Another example

S -> iEtSS' | a
S' -> eS | ɛ
E -> b

First(S) = {i,a}
First(S') = {e, ɛ}
First(E) = {b}

Follow(S) = {$,e}
Follow(S') = {$,e}
Follow(E) = {t}

Non - terminal	Input Symbol					
	a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> ɛ S' -> eS			S' -> ɛ
E		E -> b				

Non-recursive predicting parsing
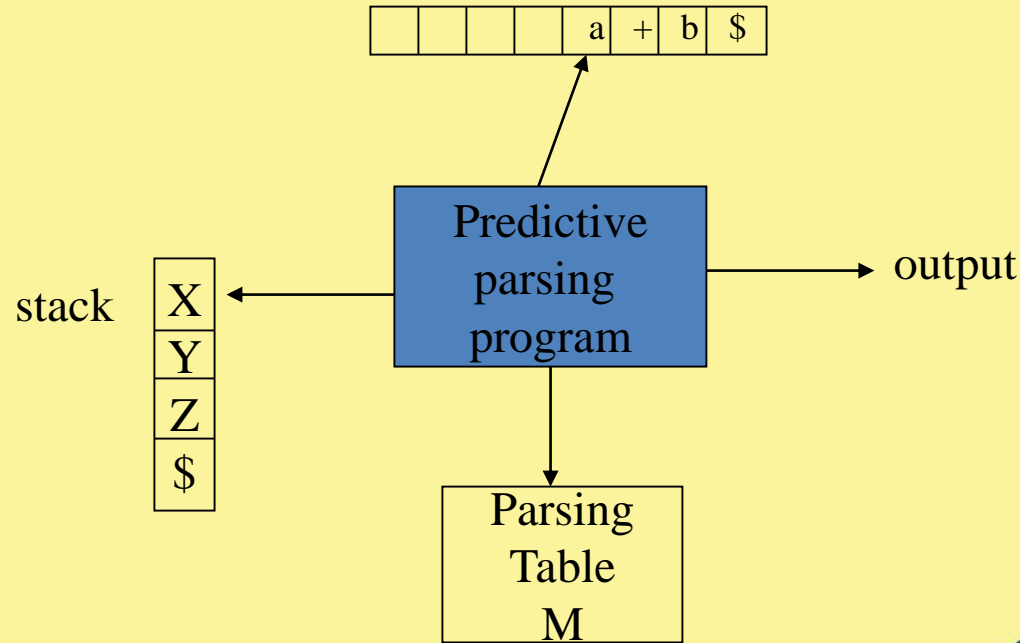
Predictive parsing algorithm

```
While (stack is not empty) do
     Let X be the top symbol in the stack;
     Let a be the next input symbol;
     if (X is a) pop the stack and advance input pointer;
     else if (X is a terminal) error();
     else if (M[X,a] is an error entry) error();
     else if (M[X,a] = X->Y1Y2..Yk) {
             output the production X->Y1Y2..Yk;
             pop the stack;
             push Yk,...,Y2,Y1 on to the stack with Y1 on top;
     }
```

Example

id+id*id$

Stack	Input	Action
E	id+id*id$	Parse E -> TE'
E'T	id+id*id$	Parse E' ->FT'
E'T'F	id+id*id$	Parse F -> id
E'T'id	id+id*id$	Advance input
E'T'	+id*id$	Parse T' -> Ɛ
E'	+id*id$	Parse E' -> +TE'
E'T+	+id*id$	Advance input
E'T	id*id$	Parse T ->FT'

Stack	Input	Action
E'T'F	id*id$	Parse F -> id
E'T'id	id*id$	Advance input
E'T'	*id$	Parse T'->*FT'
E'T'F*	*id$	Advance input
E'T'F	id$	Parse F -> id
E'T'id	id$	Advance input
E'T'	$	Parse T' -> Ɛ
E'	$	Parse E' -> Ɛ
	$	Done

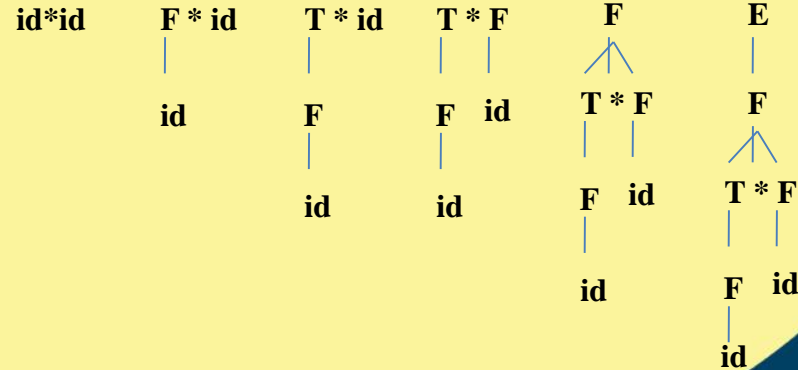| Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E'-> ε	E'-> ε
T	T -> FT'			T -> FT'		
T'		T'-> ε	T'-> *FT'		T'-> ε	T'-> ε
F	F -> **id**			F -> (E)		

4

Bottom-up Parsing

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

```
id*id    F * id    T * id    T * F      F           E
          |         |         |  |      /\          |
id        F         F   id    T * F     T * F        F
          |         |         |  |      /\
          id        id        F   id    T * F
                              |         |  |
                              id        F   id
                                        |
                                        id
```

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

Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied

- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production

- The key decisions during bottom-up parsing are about when to reduce and about what production to apply

- A reduction is a reverse of a step in a derivation

- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

7

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

  Stack   Input
  $ w$

- Acceptance configuration

  Stack   Input
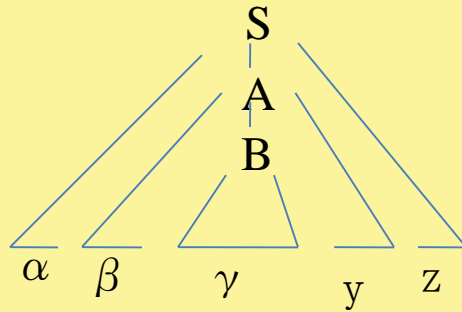 $S $

Shift reduce parsing (cont.)

