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

First set

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(**X**) is the set of terminals that can begin strings or sequence derivable from **X.**

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

First set

Definition: FIRST(**X**) = { **t** | **X** ==> * **t** **ω** for some **ω**} ∪ {**ε** | **X** ==> * **ε** }

If X → 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 → 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) – { **ε** } ∪ FIRST(BC)

Similarly, for FIRST(BC) we have:
FIRST(BC) = {B} 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) – { **ε** } ∪ FIRST(C)

And so on…

First set

Example:

$S \rightarrow A\ B\ C \mid C\ b\ B \mid B\ a$

$A \rightarrow d\ a \mid B\ C$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

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 {ε } = {g, ε }

FIRST(C) = FIRST(h) $\cup$ First {ε } = {h, ε }

Now we can compute:

FIRST(BC) = FIRST(B) – {ε } $\cup$ {h, ε } = {g, ε } – { ε } $\cup$ {h, ε } = {g, h, ε }

and

FIRST(A) = { d } $\cup$ {g, h, ε } = { d, g, h, ε }

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

First set

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 → T + T → F + T → id + T
E + T → T + T → F + T → (E) + T

First(E) = { id, (}

Because: E → T → F → id
E → T → F → (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 → d	Y → ε	X → Y
Z → X Y Z	Y → c	X → a

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

X → Y → ε

Y → ε

Z → d

Z → X Y Z

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

Follow set

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

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 → d$ **Y → ε** **X → Y**
Z → X Y Z **Y → c** **X → a**

Compute First, Follow, and nullable.

	Nullable	First	Follow
X	Yes	{ a, c, ε }	{ a, c, d }
Y	Yes	{c, ε}	{ a, c, d }
Z	No	{a, c, d }	{$ ←EOF}

Predictive parsing table

<u>Method to construct the predictive parsing table</u>

For each production A → α of the grammar, do the following:

1.- For each terminal t in First (A), add A → α to m[A , t], where m is the table.

2.- If nullable(α) is true, add the production A → α in row A, column t, for each t in Follow(A).

Example: Given the grammar:

Z → d	Y → ε	X → Y
Z → X Y Z	Y → c	X → a

Table m

	a	c	d
X	X → a X → Y	X → Y	X → Y
Y	Y → ε	Y → c Y → ε	Y → ε
Z	Z → XYZ	Z → XYZ	Z → d Z → XYZ

m[Y , d]

Predictive parsing table

Example: Given the grammar:

S → E$

E → E + T	T → T * F	F → **id**
E → T	T → F	F → **(** E **)**

We can rewrite the grammar to avoid left recursion obtaining thus:

S → E$

E → T E’	T → F T’	F → **id**
E’ → **+** T E’	T’ → ***** F T’	F → **(** E **)**
E’ → **ε**	T’ → **ε**	

Compute First, Follow, and nullable.

	Nullable	First	Follow
E	No	{ **id** , **(** }	{ **)**, **$** }
E’	Yes	{ **+**, **ε** }	{ **)**, **$** }
T	No	{ **id** , **(** }	{ **)** , **+**, **$** }
T’	Yes	{ *****, **ε** }	{ **)** , **+**, **$** }
F	No	{ **id** , **(** }	{ **)** , ***** , **+**, **$** }

Predictive parsing table

Parsing table for the expression grammar:

	+	*	id	(	)	$
E			**E → T E'**	**E → T E'**		
E'	**E' → +T E'**				**E' → ε**	**E' → ε**
T			**T → F T'**	**T → F T'**		
T'	**T' → ε**	**T' → *F T'**			**T' → ε**	**T' → ε**
F			**F → id**	**F → (E)**		

Predictive parsing table

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

	+	*	id	(	)
T'	T' $\rightarrow \varepsilon$	T' $\rightarrow$ *FT'			T' $\rightarrow \varepsilon$

```
Void Tprime (void) { swith  (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 → **if** E **then** S
S → **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 → **if** E **then** S X
X → **else** S
X → ε

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

Nonrecursive predictive parsing

Example: Given the grammar:

S → E$
E → T E'
E' → **+** T E'
E' → **ε**

T → F T'
T' → ***** F T'
T' → **ε**

F → **id**
F → **(** E **)**

With the following First, Follow, and nullable.

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

Nonrecursive predictive parsing

	+	*	id	(	)	$
E			E → T E'	E → T E'		
E'	E' → +T E'				E' → ε	E' → ε
T			T → F T'	T → F T'		
T'	T' → ε	T' → *F T'			T' → ε	T' → ε
F			F → id	F → (E)		

STACK **INPUT**

$E **id + id * id$**

Current input symbol(cis)

Top of stack symbol (X)

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 $.

Nonrecursive predictive parsing

	+	*	id	(	)	$
E			**E → T E'**	**E → T E'**		
E'	**E' → +T E'**				**E' → ε**	**E' → ε**
T			**T → F T'**	**T → F T'**		
T'	**T' → ε**	**T' → *F T'**			**T' → ε**	**T' → ε**
F			**F → id**	**F → (E)**		

STACK **INPUT**

$E **id + id * id$**

Current input symbol(cis)

Top of stack symbol (X)

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}

Nonrecursive predictive parsing

Stack	Input	Production
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	match id
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	match +
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	match id
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	match *
$E'T'id	id$	F → id
$E'T'	$	match id
$E'	$	T' → ε
$	$	E' → ε

Algorithm:

```
push $ onto the stack
push start symbol E onto the stack
repeat (X != $){ /*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}
else {error() }
```

Predictive Parsing (First and Follow Sets)

The End