SYNTAX ANALYSIS

BOTTOM-UP PARSING

Operator Precedence Parser

- Operator precedance parser can be constructed from Operator Grammar
- Operator Grammar: The grammar which has property that no production on right side is **ε or has two adjacent non-terminal**
- Example:

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Operator Precedance Parser

- There are three disjoint precedance relations
  - < less than
  - = Equal to
  - > Greater than
- Suppose there are two operators a and b then relations give following meaning:
  - a < b – a gives precedance to b
  - a = b – a has same precedance as of b
  - a > b – a takes precedance over b

Operator Precedance Parser

Rules for finding operator precedance relations if a1 & a2 are operators

1. If a1 has higher precedance than a2 then make a1 > a2 and a2 < a1
2. If a1 has equal precedance with a2 and if operators are left associative then make a1 > a2 and a2 > a1
3. If a1 has equal precedance with a2 and if operators are right associative then make a1 < a2 and a2 < a1

Operator Precedance Parser

Rules for finding operator precedance relations if a1 & a2 are operators

4. For all operators a,

   a < id and id > a

   a < (and (< a

   a >) and) > a

   a > $ and $ < a

5. Operator ↑ **has highest precedance and right associativity**

6. Operator * and / has next higher precedance and left associativity

Operator Precedance Parser

Rules for finding operator precedance relations if a1 & a2 are operators

7. **Operator + and - has lowest precedance and left associativity**

8. The blank entries in operator precedance relation table indicates an error

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+									
-									
*									
/									
↑									
id									
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-									
*									
/									
↑									
id									
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*									
/									
↑									
id									
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/									
↑									
id									
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑									
id									
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(									
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)									
$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$									

Operator Precedence Relation Table

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		Accept

Operator Precedence Algorithm

Input: A string w and table of precedence relations

Output: If w is well formed, a Skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes otherwise an error indication

Method: Initially the stack contains $ and the input buffer the string w $

Operator Precedence Algorithm

```
Set ip to point to the first symbol of w$
Repeat forever
   If $ is on top of stack and ip points to $ then
         Return
   else begin
        Let a be the topmost terminal symbol on top of stack
        And let b be the symbol pointed to by ip;
        If a < b or a = b then begin
             Push b on the top of the stack
             Advance ip to the next symbol
        End
        Else if a > b then
           Repeat
                  Pop of the stack
           Until the top stack terminal is related by <
           to the terminal most recently popped
        Else
           Error( )
End
```

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *
$ + *	<	id $	Push id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *
$ + *	<	id $	Push id
$ + * id	>	$	Pop id

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *
$ + *	<	id $	Push id
$ + * id	>	$	Pop id
$ + *	>	$	Pop *

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *
$ + *	<	id $	Push id
$ + * id	>	$	Pop id
$ + *	>	$	Pop *
$ +	>	$	Pop +

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

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

Stack	Relation	Input	Comment
$	<	id + id * id $	Push id
$ id	>	+ id * id $	Pop id
$	<	+ id * id $	Push +
$ +	<	id * id $	Push id
$ + id	>	* id $	Pop id
$ +	<	* id $	Push *
$ + *	<	id $	Push id
$ + * id	>	$	Pop id
$ + *	>	$	Pop *
$ +	>	$	Pop +
$	Accept	$	Accept

Operator Precedence Parser

Steps to construct syntax tree (Expression Tree):

1. **Keep track of elements that are popped in the same order. Consider that sequence as input string for processing using stack.**
2. **Read the processing sequence from left to right**
3. **If operand (identifier) is found, then push that onto stack**
4. **If an operator is found, then pop out top 2 elements and construct subtree with operator as root node.**
5. **Push this newly constructed subtree on top of stack**
6. **Repeat step 3 to 5 until all the symbols in input string are processed.**
7. **When input string is completed then pop out topmost element of stack.**
8. **If stack is not empty after pop then declare ERROR otherwise POPPED ELEMENT is final SYNTAX TREE**

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Push id
Pop id
Push +
Push id
Pop id
Push *
Push id
Pop id
Pop *
Pop +
Accept

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (1)
Pop id (2)
Pop id (3)
Pop * (4)
Pop + (5)
Accept

Sequence to be processed:

id , id , id, * , +

Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
id
id
id
$

Sequence to be processed:

id , id , id, * , +

Operator Precedence Parser

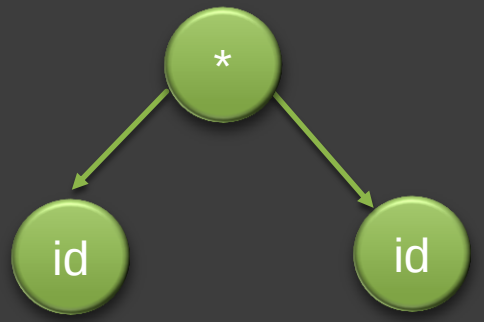
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
id
$

Sequence to be processed:

id , id , id, * , +



Sub Tree 1

Operator Precedence Parser

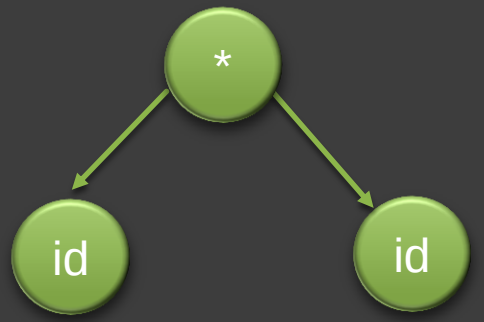
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
Sub Tree 1
id
$

