Dr.Y.Bevish Jinila
Department of IT

WORKED OUT EXAMPLES

OPERATOR PRECEDENCE PARSING

1. **Given a grammar,**

E → E + T | T

T → T * F | F

F → (E) | id

Input String : id + id * id

Verify whether the given input string is valid or not using Operator Precedence parsing method.

Soln:

The given grammar is an operator grammar

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

1. COMPUTATION OF LEADING
2. COMPUTATION OF TRAILING
3. CONSTRUCTION OF PRECEDENCE TABLE
4. PARSING THE GIVEN STRING

(1) COMPUTATION OF LEADING (L—R)

Leading (E) = { +, Leading(T)} = { +, *, (, id }
Leading (T) = { *, Leading(F)} = { *, (, id}
Leading (F) = { (, id}

(2) COMPUTATION OF TRAILING (R---L)

Trailing (E) = { +, Trailing (T) } = { +, *,), id }
Trailing (T) = { *, Trailing (F) } = { *,), id }
Trailing (F) = {), id}

(3) COMPUTATION OF PRECEDENCE TABLE

Rules :

1. **$ <. Leading (Starting NT)**
   $ <. Leading (E)
   $ <. {+, *, (, id }
2. **Trailing (Starting NT) .> $**
   Trailing(E) .> $
   {+,*,),id} .> $
3. **On the RHS of the production rule,**
   (a) Terminal NT => Terminal <. Leading (NT)

+ T => + <. Leading(T) => + <. { *, (, id}

*** F => * <. Leading(F) => * <. { (, id}**

(E => (<. Leading(E) = (<.{ +, *, (, id }

(b) NT Terminal => Trailing (NT) .> Terminal

E + => Trailing(E) .> +
=> { +, *,), id } .> +

T * => Trailing(T) .> *

=> { *,), id } .>*

E) => Trailing (E) .>)

=> { +, *,), id } .>)

(c) Ter1 NT Ter2 => Ter1 = Ter2

(=)

	+	*	id	(	)	$
+	.>	<.	<.	<.	.>	.>
*	.>	.>	<.	<.	.>	.>
id	.>	.>	e	e	.>	.>
(	<.	<.	<.	<.	=	e
)	.>	.>	e	e	.>	.>
$	<.	<.	<.	<.	e	Accept

(4) Parsing the string -> id+id*id

Actions : push, pop, accept, error

Rules:

If the tos <. input, push

If the tos .> input, pop -→ continue popping the symbols unitil the tos is related by <. to the recently popped symbol

Stack	Input String	Action
$	id+id*id $	$ <.id [Push]
$id	+id*id $	id .> + [Pop]
$	+id*id $	$ <. + [Push]
$+	id*id $	+ <. id [Push]
$+id	*id $	id .>* [Pop]
$+	*id $	+ <. * [Push]
$+*	id $	*<. id [Push]
$+*id	$	id .> $ [Pop]
$+*	$	*.>$ [Pop]
$+	$	+ .> $ [Pop]
$	$	Accept

String : id*+id

Stack	Input String	Action
$	id*+id$	$ <.id [Push]
$id	*+id$	id .> * [Pop]
$	*+id$	$ <. * [Push]
$*	+id$	*.>+ [Pop]
$	+id$	$ <.+ [Push]
$+	id$	+ <.id [Push]
$+id	$	id .>$ [Pop]
$+	$	+.>$ [Pop]
$	$	Accept