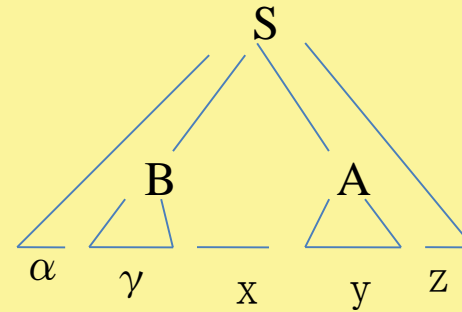- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error

- Example: id*id

Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

Handle will appear on top of the stack

S
A
B
α β γ y z

Stack   Input
$ α β γ   yz$
$ α β B   yz$
$ α β By   z$

S
B   A
α γ x y z

Stack   Input
$ α γ   xyz$
$ α Bxy   z$

Conflicts during shift reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict

- Example:

stmt ⟶ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**

Stack

… if expr then stmt

Input

else …$

Reduce/reduce conflict

stmt -> id(parameter_list)
stmt -> expr:=expr
parameter_list->parameter_list, parameter
parameter_list->parameter
parameter->id
expr->id(expr_list)
expr->id
expr_list->expr_list, expr
expr_list->expr

Stack	Input
… id(id	,id) …$

Bottom-Up Parsing

- Operator Precedence Parsing
- LR Parsing

Operator Precedence Parsing

Operator Grammar

- No ε-transition
- No two adjacent non-terminals

Eg.

  E → E op E | id

  op → + | *

  The above grammar is not an operator grammar but:

  E → E + E | E* E | id

Operator Precedence

- If a has higher precedence over b; $a \cdot > b$
- If a has lower precedence over b; $a < \cdot \, b$
- If a and b have equal precedence; $a \doteq b$

Note:
  - id has higher precedence than any other symbol
  - \$ has lowest precedence.
  - if two operators have equal precedence, then we check the **Associativity** of that particular operator.

Precedence Table

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

Example: w = $id + id * id$

$ <· id ·> + <· id ·> * <· id ·> $

Basic Principle

- Scan input string left to right, try to detect ·> and put a pointer on its location.

- Now scan backwards till reaching <·

- String between <· and ·> is our handle.

- Replace handle by the head of the respective production.

- REPEAT until reaching start symbol.

Algorithm

Initialize stack to $

while true do

    let *U* be the topmost terminal in the stack

    let *V* be the next input symbol

    if *U* = $ and *V* = $ then return

    if *U* <· *V* or *U* ≐ *V* then

        shift V onto stack

        advance input pointer

    else if *U* ·> *V* then

        do

            pop the topmost symbol, call it *V*, from the stack

        until the top of the stack is <· *V*

    else

        error

  end

Example

STACK	INPUT	ACTION/REMARK
$	id + id * id$	$ <· id
$ id	+ id * id$	id ·> +
$	+ id * id$	$ <· +
$ +	id * id$	+ <· id
$ + id	* id$	id ·> *
$ +	* id$	+ <· *
$ + *	id$	* <· id
$ + * id	$	id ·> $
$ + *	$	* ·> $
$ +	$	+ ·> $
$	$	accept

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

Establishing Precedence Relationships

- Construct two lists: Firstop+, Lastop+

- Firstop+: List of all terminals which can appear first on any right hand side of a production

- Lastop+: List of terminals that can appear last on any right hand side of a production

Firstop and Lastop

- For X -> a...|Bc, put a, B, c in Firstop(X)
- For Y -> ...u|...vW, put u, v, W in Lastop(Y)
- Compute Firstop+ and Lastop+ using Closure algorithm
  - Take each nonterminal in turn, in any order and look for it in all the Firstop lists. Add its own first symbol list to any other in which it occurs
  - Similarly process Lastop list
  - Drop all nonterminals from the lists

Constructing Precedence Matrix

- Whenever terminal a immediately precedes nonterminal B in any production, put a <·α where α is any terminal in the Firstop+ list of B

- Whenever terminal b immediately follows nonterminal C in any production, put β ·>b where β is any terminal in the Lastop+ list of C

- Whenever a sequence aBc or ac occurs in any production, put a $\doteq$ c

- Add relations $<· a and a ·> $ for all terminals in the Firstop+ and Lastop+ lists, respectively of S

Example

E -> E + T | T   Firstop(E) = {E, +, T}   Lastop(E) = {+, T}
T -> T * F | F   Firstop(T) = {T, *, F}   Lastop(T) = {*, F}
F -> (E) | id   Firstop(F) = {(, id}   Lastop(F) = {), id}

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

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

Example (Contd.)

E -> E + T | T
T -> T * F | F
F -> (E) | id

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

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

	$	(	)	id	+	*
$		<·		<·	<·	<·
(		<·	≐	<·	<·	<·
)	·>		·>		·>	·>
id	·>		·>		·>	·>
+	·>	<·	·>	<·	·>	<·
*	·>	<·	·>	<·	·>	·>