Sequence to be processed:

id , id , id, * , +



Sub Tree 1

Operator Precedence Parser

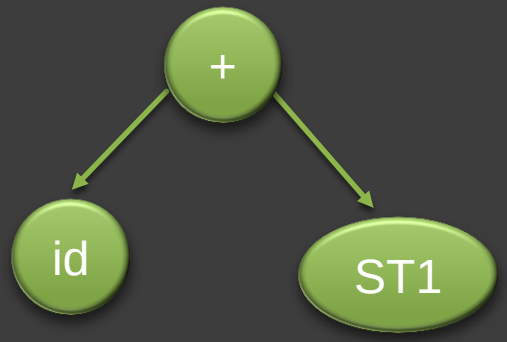
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
$

Sequence to be processed:

id , id , id, * , +



Sub Tree 1

Operator Precedence Parser

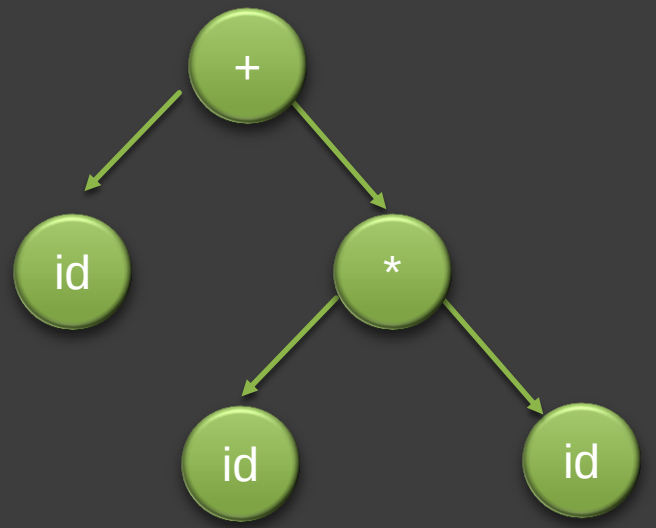
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
$

Sequence to be processed:

id , id , id, * , +



Sub Tree 2

Operator Precedence Parser

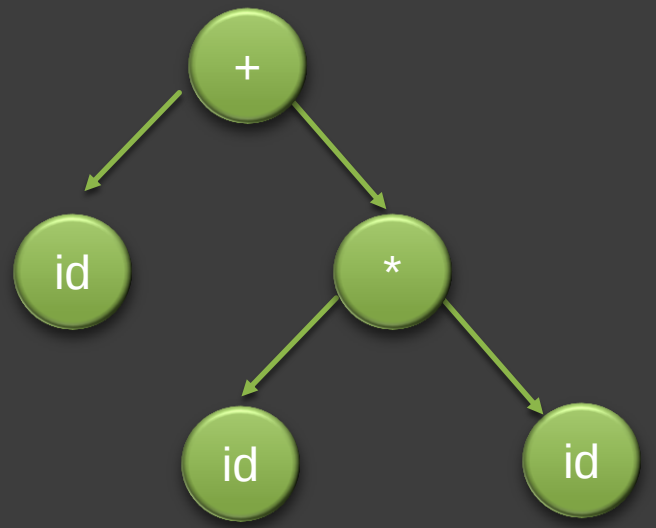
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
Sub Tree 2
$

Sequence to be processed:

id , id , id, * , +



Sub Tree 2

Operator Precedence Parser

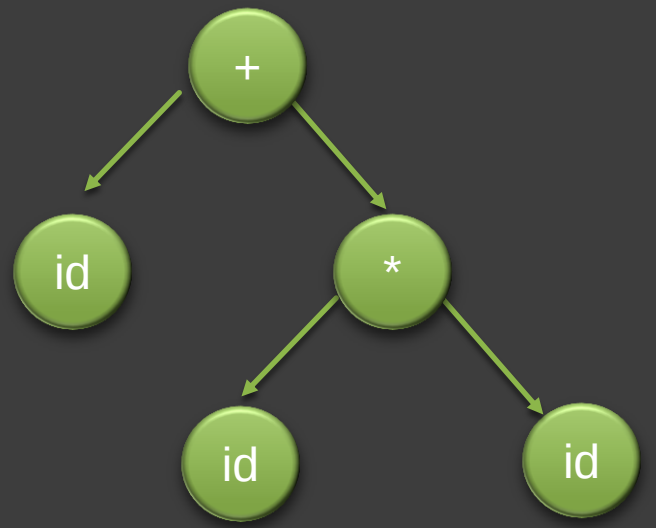
E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Stack
$

Sequence to be processed:

id , id , id, * , +



Final Tree

Operator Precedence Parser

Steps to construct Entire Derivation Tree:

1. **Keep track of elements that are popped in the reverse order. (Start from Bottom and move in Top Direction)**
2. **Select start symbol as a root node.**
3. **Select the Next unprocessed symbol from list of popped elements.**
4. **Generate sub tree in which this symbol act as a leaf node by choosing one (or set) of the suitable production**
5. **Select rightmost node of this newly generated subtree such that it is non terminal. This node will act as a root node.**
6. **Repeat step 3 to 5 until all the symbols in popped sequence are processed.**
7. **The final tree is Required Derivation Tree**

