III
Syntax Analysis
(Top Down and Bottom Up parsing)

Shyamalendu Kandar
Assistant Professor,
Department of Information Technology
IIEST, Shibpur

Parsing

- Parsing is a process of determining if a string of tokens can be generated by a grammar.
- Top down parsing can be viewed as an attempt to find left most derivation of an input.

III.A.
Top-Down Parsing

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - **Recursive-Descent Parsing**
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - **Predictive Parsing**
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

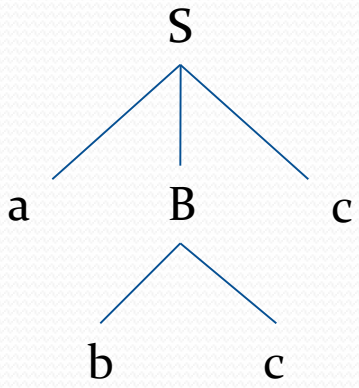
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.
- A left recursive grammar can cause a recursive descent parser.

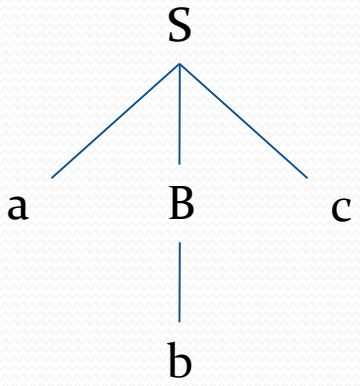
$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



fails, backtrack



Predictive Parser

- The goal of predictive parsing is to construct a top down parser that never backtracks.
- To do so we must transform the grammar in two ways
  - Eliminate Left Recursion
  - Perform Left factoring

  These rules eliminate most common causes of backtracking although they do not guarantee a complete backtrack free parsing.

  At each step, the choice of rule to be expanded is made upon the next terminal symbol.

- Suppose
- **A -> A1 | A2 | ....| An**
- **If the non-terminal to be expanded next is 'A' , then the choice of rule is made on the basis of the current input symbol 'a' only.**

III.A.1
Predictive Parser

Predictive Parser

Procedure for making an Predictive Parser:

- Make a transition diagram (like dfa/nfa) for every rule of the grammar.
- Optimize the dfa by reducing the number of states, yielding the final transition diagram.
- To parse a string, simulate the string on the transition diagram.
- If after consuming the input the transition diagram reaches an accept state, it is parsed.

Predictive Parser

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

Elimination Left Recursion it becomes:

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Predictive Parser

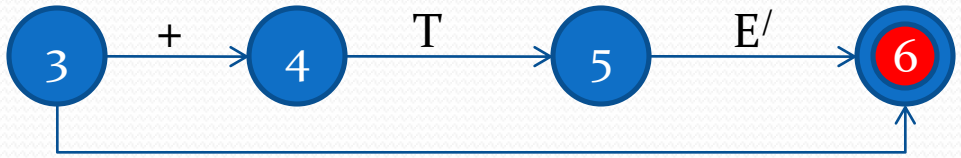
Rule 1: Make a transition diagram (like dfa/nfa) for every rule of the grammar.

$E \rightarrow T\ E'$

E:

$E' \rightarrow +T\ E' \mid \varepsilon$

E/ :

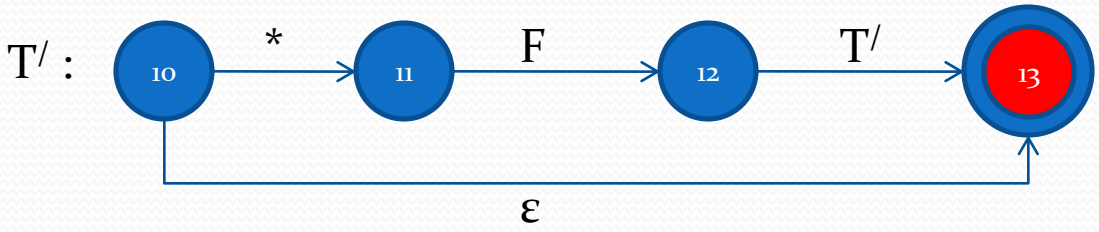


$T \rightarrow F\ T'$

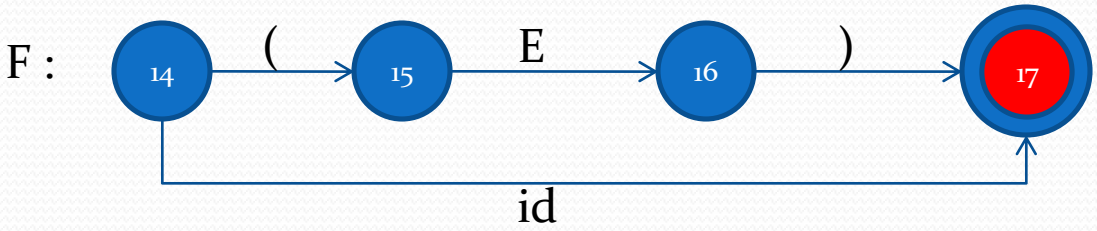
T:

