta\,A_R \mid \varepsilon$	$A_R \rightarrow (\alpha \mid \delta)\,A_R \mid \varepsilon$

Here productions are writen in two forms black and blue one. I prefer the blue one.

Example Left Recursion Elimination

$A \rightarrow B\ C \mid \mathbf{a}$
$B \rightarrow C\ A \mid A\ \mathbf{b}$
$C \rightarrow A\ B \mid C\ C \mid \mathbf{a}$

Choose arrangement: A, B, C

i = 1: nothing to do

i = 2, j = 1: $B \rightarrow C\ A \mid A\ \mathbf{b}$

$\Rightarrow B \rightarrow C\ A \mid B\ C\ \mathbf{b} \mid \mathbf{a}\ \mathbf{b}$

$\Rightarrow_{\text{(imm)}}$ $B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

i = 3, j = 1: $C \rightarrow A\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow C \rightarrow B\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

i = 3, j = 2: $C \rightarrow B\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow C \rightarrow C\ A\ B_R\ C\ B \mid \mathbf{a}\ \mathbf{b}\ B_R\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}}$ $C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ CR \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

Left Factoring

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing
- Replace productions

$$A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \ldots \mid \alpha\ \beta_n \mid \gamma$$

- with

$$A \rightarrow \alpha\ A_R \mid \gamma$$

$$A_R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Predictive Parsing

- Eliminate left recursion from the grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)

FIRST (Revisited)

FIRST(α) = the set of terminals that begin all strings derived from α

FIRST(a) = {a} if $a \in T$

FIRST(A) = $\cup_{A \to \alpha}$ {FIRST(α) : $A \to \alpha \in R$}$\cup$ {ε if $A \to \varepsilon \in R$}

Computation of the FIRST($X_1X_2 \ldots X_k$)

FIRST($X_1X_2 \ldots X_k$) = FIRST(X_1);

j = 1; **while** $\varepsilon \in$ FIRST(X_j) **do**

{ FIRST($X_1X_2 \ldots X_k$): = FIRST($X_1X_2 \ldots X_k$) $\cup$ FIRST(X_j);

j++;

}

FOLLOW

- FOLLOW(A) = the set of terminals that can immediately follow nonterminal A
- Computation of the FOLLOWS:

for all A **do** FOLLOW(A): = Ø;

FOLLOW(S) = {$};

repeat

  for all $(B \rightarrow \alpha\ A\ \beta) \in R$ **do**

    { FOLLOW(A):= FOLLOW(A) ∪ FIRST(β) - {ε} }

  for all $(B \rightarrow \alpha\ A\ \beta) \in R$ and $(\varepsilon \in FIRST(\beta)$ or $\beta = \varepsilon)$ **do**

    {FOLLOW(A):= FOLLOW(A) ∪ FOLLOW(B) }

until something added;

LL(1) Grammar

A grammar G is LL(1) if it is not left-recursive and, if for each collections of productions

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

for a nonterminal A the following holds:

1. $\mathrm{FIRST}(\alpha_{i)} \cap \mathrm{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
2. if $\alpha_i \Rightarrow^* \varepsilon$ then
   a. $\alpha_j \Rightarrow^* \varepsilon$ for all $i \neq j$
   b. $\mathrm{FIRST}(\alpha j) \cap \mathrm{FOLLOW}(A) = \emptyset$ for all $i \neq j$

Non-LL(1) Examples

Grammar	Not LL(1) because
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\,S \mid \mathbf{a}$	$\mathrm{FIRST}(\mathbf{a}\,S) \cap \mathrm{FIRST}(\mathbf{a}) \neq \emptyset$
$S \rightarrow \mathbf{a}\,R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \rightarrow^{*} \varepsilon$ and $\varepsilon \rightarrow^{*} \varepsilon$
$S \rightarrow \mathbf{a}\,R\,\mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: $\mathrm{FIRST}(S) \cap \mathrm{FOLLOW}(R) \neq \emptyset$

Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Using FIRST and FOLLOW to Write a Recursive Descent Parser

S → expr **$**
expr → term rest

rest → + term rest
        | - term rest
        | ε

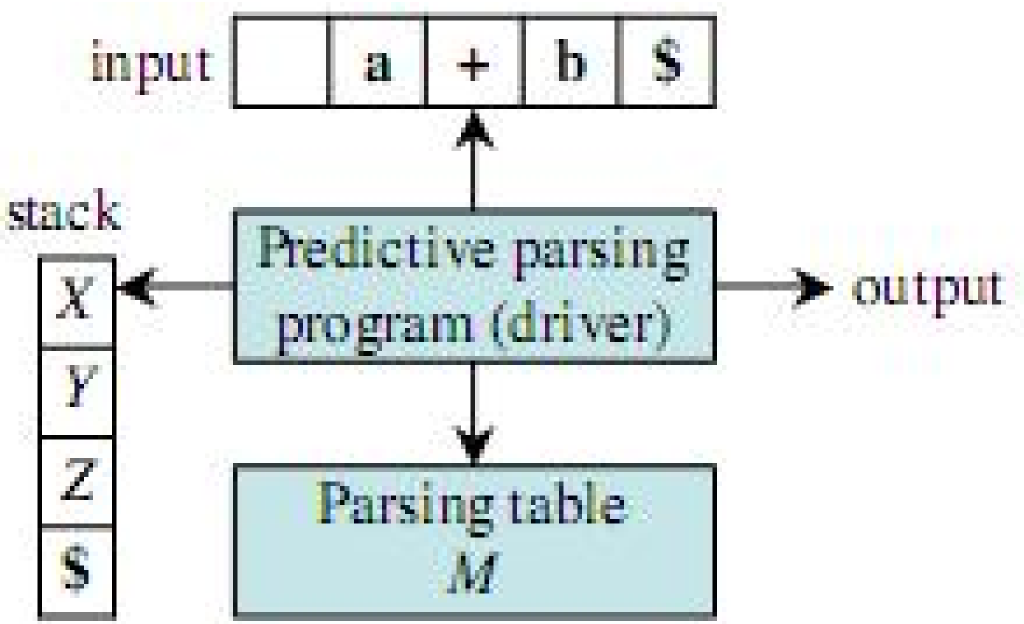
term → id

FIRST(+ term rest) = { **+** }
FIRST(- term rest) = { **-** }
FOLLOW(rest) = { **$** }

```
procedure rest();
begin
   if lookahead in FIRST(+ term rest)
     then  match('+'); term(); rest()
   else if lookahead in FIRST(- term rest)
     then match('-'); term(); rest()
   else if lookahead in FOLLOW(rest)
     then return
   else error()
end;
```

Non-Recursive Predictive Parsing

- Given an LL(1) grammar G=(N,T,P,S)
  construct a table M[A,a] for A ∈ N, a ∈ T and use a driver program with a stack



Constructing a Predictive Parsing Table

for each production A → α **do**
    for each a ∈ FIRST(α) **do**
        add A → α to M[A,a]
    enddo
    if ε ∈ FIRST(α) **then**
        for each b ∈ FOLLOW(A) **do**
            add A → α to M[A,b]
        enddo
    endif
enddo
Mark each undefined entry in M error

Example Table

$E \rightarrow T\ E_R$
$E_R \rightarrow +\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T E_R$	(id	$)
$E_R \rightarrow + T E_R$	+	$)
$E_R \rightarrow \varepsilon$	ε	
$T \rightarrow F T_R$	(id	+ $)
$T_R \rightarrow * F T_R$	*	+ $)
$T_R \rightarrow \varepsilon$	ε	
$F \rightarrow (E)$	(	* + $)
$F \rightarrow$ id	id	

	id	+	*	(	)	$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$		
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$			$T \rightarrow F T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Parsing Ambigous Grammar