Operator Precedence Parser

Construction of Entire Tree:

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1)
Accept

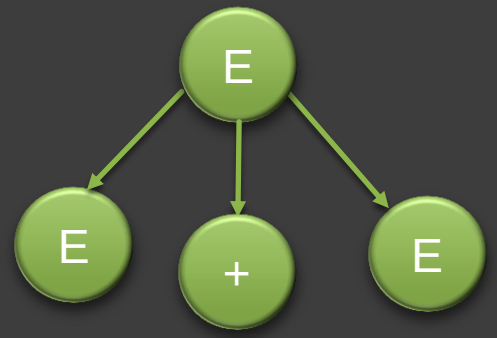
Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1): E → E + E
Accept

Construction of Entire Tree:



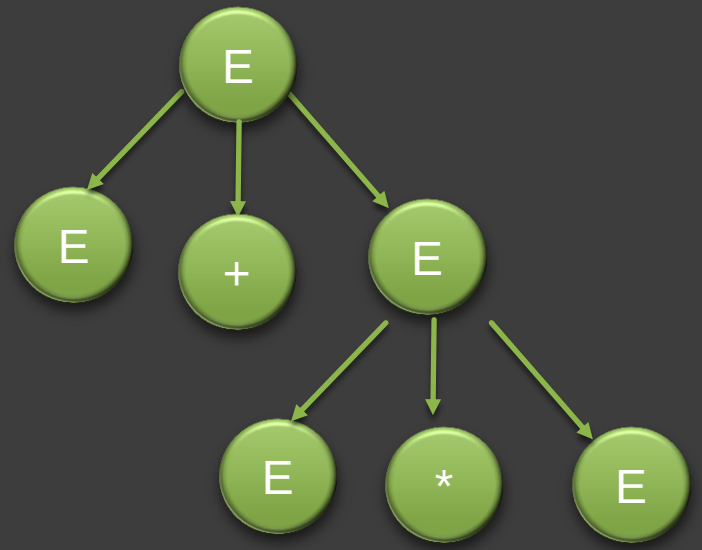
Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2) : E → E * E
Pop + (1): E → E + E
Accept

Construction of Entire Tree:



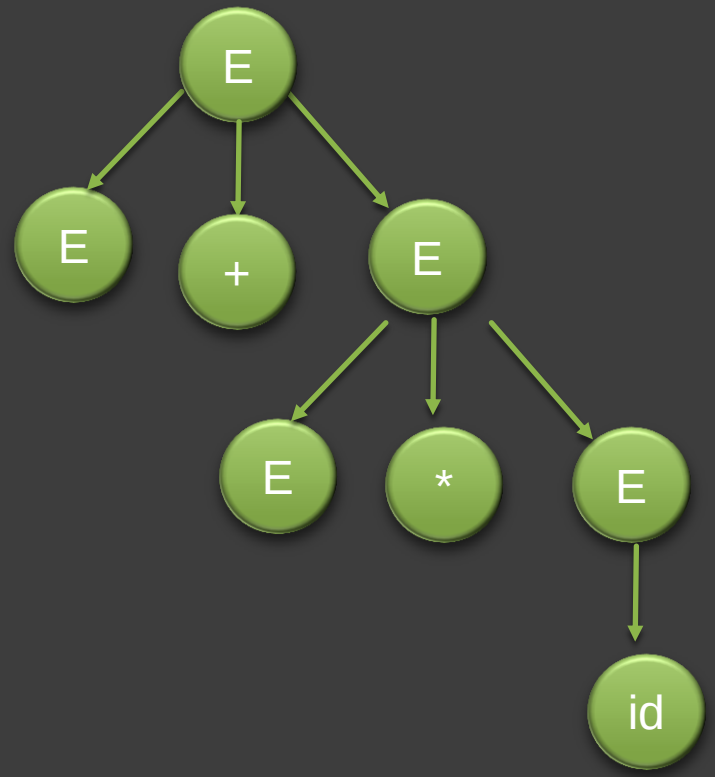
Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3) : E → id
Pop * (2) : E → E * E
Pop + (1): E → E + E
Accept

Construction of Entire Tree:



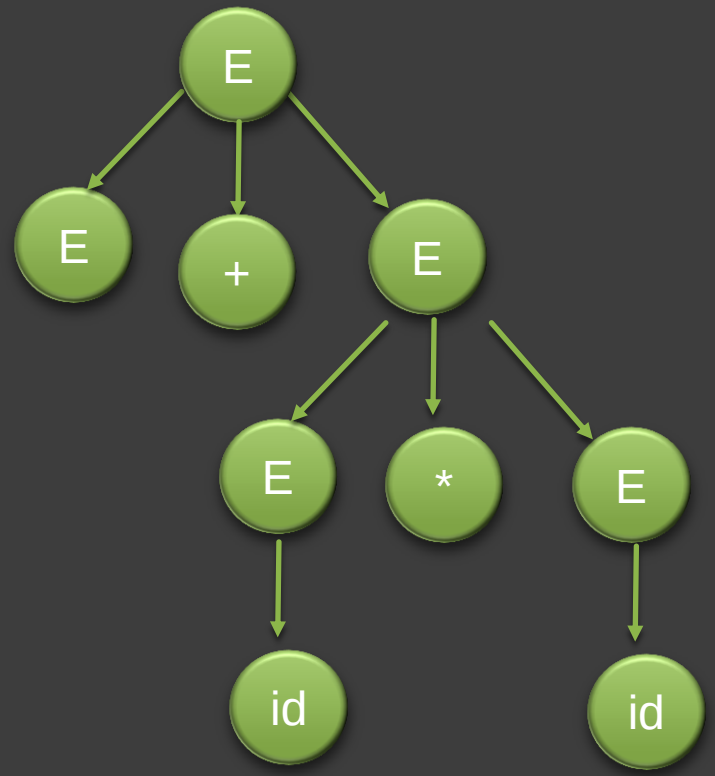
Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5)
Pop id (4): E → id
Pop id (3) : E → id
Pop * (2) : E → E * E
Pop + (1): E → E + E
Accept

