Syntax Analysis

Mooly Sagiv
Tel-Aviv University
Schriber 317
Phone (03)640-7606
sagiv@math.tau.ac.il

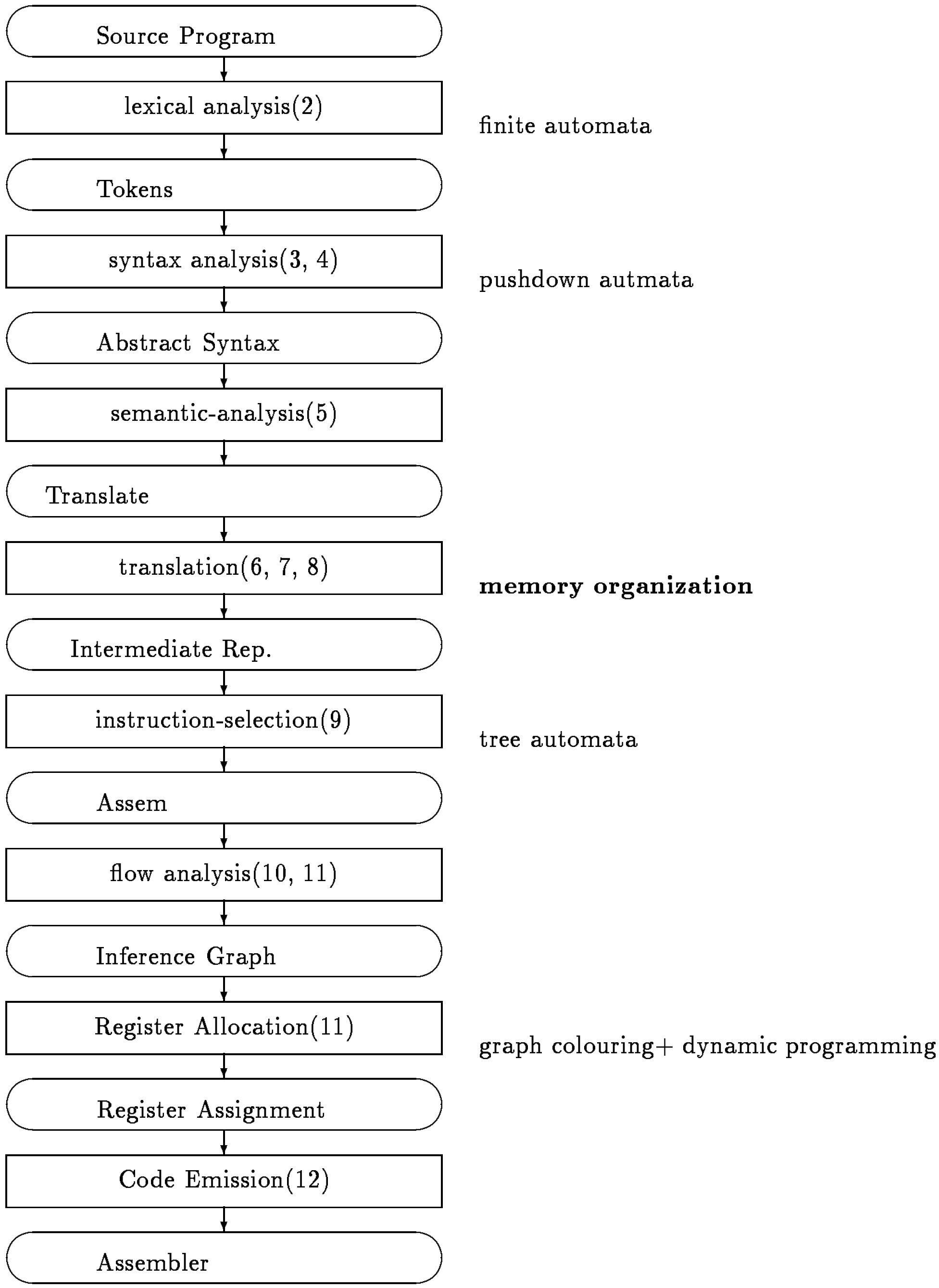
November 12, 2000

"http://www.math.tau.ac.il/∼ msagiv/courses/wcc.html"

1 Subjects

- The task of syntax analysis
- Automatic generation
- Error handling
- Context free grammars and derivations
- Ambiguous Grammars
- Predictive Parsing
- Bottom-up syntax analysis
- Bison — A parser generator