Predictive Parser

$T' \rightarrow *F\ T' \mid \varepsilon$

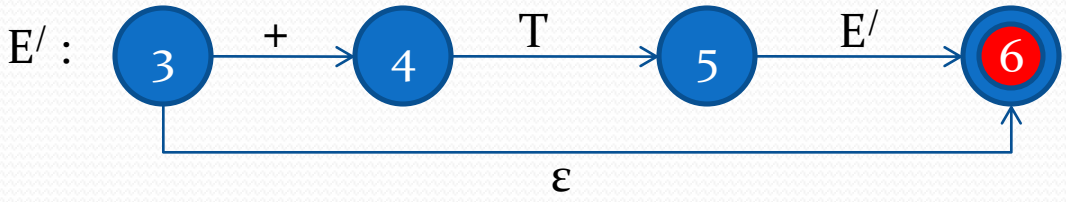


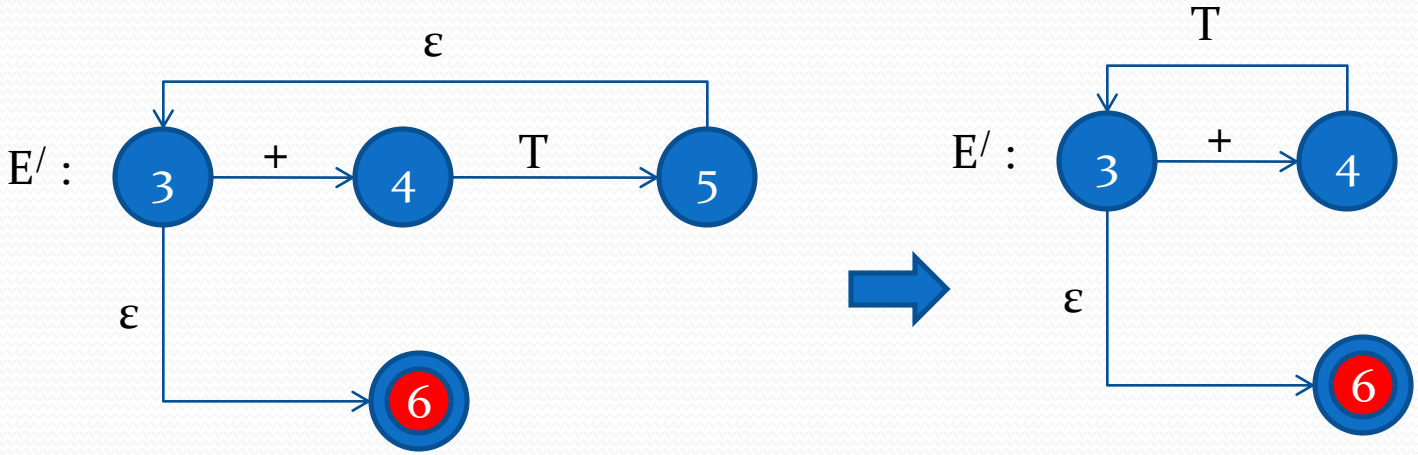
$F \rightarrow id \mid (E)$



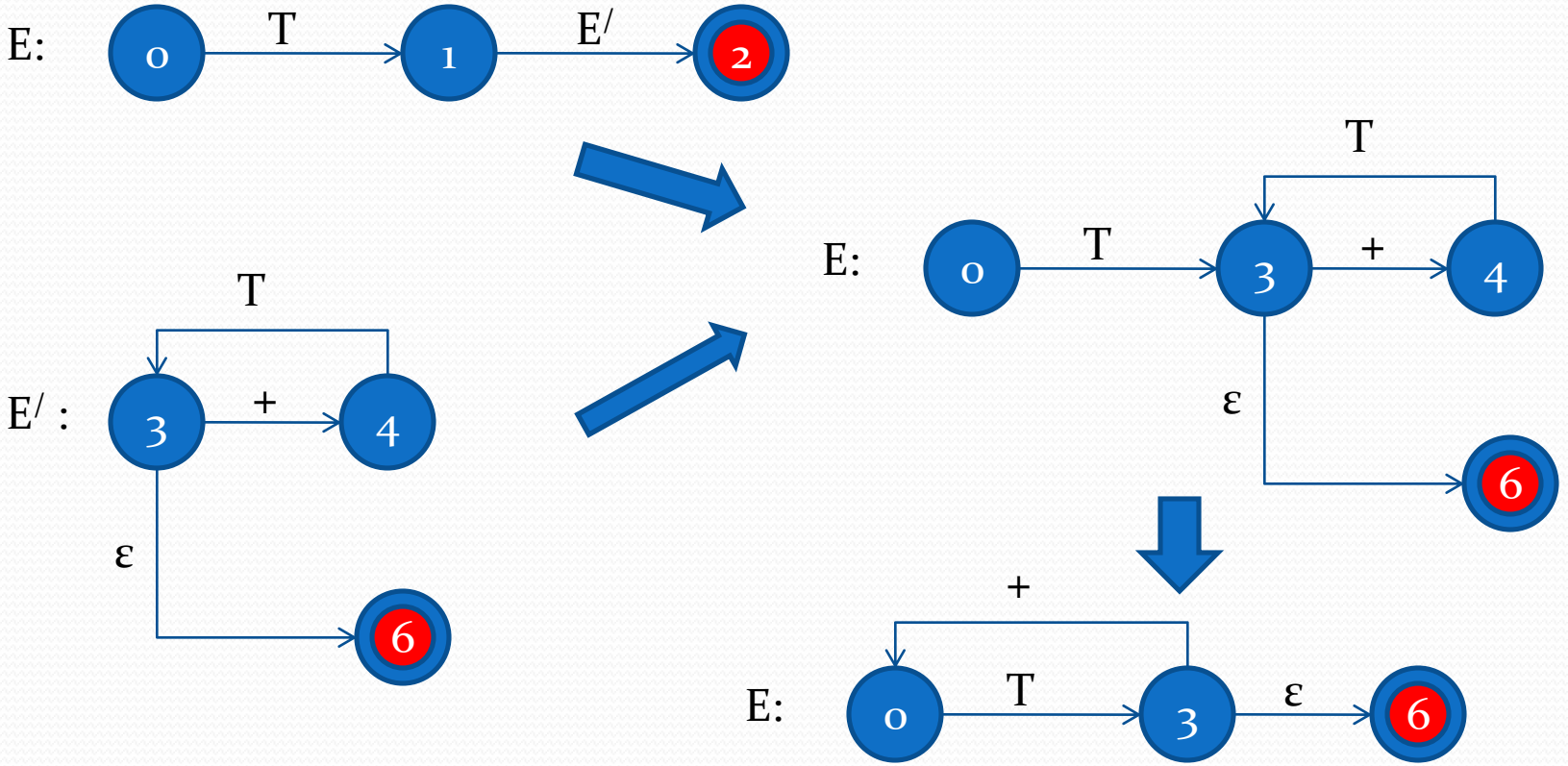
Predictive Parser

- Rule 2: Optimize the dfa by reducing the number of states, yielding the final transition diagram.