Construction of Entire Tree:



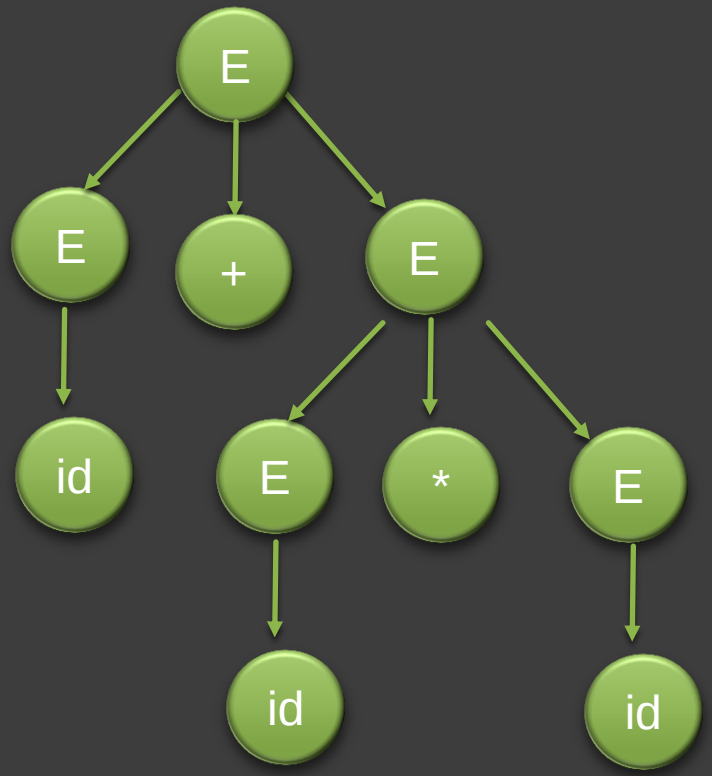
Operator Precedence Parser

E → E + E | E – E | E * E | E / E | E ↑ **E | (E) | id**

Input: id + id * id

Comment
Pop id (5): E → id
Pop id (4): E → id
Pop id (3) : E → id
Pop * (2) : E → E * E
Pop + (1): E → E + E
Accept

Construction of Entire Tree:



Operator Precedence Parser

E → E + T | T

T → T * V | V

V → a | b | c | d

Input: a + b * c * d

	a	b	c	d	+	*	$
a					>	>	>
b					>	>	>
c					>	>	>
d					>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
$	<	<	<	<	<	<	A

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *
$ + *	<	d $	Push d

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *
$ + *	<	d $	Push d
$ + * d	>	$	Pop d

Operator Precedence Parser

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *
$ + *	<	d $	Push d
$ + * d	>	$	Pop d
$ + *	>	$	Pop *

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *
$ + *	<	d $	Push d
$ + * d	>	$	Pop d
$ + *	>	$	Pop *
$ +	>	$	Pop +

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Stack	Relation	Input	Comment
$	<	a + b * c * d $	Push a
$ a	>	+ b * c * d $	Pop a
$	<	+ b * c * d $	Push +
$ +	<	b * c * d $	Push b
$ + b	>	* c * d $	Pop b
$ +	<	* c * d $	Push *
$ + *	<	c * d $	Push c
$ + * c	>	* d $	Pop c
$ + *	>	* d $	Pop *
$ +	<	* d $	Push *
$ + *	<	d $	Push d
$ + * d	>	$	Pop d
$ + *	>	$	Pop *
$ +	>	$	Pop +
$	Accept	$	Accept

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Push a
Pop a
Push +
Push b
Pop b
Push *
Push c
Pop c
Pop *
Push *
Push d
Pop d
Pop *
Pop +
Accept

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (1)
Pop b (2)
Pop c (3)
Pop * (4)
Pop d (5)
Pop * (6)
Pop + (7)
Accept