String : id id + id

Stack	Input String	Action
$	id id + id	$ <.id [Push]
$id	id + id	Error

Operator Precedence Graph

Precedence Table

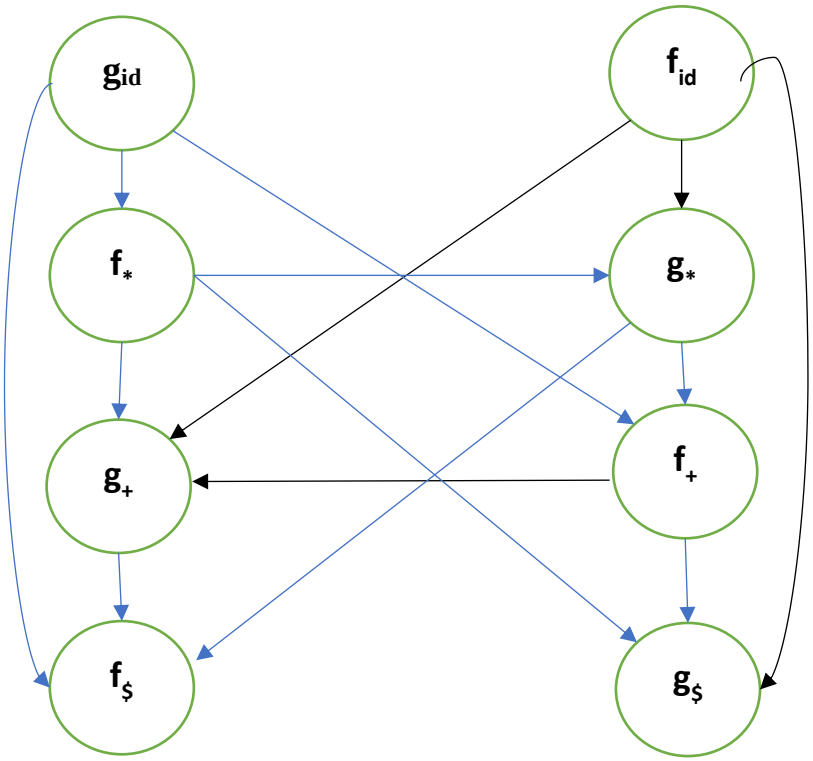
g →

f ↓

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

fid, f+, f*, f$, gid, g+, g*,g$

Operator Precedence Graph



f_{id} -> $g_{\$}$ (1)

f_{id} -> g_{*} ->f_{+} ->$g_{\$}$ (3)

f_{id} -> g_{+} -> $f_{\$}$ (2)

f_{id} -> g_{*} -> $f_{\$}$ (2)

f_{id} -> g_{*} -> f_{+} ->g_{+} -> $f_{\$}$ (4)

g_{id} -> $f_{\$}$ (1)

g_{id} -> f_{*} ->g_{+} ->$f_{\$}$ (3)

g_{id} -> f_{+} -> $g_{\$}$ (2)

g_{id} -> f+ -> g+ -> $f_{\$}$ (3)

g_{id} -> f_{*} -> g* -> $f_{\$}$ (3)

g_{id} -> f_{*} -> g* -> f+ -> g+ ->$f_{\$}$ (5)

Function Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Precedence between 2 operators

**+ *

f+ g*

2 < 3

+ <. *

***** **id**

f* gid

4 < 5

***** **<. id**

Predictive Parsing

1.Given a grammar,

E → E + T | T

T → T * F | F

F → (E) | id

Input String: id + id * id

Verify whether the given input string is valid or not using Predictive parsing method.

Soln:

Since there is left recursion in the grammar, eliminate the left recursion

E -> TE'

E' -> +TE'

E' -> €

T -> FT'

T' -> *FT'

T' -> €

F -> (E)

F -> id

Steps :

1. Computation of FIRST
2. Computation of FOLLOW
3. Construction of Predictive Parsing Table
4. Parsing the String

FIRST

FIRST (E) = {FIRST(T)} = {FIRST(F)} ={(,id}

FIRST (E') = {+, **€ }**

FIRST (T) = {FIRST(F)} ={(,id}

FIRST (T') = { *, **€}**

FIRST (F) = {(,id}

FOLLOW

Follow has to be computed for all the non-terminals in the grammar.

- **While finding the FOLLOW (Starting NT), include $ in the set.**
- **Then you have to inspect the RHS of the production rules..**
  - **If followed by the NT, if there is a terminal, include it in the set**
  - **If followed by the NT, if there is a NT, then find the FIRST(NT) and include it in the set.**
  - **If followed by the NT, if there is nothing (€), find the FOLLOW(LHS NT) and include it in the set.**

FOLLOW(E) = {$,)}

FOLLOW (E') = {FOLLOW(E)} ={$,)}

FOLLOW (T) = {FIRST(E')} = {+, **€ }= {+, FOLLOW(E')} ={+,**$,)}

FOLLOW (T') = {FOLLOW(T)} = **{+,$,)}**

FOLLOW (F) = {FIRST(T')} = { *, **€} ={ *, FOLLOW(T')} = {*, +,$,)}**

Construction of Predictive Parsing Table (Based on the computation of FIRST)

	id	+	*	(	)	$
E	**E -> TE'**			**E -> TE'**		
E'		**E'-> +TE'**			**E' -> €**	**E' -> €**
T	**T -> FT'**			**T -> FT'**		
T'		**T' -> €**	**T'-> *FT'**		**T' -> €**	**T' -> €**
F	**F -> id**			**F -> (E)**		