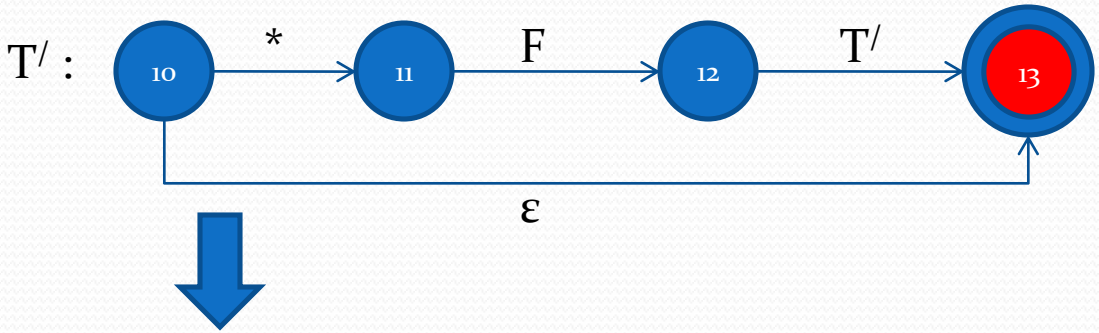


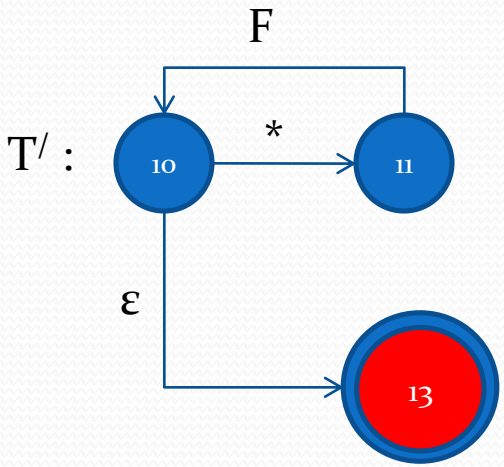


Predictive Parser

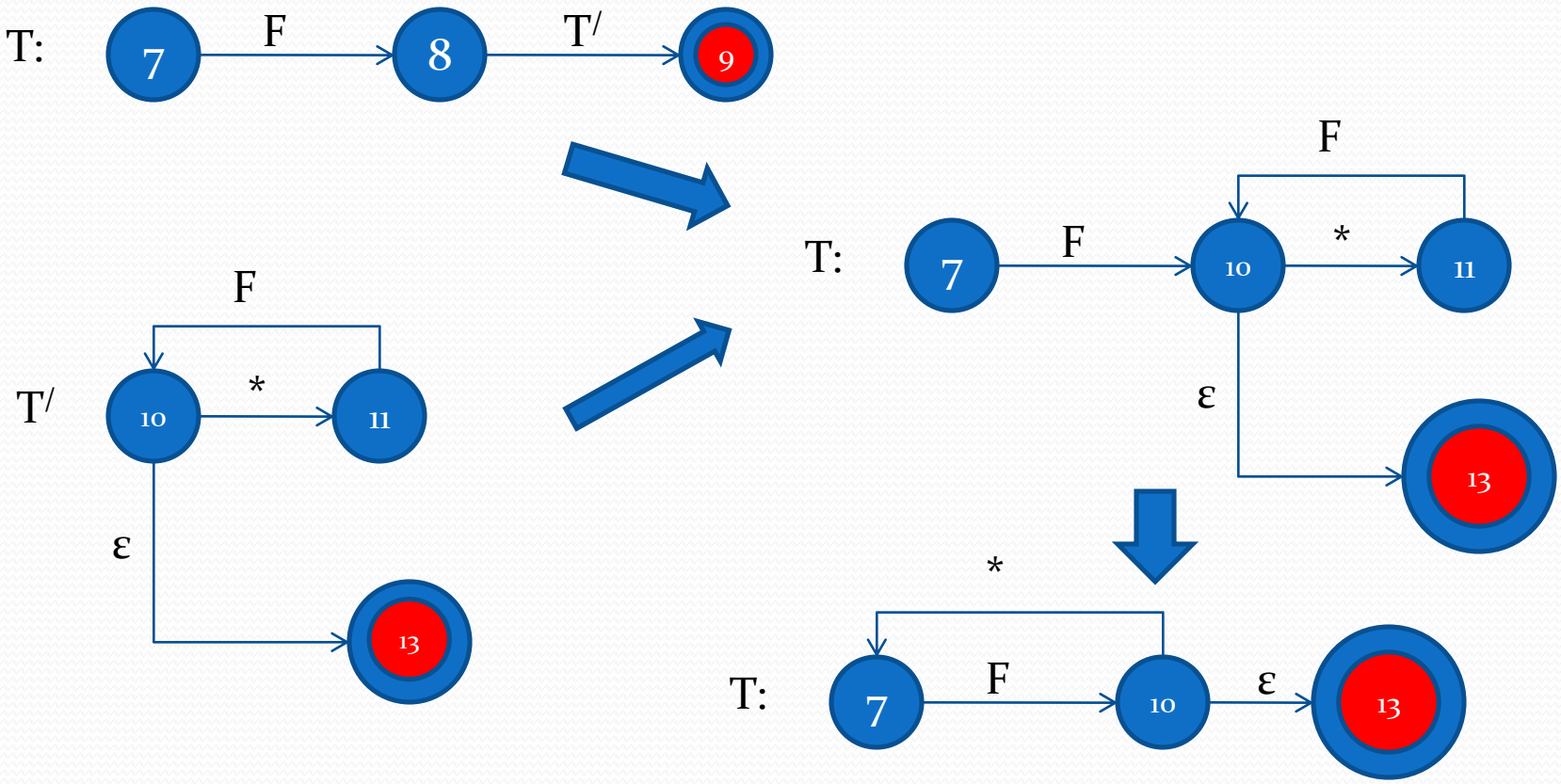


Predictive Parser

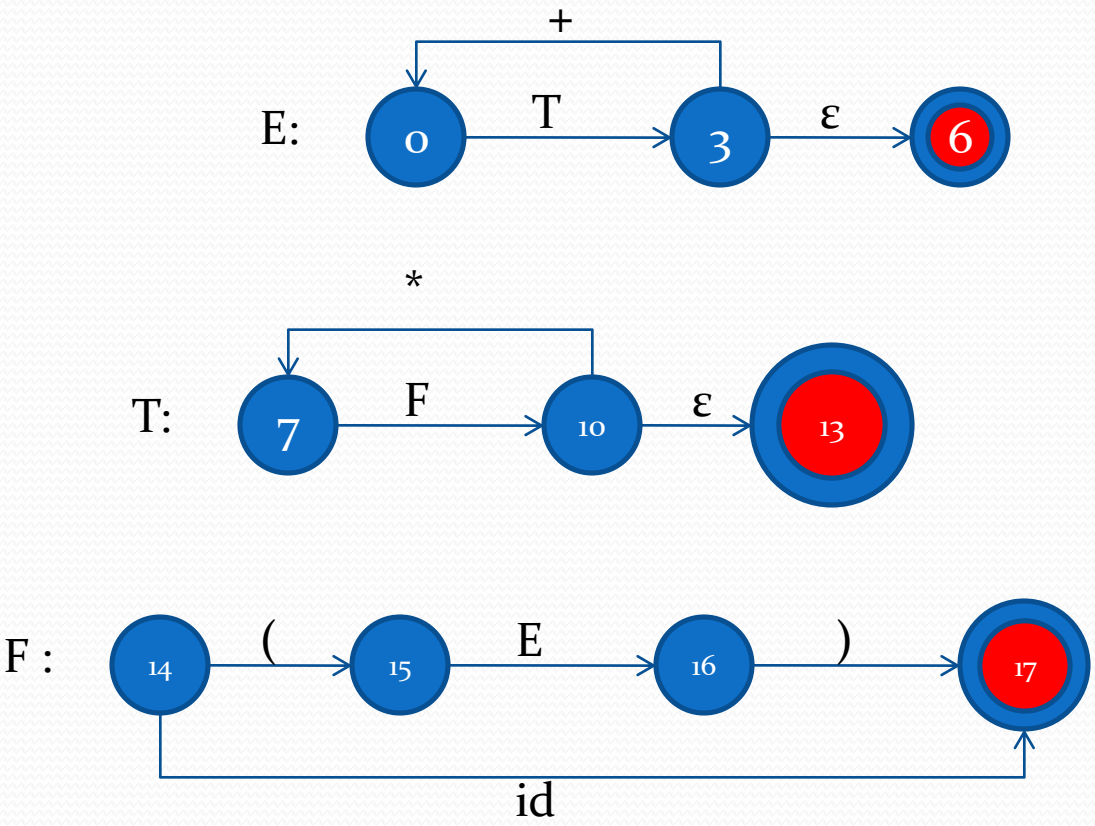




Predictive Parser



Predictive Parser



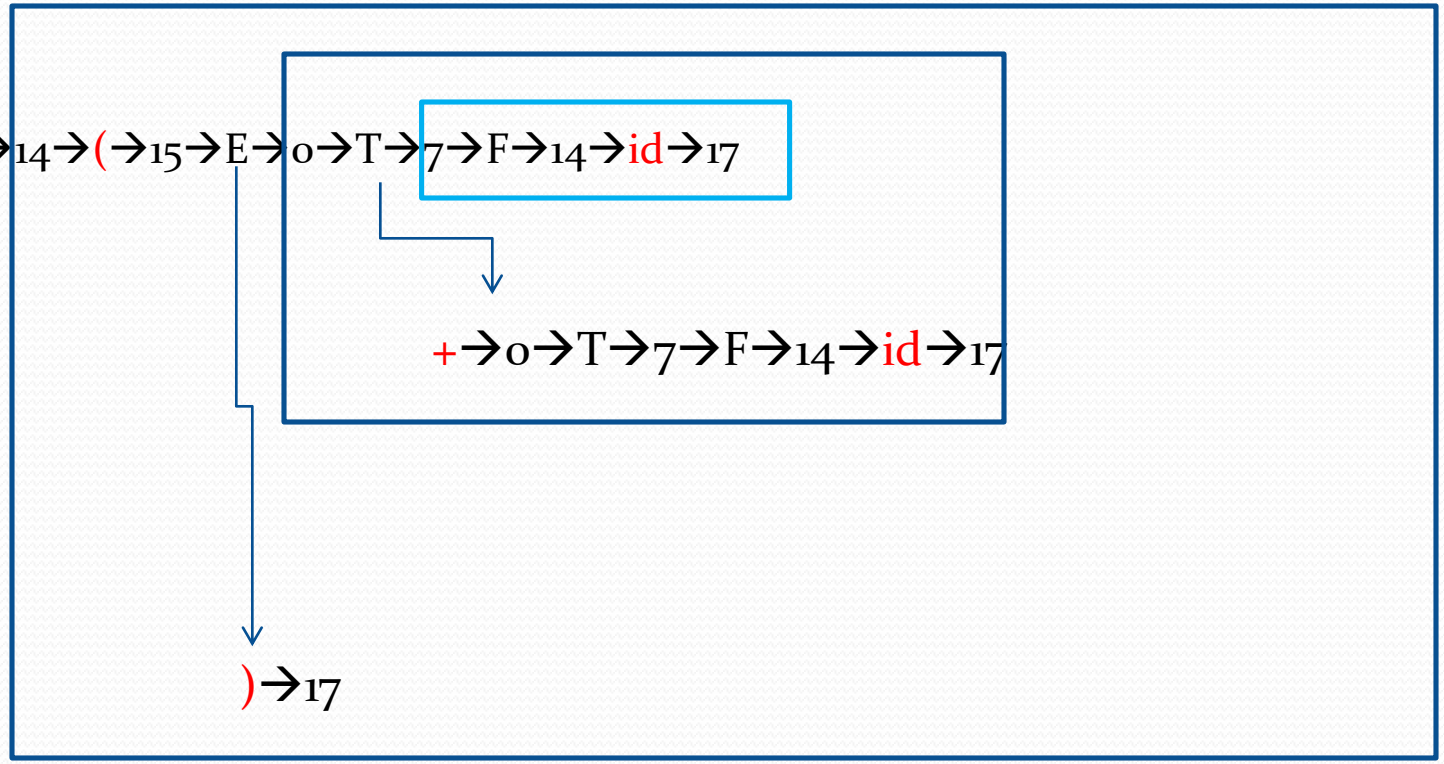
Predictive Parser

Rule 3: To parse a string, simulate the string on the transition diagram.

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non – terminal comes go to the state of the “dfa” of the non-terminal and return on reaching the final state.
- Return to the original “dfa” and continue parsing
- If on completion(reading input string completely), you reach a final state, string is successfully parsed.

Predictive Parser

- (id+id)*id
- 0→T→7→F→14→(→15→E→0→T→7→F→14→id→17



String parsing complete. And reached to final state. Parsing successful.

Assignment

- Parse: id*(id+id*id)

- Consider the grammar

P→ P^E|E ^: power

E → E+T|P|T

T → T*F | F

F → id | (E)|F^E

Parse the string $id^{(id+id)^{(id*id)}}$

Removing Left Recursion

P→EP/

P/ →^EP/ | ε

E→PE/| T E/

E/ → +T E/ | ε

T → F T/

T' → *F T/ | ε

F → idF/ | (E) F/

F/ →^E F/ | ε

Predictive Parser

Disadvantages of Predictive Parser:

- It is inherently a **recursive parser**, so it consumes a lot of memory as the stack grows.
- Doing **optimization** may not be as simple as the complexity of grammar grows
- To remove this recursion, we use **LL-parser**, which uses a table for lookup.

III.A.2
Recursive Predictive Parsing

Recursive Predictive Parsing

- This is a top down parsing method where we execute a set of recursive set of procedures to process the input.
- A procedure is associated with a nonterminal of a grammar.
- The lookahead symbol unambiguously determines the procedure selected for each nonterminal .
- The sequence of procedures called in processing the input implicitly defines a parse tree for the input.