Sequence to be processed:

a , b , c , * , d , * , +

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

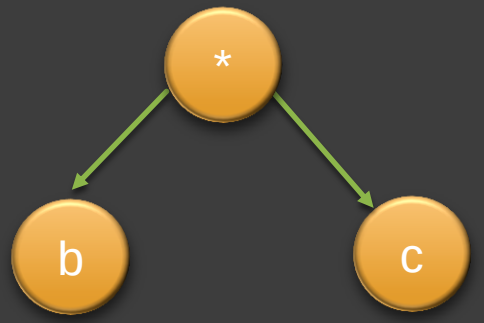
Stack
c
b
a
$

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
a
$



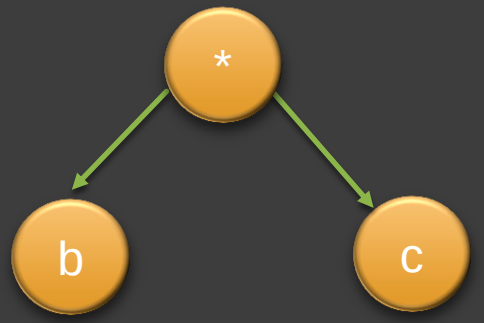
Sub Tree 1

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 1
a
$



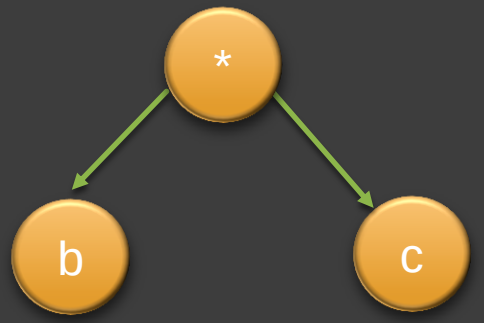
Sub Tree 1

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
d
Sub Tree 1
a
$



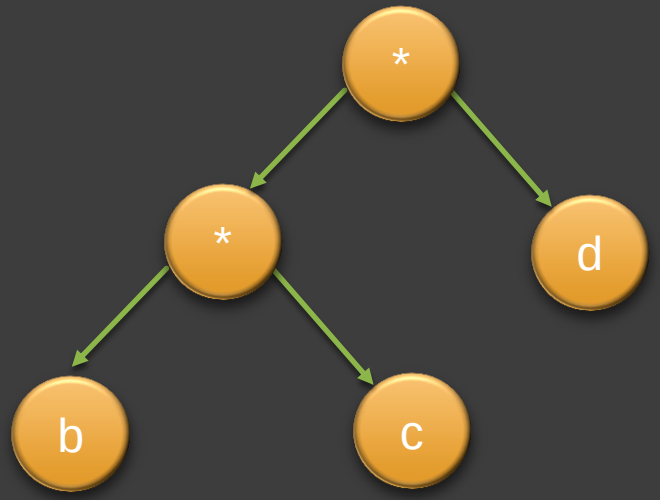
Sub Tree 1

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
a
$



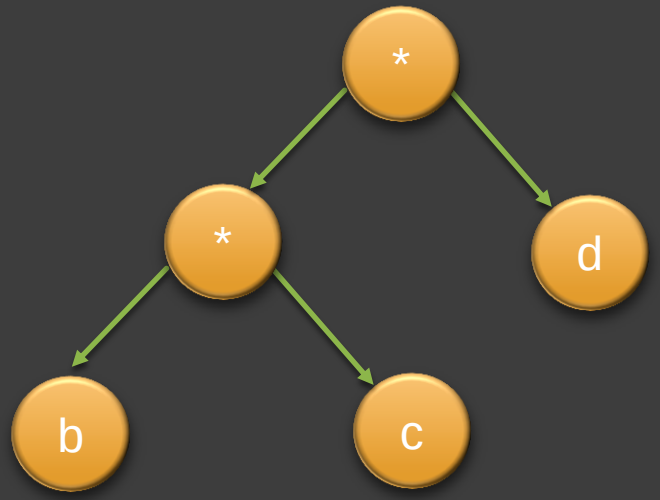
Sub Tree 2