Parsing the given input String

Stack	Input String	Action
$E	**id+id*id$**	**E -> TE' [Push]**
$E'T	**id+id*id$**	**T -> FT' [Push]**
$E'T'F	**id+id*id$**	**F -> id [Push]**
$E'T'~~id~~	**~~id~~+id*id$**	**[Pop]**
$E'T'	**+id*id$**	**T' -> € [Push]**
$E'	**+id*id$**	**E'-> +TE' [Push]**
$E'T+	**+id*id$**	**[Pop]**
$E'T	**id*id$**	**T -> FT' [Push]**
$E'T'F	**id*id$**	**F -> id [Push]**
$E'T'id	**id*id$**	**[Pop]**
$E'T'	***id$**	**T'-> *FT' [Push]**

$E'T'F*	***id$**	**[Pop]**
$E'T'F	**id$**	**F -> id [Push]**
$E'T'id	**id$**	**[Pop]**
$E'T'	**$**	**T' -> € [Push]**
$E'	**$**	**E' -> € [Push]**
$	**$**	**ACCEPT**

2. Given a grammar,

S → iEtS | iEtSeS | a

E → b

Input String: ibta

Verify whether the given input string is valid or not using Predictive parsing method.

Soln:

- **There is no left recursion**
- **There is left factoring. So, eliminate it.**

S → iEtSS'

S → a

S' → eS

S' → €

E → b

Computation of FIRST

FIRST(S) = {i,a}

FIRST(S') = {e, €}

FIRST(E) = {b}

Computation of FOLLOW

FOLLOW(S) = {$,FIRST(S'),FOLLOW(S')} ={$, e, €}

FOLLOW(S') = {FOLLOW(S)} ={$, e, €}

FOLLOW(E) = {t}

Construction of Predictive Parsing Table (According to FIRST)

	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS			S' → €
E		E → b				

Parsing the given input string (ibta)

Stack	Input String	Action
$S	ibta$	S → iEtSS' [Push]
$S'StEi	ibta$	[Pop]
$S'StE	bta$	E → b [Push]
$S'Stb	bta$	[Pop]
$S'St	ta$	[Pop]
$S'S	a$	S → a [Push]
$S'a	a$	[Pop]
$S'	$	S' → € [Push]
$	$	ACCEPT

SLR PARSING

Problem

Construct the SLR parsing table and parse the string abab for the following grammar

S -> AA

A -> aA | b

Soln:

1. Augmented Grammar

S'-> S
S -> AA
A -> aA | b

Augmented Grammar

S' -> S
S -> AA
A -> aA
A -> b

LR (0) Items

I_0

S' -> .S
S -> .AA
A -> .aA
A -> .b

GOTO (I_0, S) =

I_1

S' -> S.

GOTO (I_0, A) =

I_2

S -> A.A
A -> .aA
A -> .b

GOTO (I_0, a) =

I_3

A -> a.A
A -> .aA
A -> .b

GOTO (I_0, b) =

I_4

A -> b.

GOTO (I_2, A) =

I_5

S -> AA.

GOTO (I_2, a) =

I_3

A -> a.A
A -> .aA
A -> .b

GOTO (I_2, b) =

I_4

A -> b.

GOTO (I_3, A) =

I_6

A -> aA.

GOTO (I_3, a) =

I_3

A -> a.A
A -> .aA
A -> .b

GOTO (I_3, b) =

I_4

A -> b.

LR (0) Items

I_0
S'-> . S
S -> . AA
A -> . aA
A -> . b

Goto (I_0, S)
I_1
S'-> S .

Goto (I_0, A)
I_2
S -> A.A
A -> . aA
A -> . b

Goto (I_0, a)
I_3
A -> a.A
A -> . aA
A -> . b

Goto (I_0, b)
I_4
A -> b.

Goto (I_2, A)
I_5
S -> AA.

Goto (I_2, a)
I_3
A -> a . A
A -> . aA
A -> . b

Goto (I_2, b)
I_4
A -> b.

Goto (I_3, A)
I_6
A -> aA .

Goto (I_3, a)
I_3
A -> a.A
A -> . aA
A -> . b

Goto (I_3, b)
I_4
A -> b.

SLR PARSING TABLE

State	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			ACCEPT		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Reduce Action

I4 :A -> b.

Follow(A)=
FIRST(A)= {a,b,$}

I5: S->AA.

Follow(S)= {$}

I6: A -> aA .

Follow(A)=
FIRST(A)= {a,b,$}

1.S -> AA
2.A -> aA
3.A -> b

Parsing the given input string

Stack	Input String	Action
0	abab$	Shift(S3)
0a3	bab$	Shift(S4)
0a3b4	ab$	Reduce (R3) A -> b
0a3A6	ab$	Reduce (R2) A -> aA
0A2	ab$	Shift (S3)
0A2a3	b$	Shift (S4)
0A2a3b4	$	Reduce (R3) A -> b
0A2a3A6	$	Reduce (R2) A -> aA
0A2A5	$	Reduce (R1) S -> AA
0S1	$	Accept