Consider the grammar:

S → cAd | bd

A → ab | e

Recursive Predictive Parsing

PSEUDO CODE for a predictive parser

```
function match(token t)
{
if lookahead = t          then
    lookahead = nexttoken()
 else error
}
```

S → cAd | bd

```
function S
{
if lookahead is in { c }
            match(c) , A(),match (d);
else if lookahead is in {b }
            match(b) , match(d);
else error
}
```

A → ab | e

```
function A
{
if lookahead is in { a }
            match(a) , match(b);
else if lookahead is in { e }
            match(e);
else if error
}
```

Recursive Predictive Parsing

- input string: “ced”

- The function match() compares the current look ahead symbol with the argument token and if matched changes the look ahead symbol by advancing the input pointer.
- Parsing begins with a call to the procedure for the starting nonterminal S in our grammar. Because the lookahead 'c' is in the set { c } , the function S executes the code:

if lookahead is in { c }

match(c) , A(),match (d);

- once it matched 'c' , the function A() is called and checks out that the next input symbol 'e' is then in the set { e } , it executes the code :

else if lookahead is in { e }

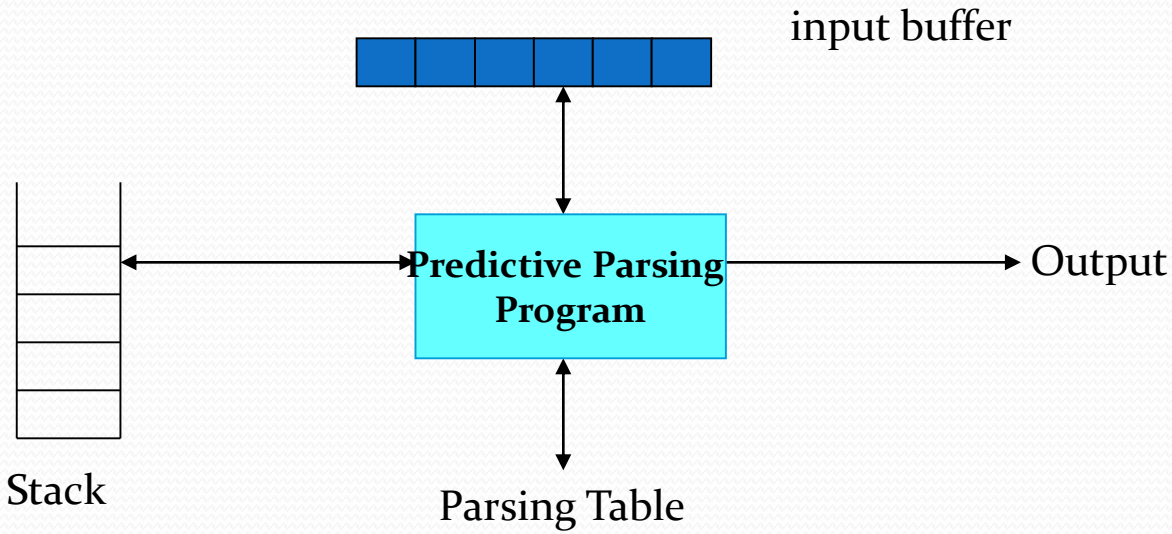
match(e);

- After the matching of 'e' is over it returns from the function A() and matches the next token with 'd'.

III.A.3
LL(1) parsing

Non-Recursive Predictive Parsing -- LL(1) Parser

- It is possible to built a non-recursive predictive parser by maintaining a stack .
- The key problem of predictive parsing is that of determining the productions to be applied for a non terminal.
- Here a table is kept which stores the productions.
- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.
- Left to Right Scanning-Left most derivation in Reverse-1 symbol look ahead



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)
   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal
   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

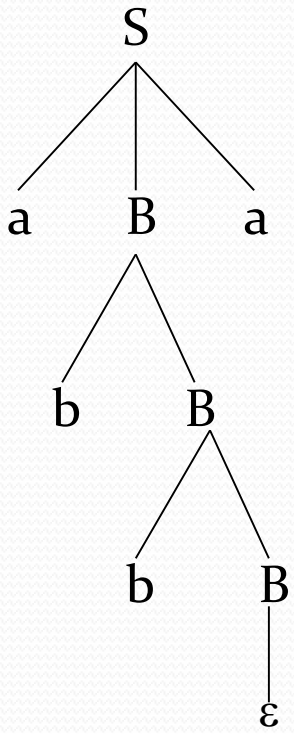
LL(1) Parsing Table

stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



Compare with LL(1) parsing it is reverse.

parse tree

LL(1) Parser – Example2

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

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

LL(1) Parser – Example2

stack	input	output
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E' T'F	id+id$	$F \rightarrow id$
$ E' T'id	id+id$	
$ E' T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E' T+	+id$	
$ E' T	id$	$T \rightarrow FT'$
$ E' T' F	id$	$F \rightarrow id$
$ E' T'id	id$	
$ E' T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

Assignment

Parse acbbgfh using the following parsing table:

	a	b	c	g	f	h	$
S	S→aBDh						
B			B→cC				
C		C→bC		C→ε	C→ε	C→ε	
D				D→EF	D→EF	D→EF	
E				E→g	E→ε	E→ε	
F					F→f	F→ε	

S→aBDh
B→cC
C→bC/ε
D→EF
E→g/ ε
F→f/ ε

Assignment

- Parse (int*int)+int using the following parsing table:

	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow int\ Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .

- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.

Compute FIRST for Any String X

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule ➔ ε is in FIRST(X)
- If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule ➔

FIRST(X) = FIRST(Y1) if Y1 does not derive ε

If Y1 derive ε then

FIRST(X) = FIRST(Y1) U FIRST(Y2) U ..... U FIRST(Yn)

Compute FIRST for Any String X (In Gist.)

- FIRST(∈) = ∈
- FIRST(a) = a
- FIRST(αβ) = FIRST(α) U FIRST(β) if FIRST(α) contains ∈
  else FIRST(α)
- FIRST(N)= FIRST(α_1)U FIRST(α_2)U.....FIRST(α_n)
  Where the productions of N are
  N →α_1/ α_2/ ...../ α_n

FIRST Example

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

The Grammar contains
Terminals= {+, *, (,), id}
Non Terminals = {E, E', T, T', F}

FIRST(∈) = {∈} **FIRST(+)={+}**
FIRST(*)={*} **FIRST('(')= {(}**
FIRST(')') = {)} **FIRST(id) ={id}**

FIRST Example

FIRST(E)= FIRST(T) U FIRST(E/) if T derive ε
else
FIRST(T)
T does not derive ε Thus FIRST(E)= FIRST(T)

FIRST(T) = FIRST(F) U FIRST(T/) if F derive ε
else
FIRST(F)
F does not derive ε Thus FIRST(T)= FIRST(F)

FIRST(F)=FIRST((E)) U FIRST(id)

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

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

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

FIRST Example

FIRST(E')= FIRST($+TE'$) U FIRST(ε) As $E' \rightarrow +TE' \mid \varepsilon$

FIRST($+TE'$)= FIRST($+$)={$+$}.

FIRST(E') = {$+$, ε}.

FIRST(T')= FIRST($*FT'$) U FIRST(ε) As $T' \rightarrow *FT' \mid \varepsilon$

FIRST($*FT'$)= FIRST($*$)={$*$}.