An ambiguous

grammar
$S \rightarrow i\ E\ t\ S\ S_R \mid a$
$S_R \rightarrow e\ S \mid \varepsilon$
$E \rightarrow b$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$	i	e \$
$S \rightarrow \mathbf{a}$	a	
$S_R \rightarrow \mathbf{e}\ S$	e	e \$
$S_R \rightarrow \varepsilon$	ε	
$E \rightarrow \mathbf{b}$	b	t

	a	b	e	i	t	\$
S	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow \mathbf{e}\ S$			$S_R \rightarrow \varepsilon$
E		$E \rightarrow \mathbf{b}$				

Strictly spoken the grammar is not LL(1). LL(1) grammars are unambigous.
We give priority to the rules with longer right-hand side.

Predictive Parsing Program (Driver)

```
push($);
push(S);
a := lookahead;
repeat
    X := pop();
    if X is a terminal or X = $ then match(X)
            /* move to next token, a := lookahead */
    else if M[X,a] = X → Y1Y2...Yk
                    then {push(Yk, Yk-1, ..., Y2, Y1) // such that Y1 is on top
                          produce output and/or invoke actions}
    else error()
    endif
until X = $;
```

Example: Table-Driven Parsing

Stack	Input	Production applied
$\$E$	**id+id*id$**	
$\$E_R T$	**id+id*id$**	$E \rightarrow T\ E_R$
$\$E_R T_R F$	**id+id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R$**id**	**id+id*id$**	$F \rightarrow$ **id**
$\$E_R T_R$	**+id*id$**	
$\$E_R$	**+id*id$**	$T_R \rightarrow \varepsilon$
$\$E_R T+$	**+id*id$**	$E_R \rightarrow +\ T\ E_R$
$\$E_R T$	**id*id$**	
$\$E_R T_R F$	**id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R$**id**	**id*id$**	$F \rightarrow$ **id**
$\$E_R T_R$	***id$**	
$\$E_R T_R F*$	***id$**	$T_R \rightarrow *\ F\ T_R$
$\$E_R T_R F$	**id$**	
$\$E_R T_R$**id**	**id$**	$F \rightarrow$ **id**
$\$E_R T_R$	**$**	
$\$E_R$	**$**	$T_R \rightarrow \varepsilon$
$	**$**	$E_R \rightarrow \varepsilon$

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

synch: pop A and skip input till synch token or skip until FIRST(A) found

FOLLOW(E) = { $,) }
FOLLOW(E_R) = { $,) }
FOLLOW(T) = { +, $) }
FOLLOW(T_R) = { + ,$) }
FOLLOW(F) = { *, + ,$) }

A	id	+	*	(	)	$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	*synch*	*synch*
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$	*synch*		$T \rightarrow F T_R$	*synch*	*synch*
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	*synch*	*synch*	$F \rightarrow (E)$	*synch*	*synch*

Phrase-Level Recovery

Change input stream by inserting missing token

For example: id id is changed into id * id

	id	+	*	(	)	$
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	*synch*	*synch*
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	*synch*		$T \rightarrow F\ T_R$	*synch*	*synch*
T_R	*insert **	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	*synch*	*synch*	$F \rightarrow (\ E\ )$	*synch*	*synch*

insert *: insert missing * and redo the production

Error Productions

$E \rightarrow T\ ER$
$ER \rightarrow +\ T\ ER \mid \varepsilon$
$T \rightarrow F\ TR$
$TR \rightarrow *\ F\ TR \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Add error production:

$TR \rightarrow F\ TR$

to ignore missing *, e.g.: id id

	id	+	*	(	)	$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	*synch*	*synch*
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$	*synch*		$T \rightarrow F T_R$	*synch*	*synch*
T_R	$T_R \rightarrow F T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	*synch*	*synch*	$F \rightarrow (E)$	*synch*	*synch*

Note to project

- There are compiler-compilers (TWSs) based on top-down syntax analysis
  – AntLR
  – CoCo/R
- The language class is narrower than in the case bottom-up syntax analysis
- but it is easier to insert semantics
  – We always know, which production is proceed
  – Semantic routines can appear anywhere in the productions