2 Basic Compiler Phases



3 Syntax Analysis(Parsing)

- Functionality

  Input Sequence of tokens

  Output Abstract Syntax Tree

- Report syntax errors, e,g., unbalanced parenthesizes
- Create "symbol-table" (sometimes)
- Create "'pretty-printed" version of the program (sometimes)
- In some cases the tree need not be generated (one-pass compilers)

4 Handling Syntax Errors

- Report and locate the error
- Diagnose the error
- Correct the error
- Recover from the error in order to discover more errors (without reporting too many "strange" errors)

5 Example

$$a := a * (b + c * d;$$

6 The Valid Prefix Property

- For every prefix tokens $t_1, t_2, \ldots, t_i$ that the parser identifies as legal there exists tokens $t_{i+1}, t_{i+2}, \ldots, t_n$ such that $t_1, t_2, \ldots t_n$ is a syntactically valid program
- If every token is considered as a single character:

  For every prefix word u that the parser identifies as legal there exists a word w such that uw is a valid program

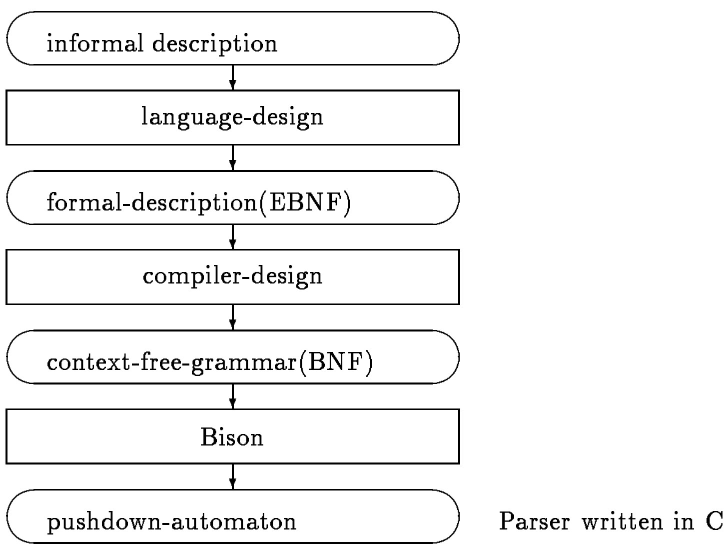
7 Error Diagnosis

- Line number (may be far from the actual error)
- The current token
- The expected tokens
- Parser configuration

8 Error Recovery

- Becomes less important in interactive environments
- Example heuristics:
  - Search for a semi-column and ignore the statement
  - Try to "replace" tokens for common errors
  - Refrain from reporting 3 subsequent errors
- Globally optimal solutions — For every input w, find a valid program w' with a "minimal-distance" from w

9 The General Picture



10 Example Context Free Grammar

1	$S \rightarrow S; S$
2	$S \rightarrow id := E$
3	$S \rightarrow print(L)$
4	$E \rightarrow id$
5	$E \rightarrow num$
6	$E \rightarrow E + E$
7	$E \rightarrow (S, E)$
8	$L \rightarrow E$
9	$L \rightarrow L, E$

11 Example Derivation

$$
\begin{array}{l}
\underline{S} \\
S\ ;\ \underline{S} \\
\underline{S}\ ;\ \mathrm{id} := E \\
\mathrm{id} := \underline{E}\ ;\ \mathrm{id} := E \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \underline{E} \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := E + \underline{E} \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \underline{E} + (S\ ,\ E\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\underline{S}\ ,\ E\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \underline{E}\ ,\ E\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := E + E\ ,\ \underline{E}\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \underline{E} + E\ ,\ \mathrm{id}\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \mathrm{num} + \underline{E}\ ,\ \mathrm{id}\ ) \\
\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \mathrm{num} + \mathrm{num}\ ,\ \mathrm{id}\ )
\end{array}
$$

DERIVATION 3.2.