FIRST(T') = {$*$, ε}.

FIRST Example

FIRST(F) = { (, id}
FIRST(T′) = {*, ε}
FIRST(T) = { (, id}
FIRST(E′) = {+, ε}
FIRST(E) = { (, id}

Assignment

1.

S→aBDh

B→cC

C→bC/ε

D→EF

E→g/ ε

F→f/ ε

Construct FIRST for the terminal and non-terminal symbols.

2. Write a C program to construct the FIRST of the grammar symbols .

Compute FOLLOW (for non-terminals)

- To reject grammars that are problematical for predictive parsing , we introduce FOLLOW set of non-terminals

RULES

- If S is the start symbol ➔ $ is in FOLLOW(S)
- if $A \rightarrow \alpha B \beta$ is a production rule ➔ everything in FIRST(β) is FOLLOW(B) except ε
- If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β)) ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

If $X \rightarrow Y_1 Y_2 \ldots\ldots Y_k$

FIRST(X)= First(Y_1) U First (Y_2 .....Y_k) if First (Y_1) contain ∈

= First(Y_1) U First (Y_2) U First (Y_3 .....Y_k) if First (Y_1) and First(Y_2) contain ∈

∈ will be added to First(X) if all of First(Y_1), First (Y_2), ... First(Y_k) contain ∈

Same will be for First(αβδγ) .

FOLLOW Example

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

Consider $E \rightarrow TE'$. It is in the form $A \rightarrow \alpha B$.
So everything in FOLLOW(E) is in FOLLOW(E').

As E is the start symbol so FOLLOW(E) = {$}
FOLLOW(E') = {$}
FOLLOW(E) is a subset of FOLLOW(E')

Consider $E' \rightarrow +TE'$. It is in the form $A \rightarrow \alpha B\beta$.
Where FIRST(E') contains $\in$.
So everything in FIRST(E') except $\in$ is in FOLLOW(T)

FIRST(E') = {+, ε} So, FOLLOW(T) = {+}
As FIRST(E') contains $\in$ then everything in FOLLOW(E') is in FOLLOW(T).
FOLLOW(E') is a subset of FOLLOW(T)

Consider $T \rightarrow FT'$. It is in the form $A \rightarrow \alpha B$.
So everything in FOLOW(T) is in FOLLOW(T').
FOLLOW(T) is a subset of FOLLOW(T')

FOLLOW Example

Consider T/ → *FT'. It is in the form A →αBβ, where FIRST(T/) contains ∈.
So everything in FIRST(T/) except ∈ is in FOLLOW(F).

FIRST(T′) = {*, ε} So, FOLLOW(F)={*}

As FIRST(T') contains ∈, everything in FOLLOW(T') is in FOLLOW(F)
FOLLOW(T/) is a subset of FOLLOW(F)

Consider F →(E). It is in the form A → αBβ.
So everything in FIRST(')') except ∈ is in FOLLOW(E)

FOLLOW(E) ={')'}

FOLLOW Example

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) } FOLLOW(E) is a subset of FOLLOW(E/)

FOLLOW(T) = { +,), $ } FOLLOW(E/) is a subset of FOLLOW(T)

FOLLOW(T') = { +,), $ } FOLLOW(T) is a subset of FOLLOW(T/)

FOLLOW(F) = {+, *,), $ } FOLLOW(T/) is a subset of FOLLOW(F)

Assignment

1.

S→aBDh

B→cC

C→bC/ε

D→EF

E→g/ ε

F→f/ ε

Construct FOLLOW for the non-terminal symbols.

2.

S → aBa

B → bB | ε

Construct FOLLOW for the non-terminal symbols.

Assignment

3.
$S \rightarrow i\ C\ t\ S\ E \mid a$
$E \rightarrow e\ S \mid \varepsilon$
$C \rightarrow b$

Construct FOLLOW for the non-terminal symbols

Ans.
FOLLOW(S) = { $,e }
FOLLOW(E) = { $,e }
FOLLOW(C) = { t }

S →	**i**	**C**	**t S E**
A	α	B	β

S →	**iCt**	**S**	**E**
A	α	B	β

S →	**iCtS**	**E**
A	α	B

Assignment

4. Write a C program to find the Follow of the Non-Terminals of a Grammar.

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G

  i) for each terminal a in FIRST(α)

  ➔ add $A \rightarrow \alpha$ to M[A,a]

  ii)

  a) If ε in FIRST(α)

  ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]

  b)If ε in FIRST(α) and $ in FOLLOW(A)

  ➔ add $A \rightarrow \alpha$ to M[A,$]

- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

- Consider the grammar

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

The grammar contains six terminal symbols {+, *, (,), id, $} &
five non-terminal symbols {E, E/, T, T/,F}.

So the table is made of 5 rows & six columns.

Consider E → TE/. FIRST(TE/) = FIRST(T) = {(, id}
The field of the table labeled M[E,(] & M[E, id] are filled with E → TE/

Consider E → +TE/. FIRST(+TE/) = FIRST(+) = {+}
The field of the table labeled M[E/,+] is filled with E → +TE/

Consider E/ → ∈. FIRST(∈) = {∈} and FOLLOW(E/) contains $.
The field of the table labeled M[E/ ,)] & M[E/, $] are filled with E/ → ∈

Constructing LL(1) Parsing Table -- Example

Consider T → FT′ . FIRST(FT′) = FIRST(F) = {(, id}.
The field of the table labeled M[T, (] & M[T, id]are filled with T → FT′

Consider T′ → *FT′. FIRST(*FT′) = FIRST(*) = {*}.
The field of the table labeled M[T′, *] is filled with T′→*FT′

Consider T′ → ∈. FIRST(∈) = {∈} & FOLLOW(T′) contains $.
The field of the table labeled M[T′, +] , M[T′,)] & M[T′, $] is filled with T′ →∈

Consider F → (E). FIRST('(') = {(}. So the field of the table labeled M[F, (] is filled with F → (E)

Consider F → (E). FIRST('(') = {(}. So the field of the table labeled M[F, (] is filled with F → (E)

Consider F → id. FIRST(id) = {id}. So the field of the table labeled M[F, id] is filled with F → id

Constructing LL(1) Parsing Table -- Example

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

Assignment

1. Construct LL(1) parsing table for the following grammar

i)

S→aBDh

B→cC

C→bC/ε

D→EF

E→g/ ε

F→f/ ε

ii)

S → aBa

B → bB | ε

2. Write a C program to construct LL(1) parsing table of a grammar.

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

$S \rightarrow i\ C\ t\ S\ E \mid a$
$E \rightarrow e\ S \mid \varepsilon$
$C \rightarrow b$

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

FOLLOW(S) = { $,e }
FOLLOW(E) = { $,e }
FOLLOW(C) = { t }

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow$ iCtSE		
E			$E \rightarrow e\ S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Assignment

- If β is ε then a grammar with production rules $A \rightarrow A\alpha \mid \beta$ can not be a LL(1) grammar.