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 2
a
$



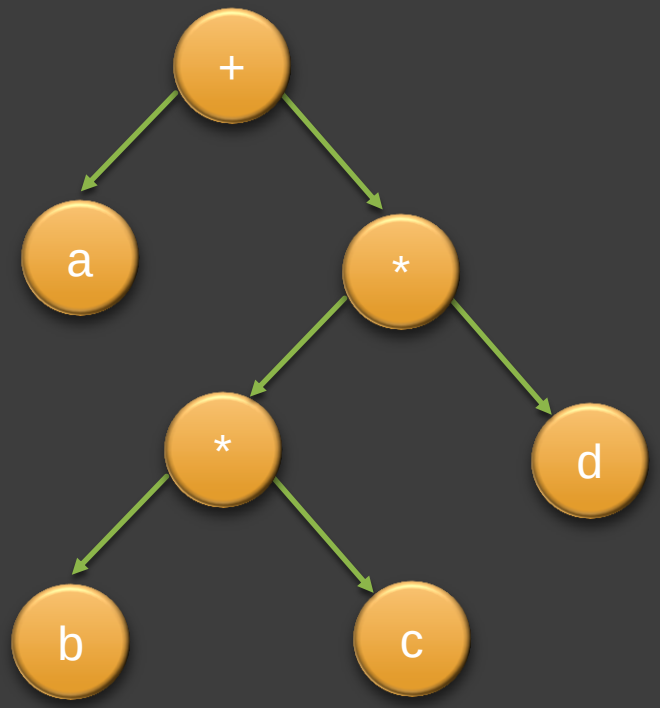
Sub Tree 2

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
$



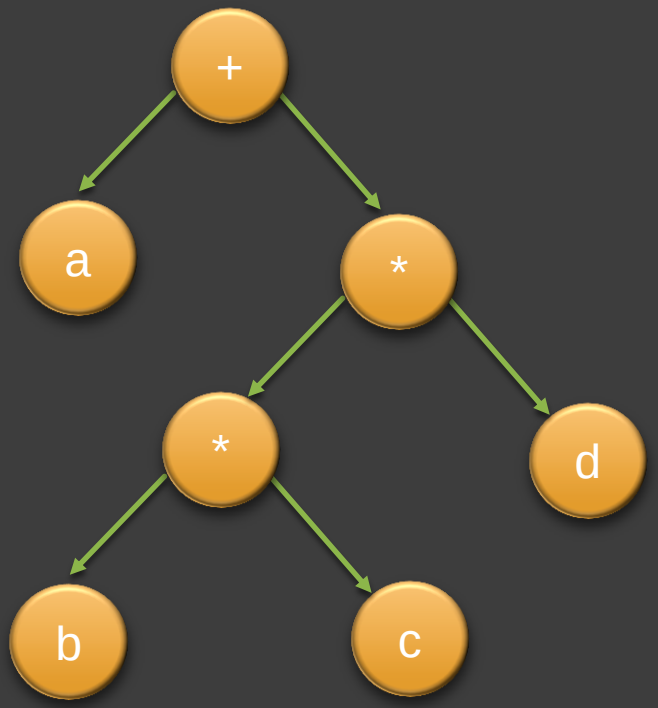
Sub Tree 3

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 3
$



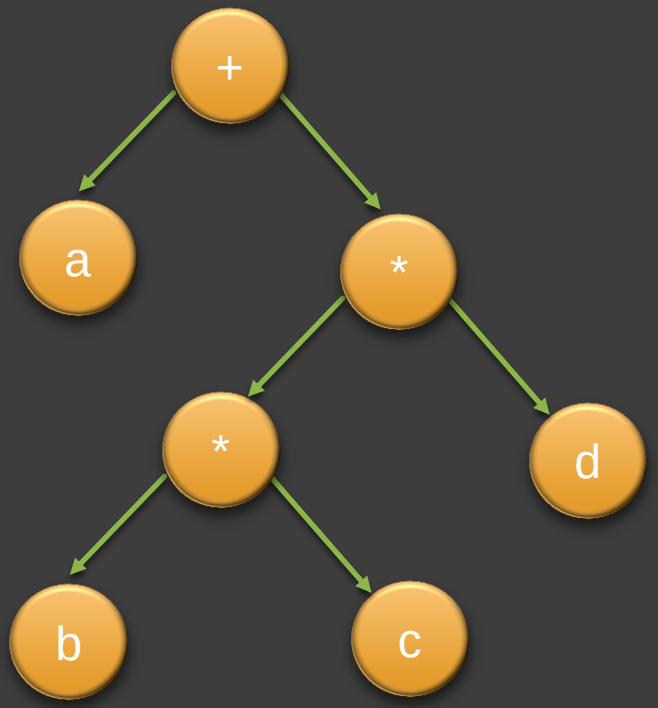
Sub Tree 3

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
$

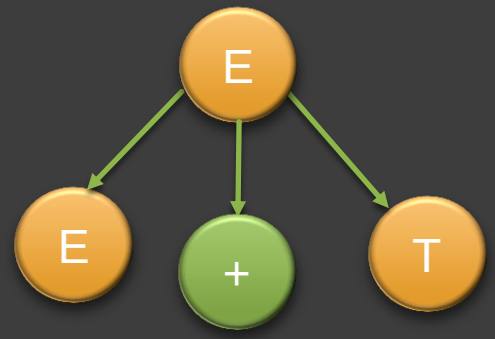


Final Tree

E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2)
Pop + (1) : E → E + T
Accept

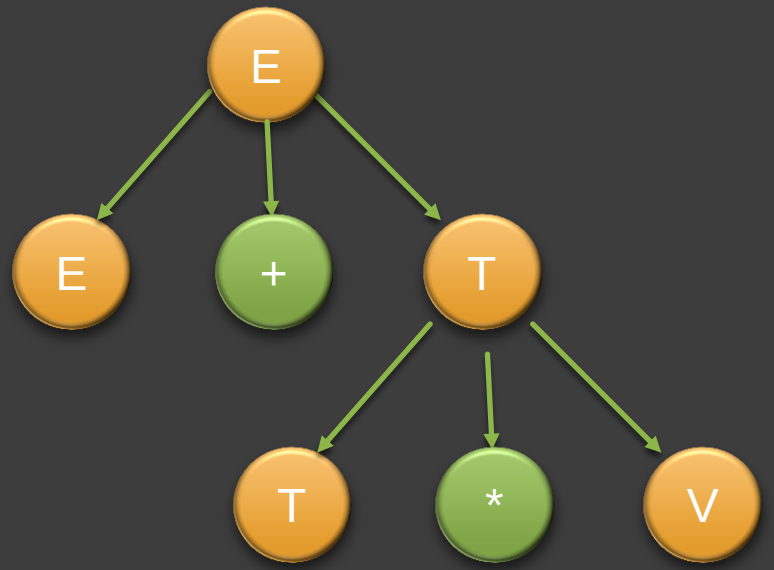
Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2) : T → T * V
Pop + (1) : E → E + T
Accept