From *Modern Compiler Implementation in ML,*

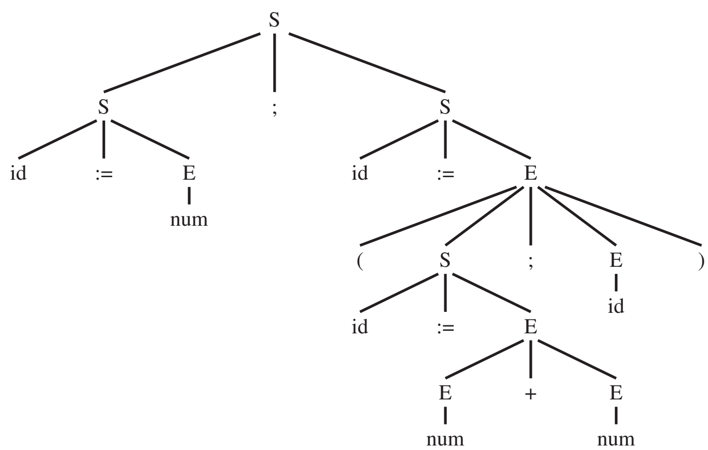


FIGURE 3.3. Parse tree.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

12 Ambiguous Grammars

- Two leftmost derivations for the same string
- Two rightmost derivations for the same string
- Two syntax trees for the same string

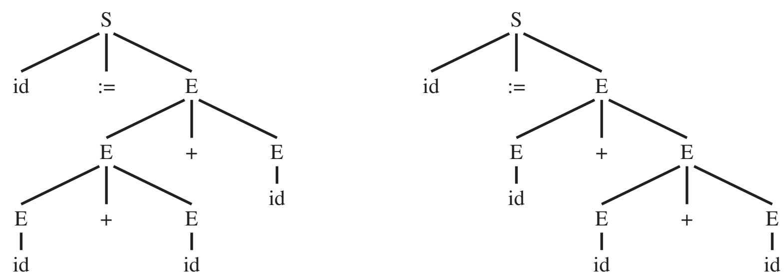


FIGURE 3.4. Two parse trees for the same sentence using Grammar 3.1.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

$$
\begin{aligned}
E &\rightarrow \mathrm{id} \\
E &\rightarrow \mathrm{num} \\
E &\rightarrow E * E \\
E &\rightarrow E \;/\; E \\
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow (\; E \;)
\end{aligned}
$$

GRAMMAR 3.5.

From *Modern Compiler Implementation in ML,*

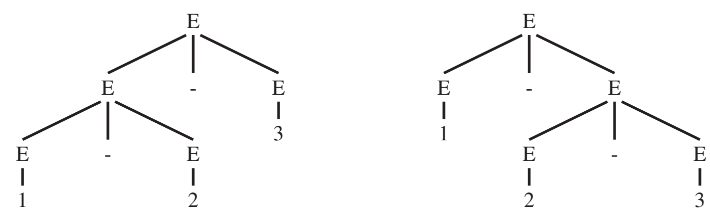


FIGURE 3.6. Two parse trees for the sentence `1-2-3` in Grammar 3.5.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

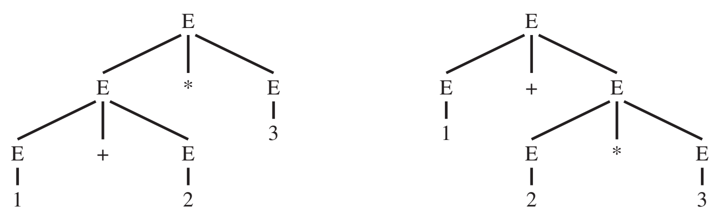


FIGURE 3.7. Two parse trees for the sentence `1+2*3` in Grammar 3.5.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

$E \to E + T$	$T \to T * F$	$F \to \mathrm{id}$
$E \to E - T$	$T \to T \,/\, F$	$F \to \mathrm{num}$
$E \to T$	$T \to F$	$F \to (E)$

GRAMMAR 3.8.

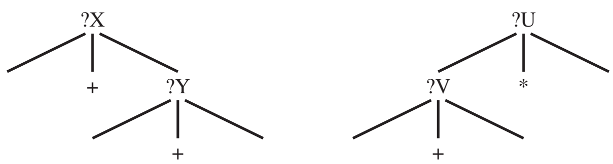


FIGURE 3.9. Parse trees that Grammar 3.8 will never produce.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

$$
\begin{array}{lll}
S \rightarrow E\ \$ & & \\
 & T \rightarrow T\ *\ F & F \rightarrow \mathrm{id} \\
E \rightarrow E\ +\ T & T \rightarrow T\ /\ F & F \rightarrow \mathrm{num} \\
E \rightarrow E\ -\ T & T \rightarrow F & F \rightarrow (\ E\ ) \\
E \rightarrow T & &
\end{array}
$$

GRAMMAR 3.10.

