### **COP 3402 Systems Software**

# Predictive Parsing (First and Follow Sets)

#### **Outline**

- 1. First Set
- 2. Nullable Symbols
- 3. Follow Set
- 4. Predictive Parsing Table
- 5. LL(1) Parsing

A recursive descent (or predictive) parser chooses the correct production by looking a fixed number of symbols ahead (typically one symbol or token).

#### First set:

Let **X** be any string of grammar symbols (terminals and non-terminals).

First(X) is defined to be the set of terminals that begin strings derived from X.

Definition: FIRST(X) = {  $\mathbf{t} \mid \mathbf{X} ==> * \mathbf{t} \mathbf{Z}$  for some  $\mathbf{Z}$ }  $\cup \{ \mathbf{\varepsilon} \mid \text{if } \mathbf{X} ==> * \mathbf{\varepsilon} \}$ 

If  $X \rightarrow A B C$ , then FIRST(X) = FIRST(A B C) and is computed as follows:

A is a terminal  $FIRST(X) = FIRST(A B C) = \{A\}$ 

For instance, if  $X \rightarrow t$  B C, then FIRST(X) = FIRST(t B C) = { t }

A is a non-terminal and A does not derive to  $\varepsilon$  FIRST(X) = FIRST(A B C) = FIRST(A)

A is a non-terminal and A derives to  $\varepsilon$  FIRST(X) = FIRST(A B C) = FIRST(A) – { $\varepsilon$ }  $\cup$  FIRST(BC)

Similarly, for FIRST(BC) we have:

B is a terminal  $FIRST(BC) = \{B\}$ 

B is a non-terminal and B does not derive to  $\varepsilon$  FIRST(BC) = FIRST(B)

B is a non-terminal and B derives to  $\varepsilon$  FIRST(BC) = FIRST(B) -  $\{\varepsilon\}$   $\cup$  FIRST(C)

And so on...

```
Example:
S \rightarrow ABC|CbB|Ba
A \rightarrow da \mid BC
B \rightarrow g \mid \epsilon
C \rightarrow h \mid \epsilon
FIRST(S) = FIRST(A B C) \cup FIRST(C b B) \cup FIRST(B a)
FIRST(A) = FIRST(d \ a) \cup First(B \ C) = \{ d \} \cup FIRST(B \ C)
FIRST(B) = FIRST(g) \cup First \{ \epsilon \} = \{ g, \epsilon \}
FIRST(C) = FIRST(h) \cup First \{ \epsilon \} = \{ h, \epsilon \}
Now we can compute:
FIRST(BC) = FIRST(B) - \{ \epsilon \} \cup \{ h, \epsilon \} = \{ g, \epsilon \} - \{ \epsilon \} \cup \{ h, \epsilon \} = \{ g, h, \epsilon \}
and
FIRST(A) = \{ d \} \cup \{ g, h, \varepsilon \} = \{ d, g, h, \varepsilon \}
```

Exercise: Compute FIRST(C b B) and FIRST(B a) in order to compute FIRST(S)

Example: Given the following expression grammar:

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

```
First(E + T) = { id, ( }

Because: E + T \rightarrow T + T \rightarrow F + T \rightarrow id + T

E + T \rightarrow T + T \rightarrow F + T \rightarrow (E) + T

First(E) = { id, ( }

Because: E \rightarrow T \rightarrow F \rightarrow id

E \rightarrow T \rightarrow F \rightarrow (E)
```

# **Nullable Symbols**

Nullable symbols those that produce the empty ( $\varepsilon$ ) string.

Example: Given the following grammar, find the nullable symbols and the FIRST sets:

$$Z \rightarrow d$$

$$Y \rightarrow \epsilon$$

$$X \rightarrow Y$$

$$Z \rightarrow X Y Z$$
  $Y \rightarrow c$ 

$$Y \rightarrow c$$

$$X \rightarrow a$$

Note that if X can derive the empty string, nullable(X) is true.

$$X \rightarrow Y \rightarrow \epsilon$$

$$3 \leftarrow Y$$

$$Z \rightarrow d$$

$$Z \rightarrow X Y Z$$

	Nullable	First
X	Yes	$\{a,c,\epsilon\}$
Y	Yes	{ c, ε }
Z	No	{ a, c, d }

#### **Follow set**

FOLLOW(A) = { 
$$\mathbf{t} \mid \mathbf{S} ==> *\alpha \mathbf{A} \mathbf{t} \mathbf{\omega}$$
 for some  $\alpha, \mathbf{\omega}$ }

Given a non-terminal A, FOLLOW( A ) is the set of terminal symbols that can immediately follow A.

Example 1: If there is a derivation containing At, then t is in FOLLOW(A) = t.

Example 2: If there is a derivation containing A B C t and B and C are nullable, then t is in FOLLOW(A).