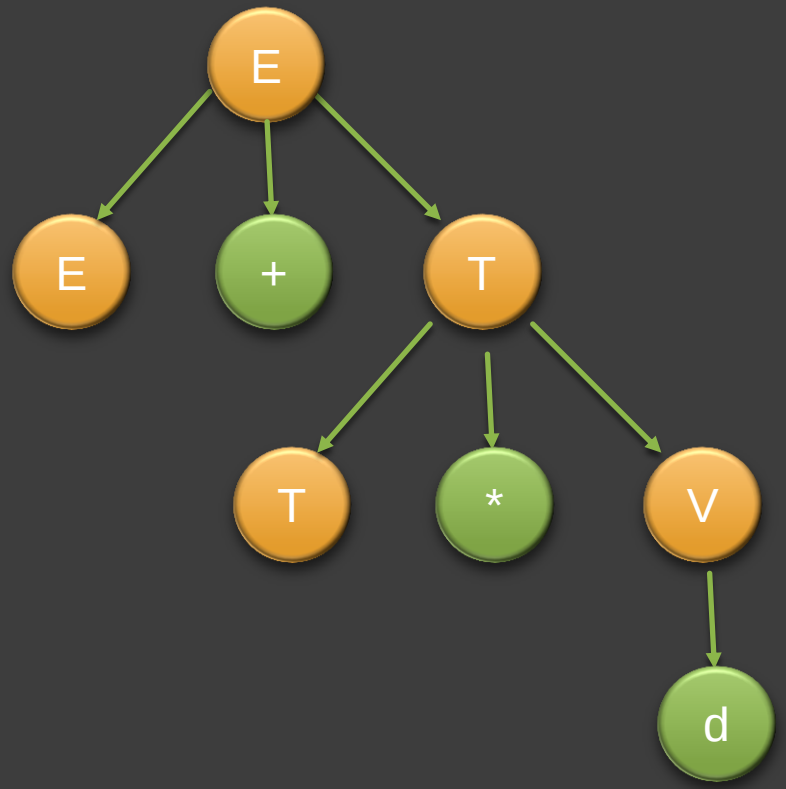
Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3) : V → d
Pop * (2) : T → T * V
Pop + (1) : E → E + T
Accept

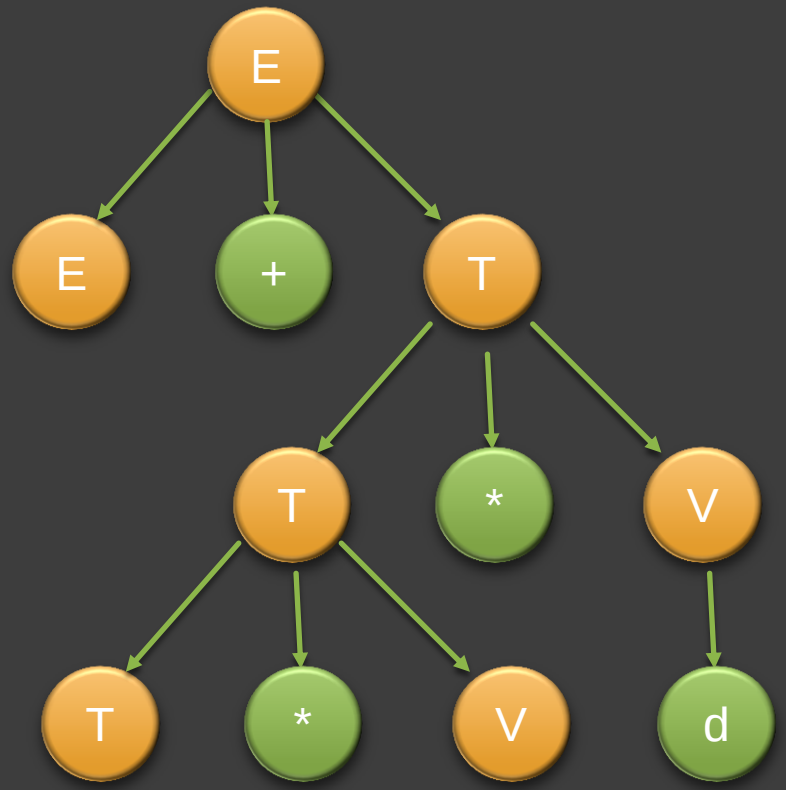
Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4) : T → T * V
Pop d (3) : V → d
Pop * (2) : T → T * V
Pop + (1) : E → E + T
Accept