13 Top-Down (Predictive) Syntax Analysis

input

- A context free grammar
- A stream of tokens

output An abstract syntax tree or error

method

- Construct the syntax tree in a top-down manner
- Find the left most derivation
- Decide which production to apply scanning one token forward
- Report error as soon as the input is not a prefix of a valid program
- After an error ignore tokens until "the end of the construct" occurs

14 Example Grammar for Predictive Parsing

$$
\begin{array}{ll}
1 & S \rightarrow if\ E\,then\ S\,else\ S \\
2 & S \rightarrow beginS\ L \\
3 & S \rightarrow print(E) \\
4 & L \rightarrow end \\
5 & L \rightarrow ;\,S\ L \\
6 & E \rightarrow num = num
\end{array}
$$

15 C Parser Program

enum token IF, THEN, ELSE, BEGIN, END, PRINT,
SEMI, NUM, EQ, LP, RP;

```
extern enum token getToken(void);
void advance(void) { tok= getToken(); }
void eat(enum token t) { if (tok==t) advance(); else error(); }

void S(void) {switch(tok) {
            case IF: eat(IF); E(); eat(THEN); S();
                    eat(ELSE) S(); break;
            case BEGIN: eat(BEGIN); S(); L(); break;
            case PRINT: eat(PRINT); eat(LP); E(); eat(RP); break;
            default: error(tok, "Expecting if, begin, or print"); } }

void L(void) {switch(tok) {
            case END: eat(END); break;
            case SEMI: eat(SEMI); S(); L(); break;
            default: error(tok, "Expecting end or semicolumn"); } }

void E(void) {switch(tok) {
            case NUM: eat(NUM);eat(EQ) eat(NUM) break;
            default: error(tok, "Expecting a number"); } }
```

16 A Parser for arithmetic Expressions

Grammar

1	$E \to E + T$	2	$E \to T$
3	$T \to T * F$	4	$T \to F$
5	$F \to id$	6	$F \to (E)$

C code

```
void E(void) {switch(tok) {
            case ID: Both 1 and 2 can be applied here;
            case '(': Both 1 and 2 can be applied here;
            default:
            error(tok, "Expecting an identifier or left parenthesis"); } }
void T(void) {switch(tok) {
            case ID: Both 3 and 4 can be applied here;
            case '(': Both 3 and 4 can be applied here;
            default:
            error(tok, "Expecting an identifier or left parenthesis"); } }
void F(void) {switch(tok) {
            case ID: eat(ID); break;
            case '(': eat('('); E(); eat(')'); break;
            default:
            error('tok, 'Expecting an identifier or left parenthesis"); } }
```

17 Eliminating Left Recursion

Grammar

$$A \to A\beta|\gamma \quad (\gamma\beta^*)$$

Modified Grammar

$$\begin{array}{ll} 1 & A \to \gamma A' \\ 2 & A' \to \beta A' \\ 3 & A' \to \epsilon \end{array}$$

Arithmetic Grammar

$$\begin{array}{ll} 1 & E \to E + T \\ 2 & E \to T \\ 3 & T \to T * F \\ 4 & T \to F \\ 5 & F \to id \\ 6 & F \to (E) \end{array}$$

Modified Arithmetic Grammar

$$\begin{array}{llll} 1 & E \to TE' & & \\ 2 & E' \to +TE' & 3 & E' \to \epsilon \\ 4 & T \to FT' & & \\ 5 & T' \to *FT' & 6 & T' \to \epsilon \\ 7 & F \to id & & \\ 8 & F \to (E) & & \end{array}$$

18 C Code

```
void E(void) {switch(tok) {
            case ∈ {ID, '('}: T(); EP(); break;
            default:
            error(tok, "Expecting an identifier or left parenthesis"); } }
void T(void) {switch(tok) {
            case ∈ {ID, '('}: F(); TP(); break;
            default:
            error(tok, "Expecting an identifier or left parenthesis"); } }
void EP(void) {switch(tok) {
              case '+': eat('+'); T(); EP(); break;
              case ∈ { '$', ')' }: break;
              default:
              error(tok, "Expecting +, right-parenthesis, or eof"); } }
void F(void) {switch(tok) {
            case ID: eat(ID); break;
            case '(': eat('('); E(); eat(')'); break;
            default:
            error(tok, "Expecting an identifier or left parenthesis"); } }
```

```
void TP() {switch(tok) {
          case '*': eat('*'); F(); TP(); break;
          case ∈ { '+', ')', '$'}: break;
          default:
          error(tok, "Expecting *, +, right-parenthesis, or eof"); } }
```

