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 looking ahead at the input string a fix number of symbols (typically one symbol or token).

First set:

Let **X** be a string of terminals and non-terminals.

First(\mathbf{X}) is the set of terminals that can begin strings or sequence derivable from \mathbf{X} .

which means that we are interested in knowing if a particular nonterminal **X** derives a string with a terminal **t**.

```
Definition: FIRST(X) = { \mathbf{t} \mid \mathbf{X} ==> * \mathbf{t} \mathbf{\omega} for some \mathbf{\omega}} \cup {\mathbf{\varepsilon} \mid \mathbf{X} ==> * \mathbf{\varepsilon}}
If X \rightarrow A B C then FIRST(X) = FIRST(A B C) and is computed as follows:
If A is a terminal.
FIRST(X) = FIRST(A B C) = {A} (for example, if X \rightarrow t B C, FIRST (X) = FIRST(t B C) = { t }
Otherwise, if X does not derive to an empty string,
FIRST(X) = FIRST(A B C) = FIRST(A).
If FIRST(A) contains the empty string then,
FIRST(X) = FIRST(A B C) = FIRST(A) - \{ \epsilon \} \cup FIRST(BC)
Similarly, for FIRST(BC) we have:
FIRST(BC) = \{B\} \text{ if B is a terminal,}
Otherwise, if B does not derive to an empty string,
FIRST(BC) = FIRST(B)
If FIRST(B) contains the empty string then,
FIRST(BC) = FIRST(B) - \{ \epsilon \} \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 are the ones that produce the empty (ε) string

Example: Given the following grammar, find the nullabel symbols and the First set:

$$Z \rightarrow d$$

$$Y \rightarrow \varepsilon$$

$$X \rightarrow Y$$

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

$$Y \rightarrow c$$

$$X \rightarrow a$$

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

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

$$Y \rightarrow \epsilon$$

$$Z \rightarrow d$$

$$Z \rightarrow X Y Z$$

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

Follow set

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

Given a production A, Follow(A) is the set of terminals symbols that can immediately follow A.

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

Example 2: If the derivation contains A B C t and B and C are nullables, t is in Follow (A).

Example 3: Given the following grammar:

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

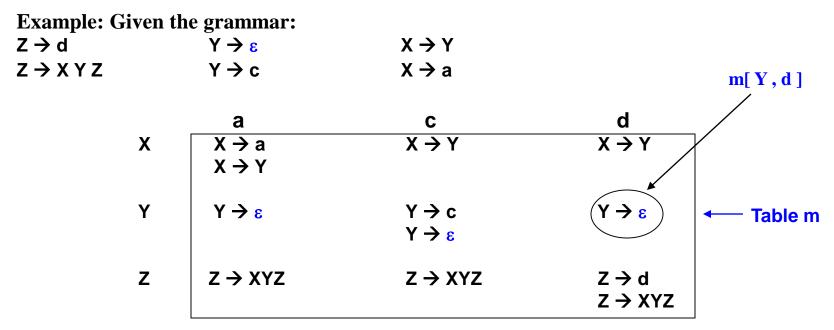
Compute First, Follow, and nullable.

<u> </u>	ato i noti i onoti, c	ATTO TTOTICADIOT	
	Nullable	First	Follow
X	Yes	{ a, c, ε }	{ a, c, d }
Υ	Yes	{c, ε }	{ a, c, d }
Z	No	{a, c, d }	{ \$ ←EOF}

Method to construct the predictive parsing table

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

- 1.- For each terminal t in First (A), add $A \rightarrow \alpha$ to m[A, t], where m is the table.
- 2.- If nullable(α) is true, add the production $A \rightarrow \alpha$ in row A, column t, for each t in Follow(A).



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 , (}	{), \$ }
E'	Yes	{ +, ε }	{), \$ }
Т	No	{ id , (}	{) , +, \$ }
T'	Yes	{ *, ε }	{) , +, \$ }
F	No	{ id , (}	{) , * , +, \$ }

Parsing table for the expression grammar:

	+	*	id	()	\$
E			E → T E'	E → T E'		
E '	E' → +T E'				E' > ε	Ε' → ε
T			T → F T'	T → F T'		
T'	T' → 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:

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: $S \rightarrow if E then S$

 $S \rightarrow If E then S else S$

Solution: Left-factor the grammar. Take allowable ending "else S" and e, 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

Example: Given the grammar:

```
S \rightarrow E\$

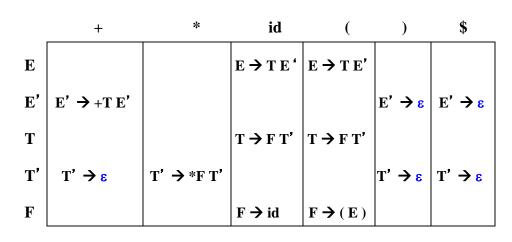
E \rightarrow TE' T \rightarrow FT' F \rightarrow id

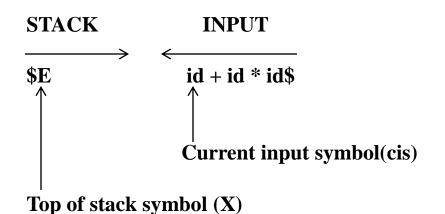
E' \rightarrow +TE' T' \rightarrow *FT' F \rightarrow (E)

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

With the following First, Follow, and nullable.

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





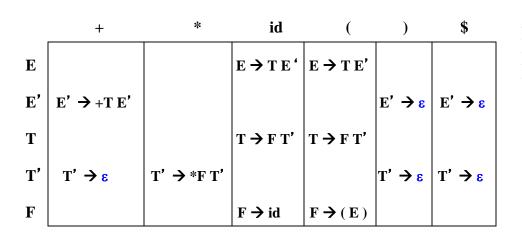
A nonrecursive predictive parser can be implemented using a stack instead of via recursive procedures calls. This approach is called table driven.

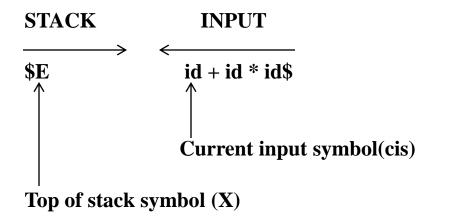
To implement it we need:

- 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 a terminal) error();
   elseif (M[X, cis] is an error entry) error();
   elseif (M[X,cis] = nonterminal) {
            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 = = \$) \{accept\}
```

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

COP 3402 Systems Software

Predictive Parsing (First and Follow Sets)

The End