Provide proper judgment for this statement.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.
2. At most one of α and β can derive to ε.
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Assignment

If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Provide proper judgment for this statement
for a grammar G to be LL(1) for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \varepsilon$
$A \rightarrow a \mid cAd$

FOLLOW(S)={$}
FOLLOW(A)={b,d}

	a	b	c	d	e	$
S	$S \rightarrow AbS$	***sync***	$S \rightarrow AbS$	***sync***	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	***sync***	$A \rightarrow cAd$	***sync***	***sync***	***sync***

stack	**input**	**output**
$S	ceadb$	$S \rightarrow AbS$
$SbA	ceadb$	$A \rightarrow cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

Panic-Mode Error Recovery - Example

	a	b	c	d	e	$
S	S → AbS	***sync***	S → AbS	***sync***	S → e	S → ε
A	A → a	***sync***	A → cAd	***sync***	***sync***	***sync***

stack	**input**	**output**
$S	aab$	S → AbS
$SbA	aab$	A → a
$Sba	aab$	pop
$Sb	ab$	Error: missing b
Pop b from stack & Generate an Error message		
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	
$Sb	b$	
$S	$	S → ε
$	$	

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

III.B
Bottom Up Parsing

Bottom Up Parsing

- It is the reverse process of top-down parsing.
- Bottom Up parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the right most derivation of w in reverse.
- This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding toward the root.

Consider the grammar

S → aABe

A → Abc/ b

B → d

Let the string generated by the grammar abbcde is parsed in bottom up.

abbcde ⇒ aAbcde ⇒ aAde ⇒ aABe ⇒ S

A → b **A → Abc** **B → d** **S → aABe**

Handle

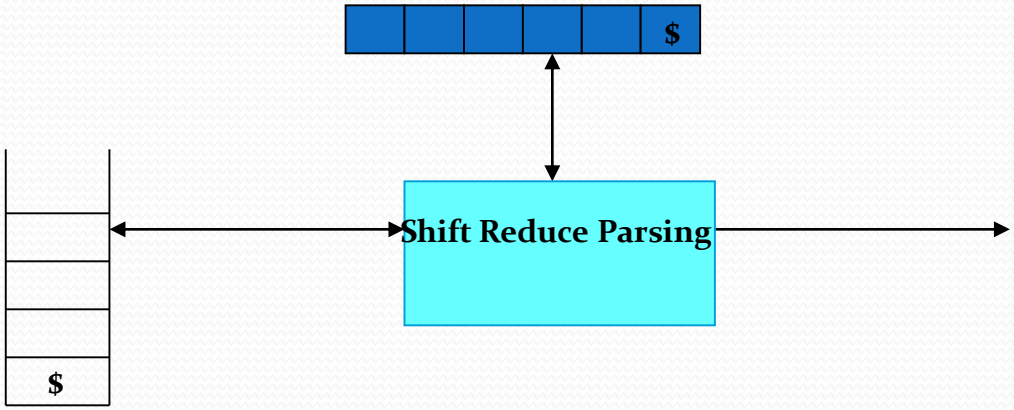
- A handle is a right sentential sequence which is the rightmost derivation within the string being parsed.
- A handle of a right sentential form γ is a production A → β and a position of β in γ.
- The string β will be found and replaced by A to produce the previous right sentential form in the right most derivation of γ

In the previous example B → d is the handle of the string aAde.

III.B.1
Shift Reduce Parsing

Shift Reduce Parsing

- Shift Reduce technique uses bottom up parsing technique.
- A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack.
- When the handle appears on the top of the stack it performs reduction.



- The stack in shift reduce parser hold the grammar symbols. Initially the stack is empty and $ is placed at the bottom of the stack
- Input buffer holds the string w to be parsed. The string is terminated by right end marker $.

Operation of Shift Reduce Parsing

- The parser operates by shifting zero or more input symbols onto the stack until a handle β is achieved on the top of the stack.
- The parser then reduces β to the left side of the appropriate production.
- The parser repeats this cycle until it detects an error or until the stack contains the start symbol and input contains only $.

Example

Construct a shift reduce parser for the string id1+id2*id3 from the grammar

E → E+E
E → E*E
E → id1 / id2 / id3

Shift Reduce Parsing

Stack	Input	Action
$	id1+id2*id3 $	shift id1
$ id1	+id2*id3 $	Reduce by E → id1
$E	+id2*id3 $	shift +
$E+	id2*id3 $	shift id2
$E+id2	*id3 $	Reduce by E → id2
$E+E	*id3 $	shift *
$E+E*	id3 $	shift id3
$E+E* id3	$	Reduce by E → id3
$E+E*E	$	Reduce by E → E*E
$E+E	$	Reduce by E → E+E
$E	$	Accept

Viable Prefixes: The set of prefixes of right sentential forms that can appear on the top of the stack of a shift reduce parser are called viable prefixes.

Problem/ Conflicts during shift reduce parsing

The two most common problems are

Shift reduce conflict: When the parser can't decide whether to shift a token onto the stack or reduce a handle.

Reduce reduce conflict: When the parser can't decide which reduction to make.

Example:

S→ if E then S/ if E then S else S/ other.

Stack	Input	Action
$	if E then S else S $	shift if
$ if	E then S else S $	shift E
$ if E	then S else S $	shift then
$ if E then	S else S $	shift S
$ if E then S	else S $	

This is called shift reduce conflict.

? ? ? ?
Shift else or reduce by stmt → if E then S

Problem/ Conflicts during shift reduce parsing

Reduce reduce conflict are less common

Example

E → E1 + E2 / E3 * E4

E, E1, E2, E3, E4 → id

Stack	Input	Action
$	id+id*id$	Shift id
$id	+id*id$	Reduce by which of the productions??

This is called reduce reduce conflict.

III.B.2
Operator Precedence Parsing

Operator Precedence Parsing

- Operator Grammar: A grammar is said to be operator Grammar if there does not exist any production rule with right hand side

  a) As ε

  b) two non terminals appearing consecutively i.e. without any terminal between them.

Operator precedence parsing can be applied to operator grammar.

Example: E→ EAE/ (E)/ -E/ id

A → +|-|*| /|↑

The previous grammar is not an operator grammar, because right hand side EAE has three consecutive non-terminals.

However if we substitute for A each of its alternatives, we obtain the following operator grammar.

E→ E+E/ E-E/ E*E/ E/E / E↑E / (E) / -E /id

Here, no two consecutive non-terminals are found in any production.