Example 3: The FIRST / FOLLOW sets and nullable symbols for the following grammar are:

$$Z \rightarrow d$$
  $Y \rightarrow \varepsilon$   $X \rightarrow Y$   $Z \rightarrow X Y Z$   $Y \rightarrow c$   $X \rightarrow a$ 

	Nullable	FIRST	FOLLOW
X	Yes	{ a, c, ε }	{ a, c, d }
Υ	Yes	{ C, <b>&amp;</b> }	{ a, c, d }
Z	No	{ a, c, d }	{ }

#### Method to construct the predictive parsing table

For each production  $A \rightarrow \alpha$  of the grammar, do the following:

- 1. Add  $A \rightarrow \alpha$  to m[A,t] for each terminal t in FIRST( $\alpha$ ).
- 2. If nullable( $\alpha$ ) is true, add  $A \rightarrow \alpha$  to m[A, t] for each t in FOLLOW(A).

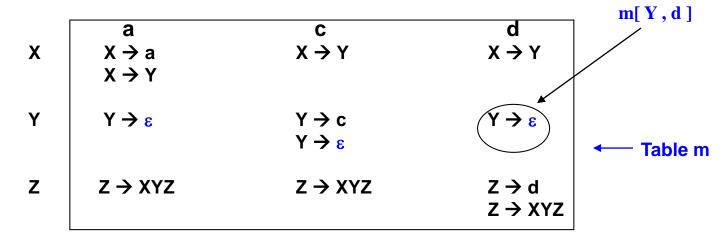
#### **Example: Given the grammar:**

$$Z \rightarrow d$$
  
 $Z \rightarrow X Y Z$ 

$$Y \rightarrow \varepsilon$$
  
 $Y \rightarrow c$ 

$$X \rightarrow Y$$

$$Z \rightarrow X Y Z$$



#### **Example: Given the grammar:**

```
S \rightarrow E\$
E \rightarrow E + T
T \rightarrow T * F
F \rightarrow id
E \rightarrow T
T \rightarrow F
F \rightarrow (E)
We can rewrite the grammar to avoid left recursion obtaining thus:
S \rightarrow E\$
E \rightarrow TE'
T \rightarrow FT'
F \rightarrow id
E' \rightarrow + TE'
T' \rightarrow * FT'
F \rightarrow (E)
E' \rightarrow \epsilon
T' \rightarrow \epsilon
```

Compute First, Follow, and nullable.

	Nullable	First	Follow
Е	No	{ id , ( }	{ <b>), \$</b> }
E'	Yes	{ +, ε }	{ <b>), \$</b> }
Т	No	{ <b>id</b> , <b>(</b> }	{ <b>)</b> , <b>+, \$</b> }
T'	Yes	{ *, ε }	{ <b>)</b> , <b>+, \$</b> }
F	No	{ <b>id</b> , <b>(</b> }	{ <b>)</b> , * , +, \$ }

#### Parsing table for the expression grammar:

	+	*	id	(	)	\$
E			E → T E'	E → T E'		
E'	E' → +T E'				E' <b>&gt;</b> ε	Ε' → ε
T			<b>T → F T'</b>	T → F T'		
T'	Τ' → ε	T' → *F T'			Τ' → ε	Τ' → ε
F			F → id	$\mathbf{F} \rightarrow (\mathbf{E})$		

Using the predictive parsing table, it is easy to write a recursive-descent parser:

```
void Tprime() {
  switch (token) {
   case PLUS : break;
   case TIMES : accept(TIMES); F(); Tprime(); break;
   case RPAREN : break;
   default : error();
  }
}
```

## Left factoring

Another problem that we must avoid in predictive parsers is when two productions for the same non-terminal start with the same symbol.

Example:  $S \rightarrow \text{if } E \text{ then } S$ 

 $S \rightarrow if E then S else S$ 

Solution: Left-factor the grammar. Take allowable ending "else S" and  $\epsilon$ , and make a new production (new non-terminal) for them:

 $S \rightarrow if E then S X$ 

 $X \rightarrow else S$ 

 $X \rightarrow \epsilon$ 

Grammars whose predictive parsing tables contain no multiples entries are called LL(1).

The first L stands for left-to-right parse of input string. (input string scanned from left to right)

The second L stands for leftmost derivation of the grammar

The "1" stands for one symbol lookahead

## **Left Factoring**

The following (unambiguous) grammar for arithmetic expressions is not LL(1):

```
E -> E + E | T
T -> T * F | F
F -> id | num | ( E )
```

We obtain an LL(1) by using a grammar transformation called left factoring:

```
E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> * F T' | ε

F -> id | num | ( E )
```

#### **Left Recursive Grammars**

A grammar is called left recursive if there is a derivation A -> A a for some string a and some non-terminal symbol.

Left recursive grammars are not suitable for LL(k) parsers.

## **Left Factoring**

Left factoring is a grammar transformation that eliminates left recursion.

For example, the pair

$$A \rightarrow A a \mid b$$

could be replaced by the following two non-left-recursive productions:

$$A \rightarrow b A'$$
  
 $A' \rightarrow a A' \mid \epsilon$ 

# A Non-LL(1) Grammar

For instance, a grammar having a production such as

$$A \rightarrow a b_1 \mid a b_2$$

is not suitable for an LL(1) parser.

If the parser looks only one token ahead and sees the token a, then it cannot determine which choice of the alternation to follow.

## **Left Factoring**

Using again left factoring, the production

$$A \rightarrow a b_1 \mid a b_2$$

can be left-factored to the following two productions:

A 
$$\rightarrow$$
 a A'  
A'  $\rightarrow$  b<sub>1</sub> | b<sub>2</sub>

#### **Example: Given the grammar:**

```
S \rightarrow E\$

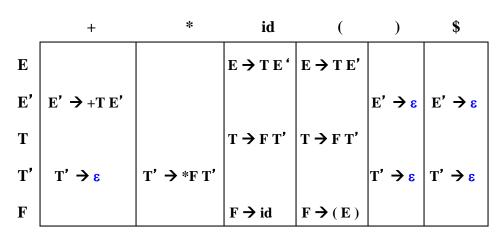
E \rightarrow TE' T \rightarrow FT' F \rightarrow id

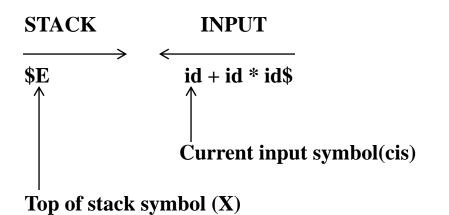
E' \rightarrow +TE' T' \rightarrow *FT' F \rightarrow (E)

E' \rightarrow \epsilon T' \rightarrow \epsilon
```

With the following First, Follow, and nullable.

	Nullable	First	Follow
S	No	{ <b>id</b> }	
E	No	{ <b>id</b> , <b>(</b> }	{ <b>), \$</b> }
E'	Yes	{ <b>+</b> }	{ <b>), \$</b> }
Т	No	{ <b>id</b> , <b>(</b> }	{ <b>)</b> , <b>+, \$</b> }
T'	Yes	{ * }	{ <b>)</b> , <b>+, \$</b> }
F	No	{ <b>id</b> , <b>(</b> }	{ <b>)</b> , * , +, \$ }





A nonrecursive predictive parser can also be implemented by using a stack instead of recursively calling procedures.

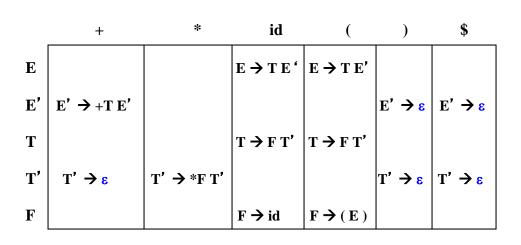
This approach is called table driven.

To implement it we need:

- 1) As input a string "w".
- 2) A parsing table.
- 3) A stack.

**Initial configuration:** 

- 1)The string w\$ in the input buffer
- 2) The start symbol S on top of the stack, above the end of file symbol \$.



#### 

#### **Algorithm:**

```
push $ onto the stack
push start symbol E onto the stack
repeat { /*stack not empty */
  if (X == cis) {
    pop the stack;
    advance cis to next symbol;
  elseif (X is terminal) error();
  elseif (M[X, cis] is error entry) error();
  elseif (M[X, cis] is production) {
    pop the stack;
    push the right hand side of
    the production in reverse order;
  let X point to the top of the stack.
until (X == \$);
```

Stack	Input	Production	Algorithm:
\$E \$E'T' \$E'T'F \$E'T'id \$E'T' \$E'T'F \$E'T'F \$E'T'F* \$E'T'F \$E'T'F \$E'T'G \$E'T'	id + id * id\$ + id * id\$ + id * id\$ + id * id\$  * id\$  * id\$  * id\$  * id\$  * id\$	$E \rightarrow TE'$ $T \rightarrow FT'$ $F \rightarrow id$ match $id$ $T' \rightarrow \varepsilon$ $E' \rightarrow +TE'$ match $+$ $T \rightarrow FT'$ $F \rightarrow id$ match $id$ $T' \rightarrow *FT'$ match $*$ $F \rightarrow id$ match $id$ $T' \rightarrow \varepsilon$ $E' \rightarrow \varepsilon$	<pre>push \$ onto the stack push start symbol E onto the stack  repeat { /* stack not empty*/   if (X == cis) {     pop the stack;     advance cis to next symbol;   }   elseif (X is terminal) error();   elseif (M[X, cis] is error entry) error();   elseif (M[X, cis] is production) {     pop the stack;     push the right hand side of       the production in reverse order;     }   let X point to the top of the stack. } until (X == \$);</pre>

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or  $\epsilon$  can be added to any FIRST set.

- 1. If X is a terminal, then FIRST(X) = { X }.
- 2. If X is a non-terminal and  $X \rightarrow Y$

# **COP 3402 Systems Software**

# Predictive Parsing (First and Follow Sets)

The End