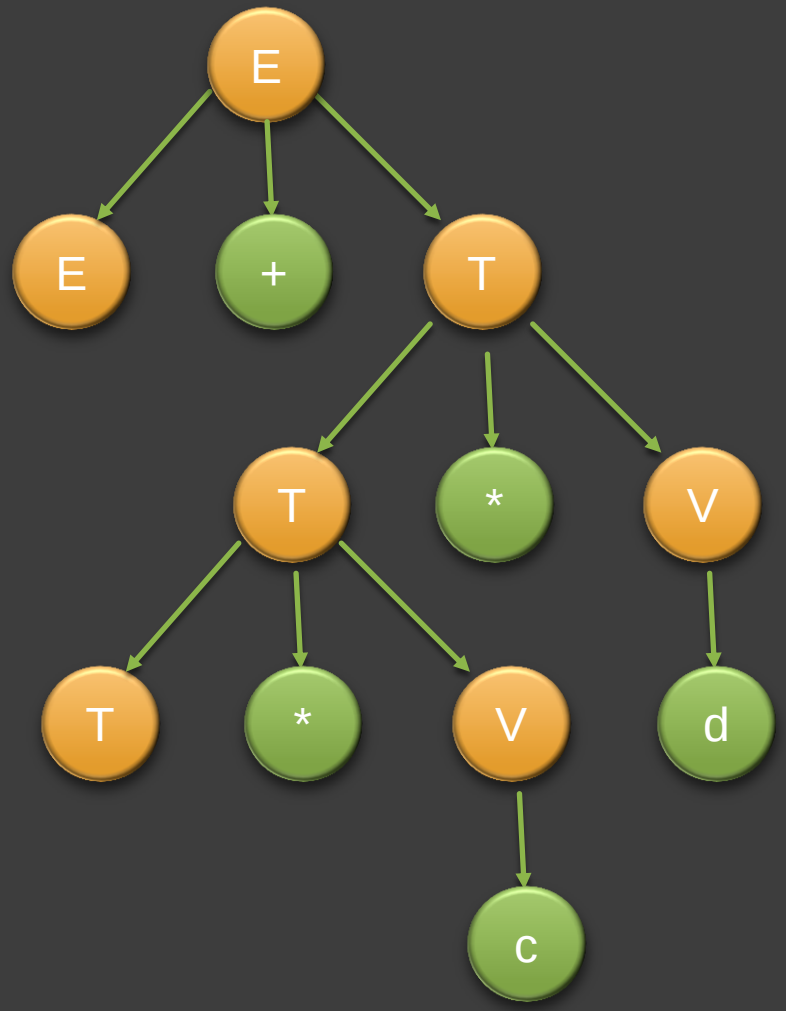
Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6)
Pop c (5) : V → c
Pop * (4) : T → T * V
Pop d (3) : V → d
Pop * (2) : T → T * V
Pop + (1) : E → E + T
Accept

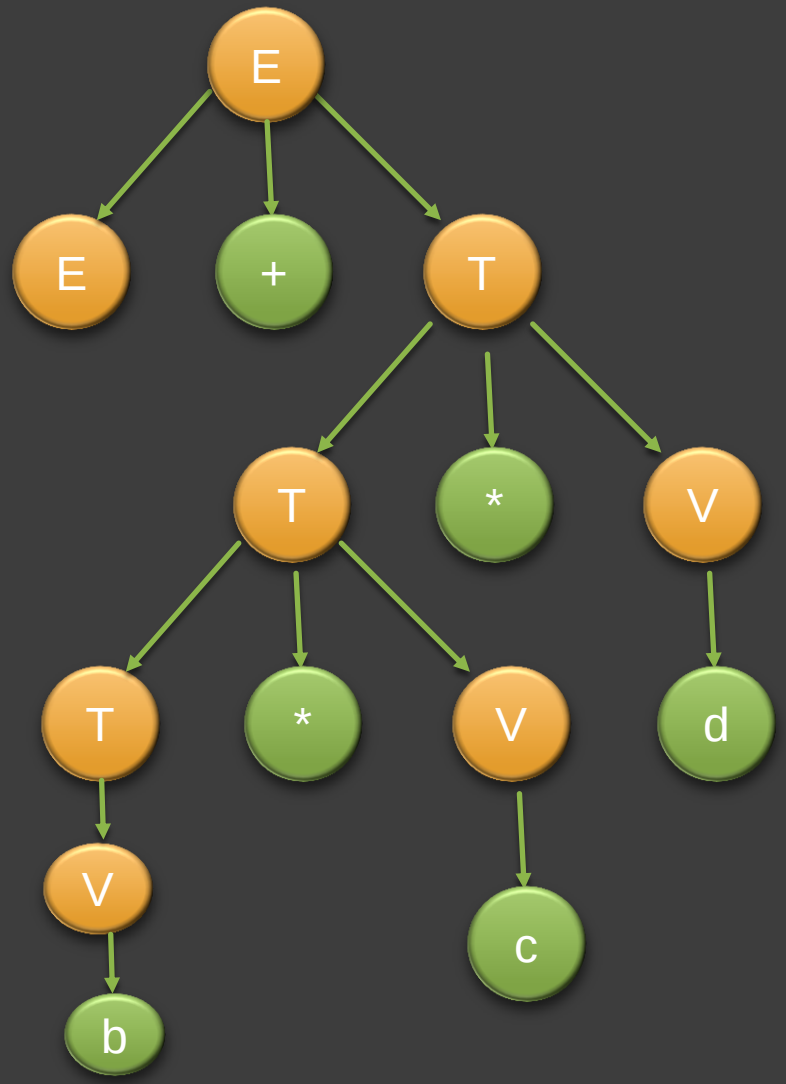
Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7)
Pop b (6) : T → V, V → b
Pop c (5) : V → c
Pop * (4) : T → T * V
Pop d (3) : V → d
Pop * (2) : T→ T * V
Pop + (1) : E → E + T
Accept

Construction of Entire Tree:



E → E + T | T T → T * V | V V → a | b | c | d Input: a + b * c * d

Comment
Pop a (7) : E → T, T→ V, V → a
Pop b (6) : T → V, V → b
Pop c (5) : V → c
Pop * (4) : T → T * V
Pop d (3) : V → d
Pop * (2) : T → T * V
Pop + (1) : E → E + T
Accept

Construction of Entire Tree: