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 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.**

First set

Definition: FIRST(**X**) = { **t** | **X** ==> * **t** **Z** for some **Z**} $\cup$ {**ε** | if **X** ==> * **ε** }

If X → 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 → t B C, then	FIRST(X) = FIRST(t B C) = { t }
A is a non-terminal and A does not derive to ε	FIRST(X) = FIRST(A B C) = FIRST(A)
A is a non-terminal and A derives to ε	FIRST(X) = FIRST(A B C) = FIRST(A) – { **ε** } $\cup$ 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 ε	FIRST(BC) = FIRST(B)
B is a non-terminal and B derives to ε	FIRST(BC) = FIRST(B) – { **ε** } $\cup$ 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:

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

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 those that produce the empty (ε) string.

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

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

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

$X \rightarrow Y \rightarrow \varepsilon$

$Y \rightarrow \varepsilon$

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

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

Follow set

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

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

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

Predictive parsing table

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

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(α).

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

Example: Given the grammar:

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

m[Y , d]

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

Table m

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() {
  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 → **if** E **then** S
S → **if** E **then** S **else** S

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

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 -> A a | b
```

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

```
A  -> b A'
A' -> a A' | ε
```

A Non-LL(1) Grammar

For instance, a grammar having a production such as

```
A -> a b₁ | a b₂
```

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

$$A \ \text{->}\ a\ b_1 \mid a\ b_2$$

can be left-factored to the following two productions:

$$A \ \text{->}\ a\ A'$$
$$A' \ \text{->}\ b_1 \mid b_2$$

Nonrecursive predictive parsing

Example: Given the grammar:

$S \rightarrow E\$$

$E \rightarrow T\ E'$	$T \rightarrow F\ T'$	$F \rightarrow$ **id**
$E' \rightarrow$ **+** $T\ E'$	$T' \rightarrow$ ***** $F\ T'$	$F \rightarrow$ **(** E **)**
$E' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	

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 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 $.

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 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 == $);
```

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 { /* 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 == $);
```

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε 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