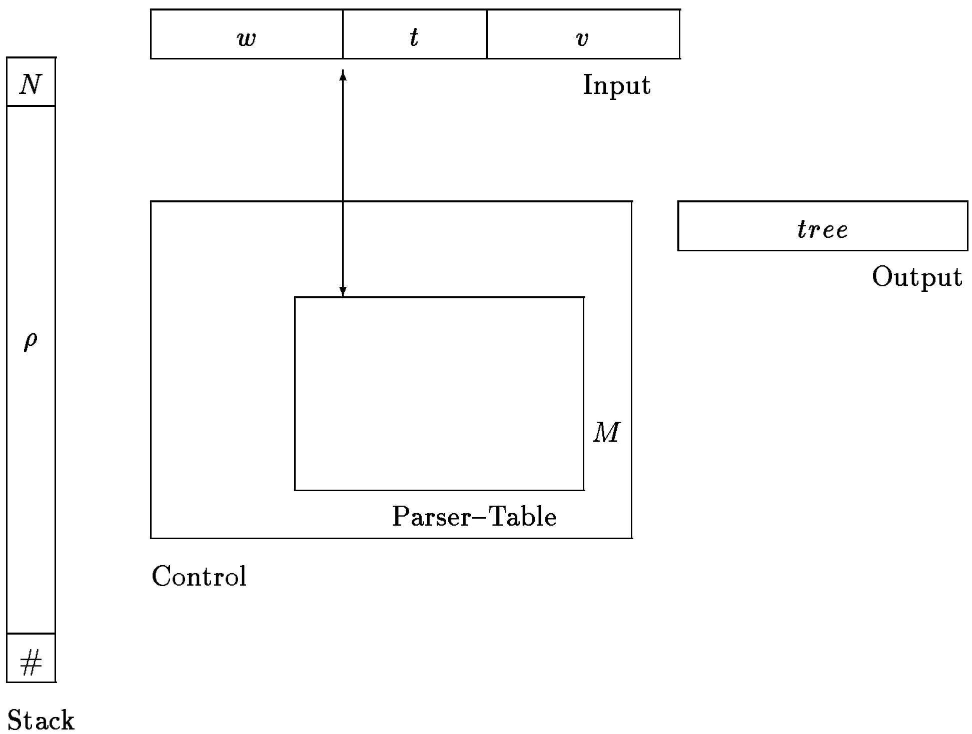
19 A Generic Solution

- For every production p, construct a set of tokens **SELECT**(p) which contains the tokens that may begin p
- A grammar is $LL(1)$ if for every two productions $p_1\colon N \to \alpha_1$ and $p_2\colon N \to \alpha_2$, $\mathbf{SELECT}(p_1) \cap \mathbf{SELECT}(p_2) = \phi$
- Sometimes a non LL(1) grammar can be rewritten to become LL(1)

20 The Generic Code for a Procedure N

void N()
{ switch (tok) {
  case $\in \mathbf{SELECT}(p_1\colon N \to \alpha_1)$: "apply" $N \to \alpha_1$;
  case $\in \mathbf{SELECT}(p_2\colon N \to \alpha_2)$: "apply" $N \to \alpha_2$;
  $\vdots$
  case $\in \mathbf{SELECT}(p_n\colon N \to \alpha_n)$: "apply" $N \to \alpha_n$;
  default:
  error("expecting:
        $\mathbf{SELECT}(p_1) \cup \mathbf{SELECT}(p_2) \cup \cdots \cup$
        $\mathbf{SELECT}(p_n)$ "); }

21 Explicit Stack Deterministic Pushdown Automaton



22 The *FIRST* Sets

- A production $N \to \alpha$ is applicable for tokens that "begin" α
- Arithmetic Example:
  - The production $F \to id$ is applied for the token 'id'
  - The production $E \to TE'$ is applied for tokens '(' and 'id'
- A formal definition $\mathbf{FIRST}(\alpha) = \{t \mid t \in \mathit{Tokens}, \alpha \overset{*}{\Longrightarrow} t\gamma\}$
- A recursive definition - The smallest sets such that
  - For every terminal Z, $Z \in \mathbf{FIRST}(Z)$
  - For every production $p\colon X \to Y_1 Y_2 \cdots Y_{j-1} Y_j Y_{j+1} Y_{j+2} \cdots Y_k$ such that $Y_1, Y_2, \cdots Y_{j-1}$ is nullable, $\mathbf{FIRST}(Y_j) \subseteq \mathbf{FIRST}(X)$

- An algorithm

> **for** each non-terminal Z, $\mathbf{FIRST}[Z] := \phi$
> **for** each terminal Z, $\mathbf{FIRST}[Z] := \{Z\}$
> **repeat**
>   **for** each production p: $X \rightarrow Y_1 Y_2 \cdots Y_k$
>   **for** each j from 1 to k s.t., $Y_1, Y_2, \ldots, Y_{j-1} \in \mathit{nullable}$
> $\mathbf{FIRST}[X] := \mathbf{FIRST}[Y_j] \cup \mathbf{FIRST}[X]$
> **until** no more changes occur

23 Computing FIRST for Arithmetic Expressions

0	$S \rightarrow E\$$	1	$E \rightarrow TE'$
2	$E' \rightarrow +TE'$	3	$E' \rightarrow \epsilon$
4	$T \rightarrow FT'$		
5	$T' \rightarrow *FT'$	6	$T' \rightarrow \epsilon$
7	$F \rightarrow id$	8	$F \rightarrow (E)$

$$\mathit{nullable} = \{T', E'\}$$

Prod	S	E	E'	T	T'	F
8						(
7						$(, id$
6						$(, id$
5					$*$	$(, id$
4				$(, id$	$*$	$(, id$
3				$(, id$	$*$	$(, id$
2			$+$	$(, id$	$*$	$(, id$
1		$(, id$	$+$	$(, id$	$*$	$(, id$
0	$(, id$	$(, id$	$+$	$(, id$	$*$	$(, id$
$8, 7, 6, 5, 4, 3, 2, 1, 0$	$(, id$	$(, id$	$+$	$(, id$	$*$	$(, id$

24 The *FOLLOW* Sets

- A production $N \rightarrow \epsilon$ is applicable for tokens that "can follow" N in some derivation
- Arithmetic Example:

- The production $E' \to \epsilon$ is applied for tokens '\$' and ')'
- The production $T' \to \epsilon$ is applied for tokens '\$', ')' and '+'

- Formal definition: $\mathbf{FOLLOW}(N) = \{t \in \mathit{Tokens} | \exists \alpha, \gamma : S \stackrel{*}{\Longrightarrow} \alpha N t \gamma\}$

- A recursive definition - The smallest sets such that

for each production $p\colon X \to Y_1 Y_2 \cdots Y_k$
  for each i from 1 to k
    if $Y_{i+1}, Y_{i+2}, \ldots, Y_k$ is nullable **then**
      $\mathbf{FOLLOW}(X) \subseteq \mathbf{FOLLOW}(Y_i)$
    for each j from $i+1$ to k s.t. $Y_{i+1}, Y_{i+2}, \ldots, Y_{j-1}$ is nullable
      $\mathbf{FIRST}(Y_j) \subseteq \mathbf{FOLLOW}(Y_i)$

- An algorithm

repeat
  for every grammar symbol Z, $\mathbf{FOLLOW}[Z] := \phi$
  for each production $p\colon X \to Y_1 Y_2 \cdots Y_k$
    for each i from 1 to k
      if $Y_{i+1}, Y_{i+2}, \ldots, Y_k \in \mathit{nullable}$ **then**
        $\mathbf{FOLLOW}[Y_i] := \mathbf{FOLLOW}[X] \cup \mathbf{FOLLOW}[Y_i]$
      for each j from $i+1$ to k s.t., $Y_{i+1}, Y_{i+2}, \ldots, Y_{j-1} \in \mathit{nullable}$
        $\mathbf{FOLLOW}[Y_i] := \mathbf{FIRST}[Y_j] \cup \mathbf{FOLLOW}[Y_i]$
  until no more changes occur

25 Computing FOLLOW sets for Arithmetic Expressions

0	$S \to E\$$	1	$E \to TE'$
2	$E' \to +TE'$	3	$E' \to \epsilon$
4	$T \to FT'$		
5	$T' \to *FT'$	6	$T' \to \epsilon$
7	$F \to id$	8	$F \to (E)$

$$\mathit{nullable} = \{T', E'\}$$

Prod	S	E	E'	T	T'	F
0		$\$$				
1		$\$$	$\$$	$\$, +$		
2		$\$$	$\$$	$\$, +$		
3		$\$$	$\$$	$\$, +$		
4		$\$$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
5		$\$$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
6		$\$$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
7		$\$$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
8		$\$,)$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
0		$\$,)$	$\$$	$\$, +$	$\$, +$	$\$, +, *$
1		$\$,)$	$\$,)$	$\$, +,)$	$\$, +$	$\$, +, *$
2, 3		$\$,)$	$\$,)$	$\$, +,)$	$\$, +$	$\$, +, *$
4		$\$,)$	$\$,)$	$\$, +,)$	$\$, +,)$	$\$, +, *,)$
5, 6, 7, 8, 0, 1, 2, 3, 4		$\$,)$	$\$,)$	$\$, +,)$	$\$, +,)$	$\$, +, *,)$

26 The SELECT sets

- A production $p: N \rightarrow \alpha$ is applicable for tokens in **SELECT**(p) where

$$\textbf{SELECT}(p) = \left\{ \begin{array}{ll} \textbf{FIRST}(\alpha) \cup \textbf{FOLLOW}(N) & \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ \textbf{FIRST}(\alpha) & \text{otherwise} \end{array} \right\}$$

- An algorithm for **SELECT**

> **for** each production $p: X \rightarrow Y_1 Y_2 \cdots Y_k$
>   **SELECT**$[p] := \phi$
>   **for** each i from 1 to k s.t.,$Y_1, Y_2, \ldots, Y_{i-1} \in \mathit{nullable}$
>   **SELECT**$[p]$:= **FIRST**$[Y_i] \cup$ **SELECT**$[p]$
>   **if** $Y_1, Y_2, \ldots, Y_k \in \mathit{nullable}$ **then**
>   **SELECT**$[p]$:= **FOLLOW**$[X] \cup$ **SELECT**$[p]$

- LL(1) grammar — For every non-terminal N and productions $p_1: N \rightarrow \alpha_1, p_1: N \rightarrow \alpha_2$,

$$\textbf{SELECT}(p_1) \cap \textbf{SELECT}(p_2) = \phi$$

27 The SELECT for Arithmetic Expressions

production	**tokens**
$S \to E\$'$	$\{(, id\}$
$E \to TE'$	$\{(, id\}$
$E' \to +TE'$	$\{+\}$
$E' \to \epsilon$	$\{), \$\}$
$T \to FT'$	$\{(, id\}$
$T' \to *FT'$	$\{*\}$
$T' \to \epsilon$	$\{+,), \$\}$
$F \to id$	$\{id\}$
$F \to (E)$	$\{(\}$

28 A Possible error recovery

- After an error occurs at procedure N skip all the tokens until a token in **FOLLOW**(N)

29 Error Recovery for Arithmetic Expressions

```
void E() {switch(tok) {
        case ∈ { ID, '('}: T(); E'(); break;
        default:
        error(tok, "Expecting an identifier or left parenthesis");
        skipTo({'$', ')' }) } }
void T() {switch(tok) {
        case ∈ { ID, '('}: F(); T'(); break;
        default:
        error(tok, "Expecting an identifier or left parenthesis");
        skipTo({'+', '$', ')' }) } }
void E'() {switch(tok) {
        case '+': eat('+'); T(); E'(); break;
        case ∈ { '$', ')'}: break;
        default:
        error(tok, "Expecting +, right-parenthesis, or eof");
        skipTo({'$', ')' }) } }
void F() {switch(tok) {
        case ID: eat(ID); break;
        case '(': eat('('); E(); eat(')'); break;
        default:
        error(tok, "Expecting an identifier or left parenthesis");
        skipTo({'*', '+', '$', ')' }) } }
void T'() {switch(tok) {
        case '*': eat('*'); F(); T'(); break;
```

case ∈ { '+', ')', '$'}: break;
default:
error(tok, "Expecting *, +, right-parenthesis, or eof");
skipTo({'+', '$', ')' }) } }

30 Extensions

- Regular Expressions in the right hand side
- $LL(k)$

31 Interesting Non $LL(1)$ Grammars

- Left-recursive — $N \stackrel{+}{\Longrightarrow} N\alpha$
- Common Prefix — $S \rightarrow id := Exp|id(Exp)$
- Ambiguous:
  - Ambiguous arithmetic expressions
  - Dangling-else