Operator Precedence Parsing

- In operator precedence parsing we define three disjoint precedence relations <· , =·, ·> between certain pairs of terminals.
- a ·> b means a has higher precedence than b.
- a =· b means a has same precedence as b
- a<·b means a has less precedence than b.

For the grammar E→ E+E

E → E*E

E → id

The precedence table for the grammar is

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

$ is the end marker of the string

Operator Precedence Parsing

Rule:

1) Rewrite the given string with precedence relation with $ as two end markers.
2) Scan the resulting sequence left to right until first ·> is achieved and then backtrack to immediately left <· ignoring any =· relation.
3) The handle exist in between <· and ·> , in terms of terminals.

- To parse $id+id*id$ now we apply these precedence relation ignoring the non-terminals.

Ignore

id+id*id ➔ $ <·id ·>+ <·id·>* <·id·>$

➔ $ <·id ·>+ <·id·>* <·id·>$

Found

Traverse Back until a <· is found

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

➔ $ E+ <·id·>* <·id·>$

➔ $ E+E*<·id·>$ Reduce it by E because there is a production rule E➔id

➔ $ E+E*E$

Again apply the precedence relation from the table $ <·E+ <·E*E·>$

➔ $ <·E+ E ·>$

➔ E

Constructing Precedence Table

Step I: Construct *Firstop* and *Lastop* for each non-terminal.

Rule for Firstop:

a) First Terminal appearing at the right hand side production.
b) When a Non-Terminal is the first symbol at the right hand side include both it and the first terminal symbol followed by it.

Question: If right hand side starts with a non-terminal ;are you sure that a terminal will follow it?

Rule for Lastop:

a) First terminal appearing at right to left scanning of the right hand side production.
b) If a Non-Terminal is the last symbol at the right hand side include both it and the first terminal symbol precede by it.

Some Problem in Precedence table construction.
Try by removing unit production, e production etc..

Constructing Precedence Table

- Step II: Compute Firstop+ and Lastop+ using Warshall's Closure Algorithm.

Take each nonterminals in any order and look into the Firstop list. Add its own Firstop symbols to other Nonterminal's Firstop+ list in which it occours.

Simillar for Lastop+

Example: If Firstop(NT1) ={α,γ} Both are terminal symbols.
Let Firstop(NT2) = {δ,NT1}.

Then Firstop+(NT1) ={α,γ}
Firstop+(NT2) = {δ,NT1,α,γ}.

- Delete the non-terminals from the list.

Constructing Precedence Table

- StepIII: Construct the precedence matrix

a) When a terminal α immediately precedes non-terminal X in any production put α <·ρ, where ρ is any terminal in the Firstop+ list of X.

b) When a terminal α immediately follow non-terminal X in any production put α ·>ρ, where ρ is any terminal in the Lastpos+ list of X.

c) Whenever a production αXγ occur in any production put α =· γ.

Step IV: Put the relation $ <· α and α ·> $ for all terminals in the Firstop+ and Lastop+ list respectively , for S.

Example

- S →A
  A→T | A+T | A-T
  T →F |T*F | T/F
  F → P | P^F
  P → i | n | (A)

Remove unit production then try…

Symbol	Firstop	Lastop
S	A	A
A	T+A-	T+-
T	F*T/	F*/
F	P^	P^F
P	i n(	i n)

Symbol	Firstop+	Lastop+
S	T + A – F */ P ^ i n (	AT + - F * /P ^ i n)
A	T + A – F */ P ^ i n (	T + - F * /P ^ i n)
T	F*T/P^i n (	F*/P^i n)
F	P^i n (	P^F i n)
P	i n(	i n)

Example cont…

Removing the non-terminal symbols we get

Symbol	Firstop+	Lastop+
S	+ – */ ^ i n (	+ - * / ^ i n)
A	+ – */ ^ i n (	+ - * / ^ i n)
T	* / ^ i n (	* / ^i n)
F	^i n (	^ i n)
P	i n(	i n)

Example cont…

Finally the Precedence table is.

	$	(	)	i	n	^	*	/	+	-
$		<·		<·	<·	<·	<·	<·	<·	<·
(		<·	=·	<·	<·	<·	<·	<·	<·	<·
)	·>		·>			·>	·>	·>	·>	·>
i	·>		·>			·>	·>	·>	·>	·>
n	·>		·>			·>	·>	·>	·>	·>
^	·>	<·	·>	<·	<·	<·	·>		·>	·>
*	·>	<·	·>	<·	<·	<·	·>	·>	·>	·>
/	·>	<·	·>	<·	<·	<·	·>	·>	·>	·>
+	·>	<·	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	<·	·>	<·	<·	<·	<·	<·	·>	·>

Assignment

1. Construct precedence table for the operator in the following productions

E→ E+E

E → E*E

E → id

2.

S→ (L)|a

L→L , S|S whether it will be L → S, S?

3.

E → E or E

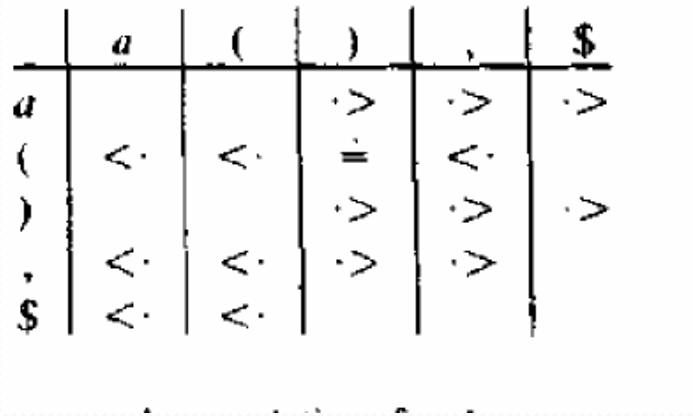
E → E and E

E → not E

E → (E)

E → id

Construct Operator precedence table and parse (a,(a,a)).



	a	(	)	,	$
a			·>	·>	·>
(	<·	<·	=·	<·	
)			·>	·>	·>
,	<·	<·	·>	·>	
$	<·	<·			

III.B.3
LR Parsing

LR parsing

- One of the best methods for syntactic recognition of programming languages is LR parsing.
- It is also a shift reduce parsing technique.
- Here L stands for left to right scanning and R stands for rightmost derivation in reverse.
- Here LR parsing, we are speaking about are LR(1) parsing- i.e. one symbol lookahead. In general we have LR(k) parsing with k symbols lookahead.

Advantages of LR parsing:

a) An LR parser can recognize virtually all programming language constructs written in CFG.
b) It is most general non-backtracking technique known.
c) It can be implemented in very efficient manner.
d) The language it can recognize is a proper superset of that for predictive parser.
e) It can detect syntax error quickly.

Disadvantage:

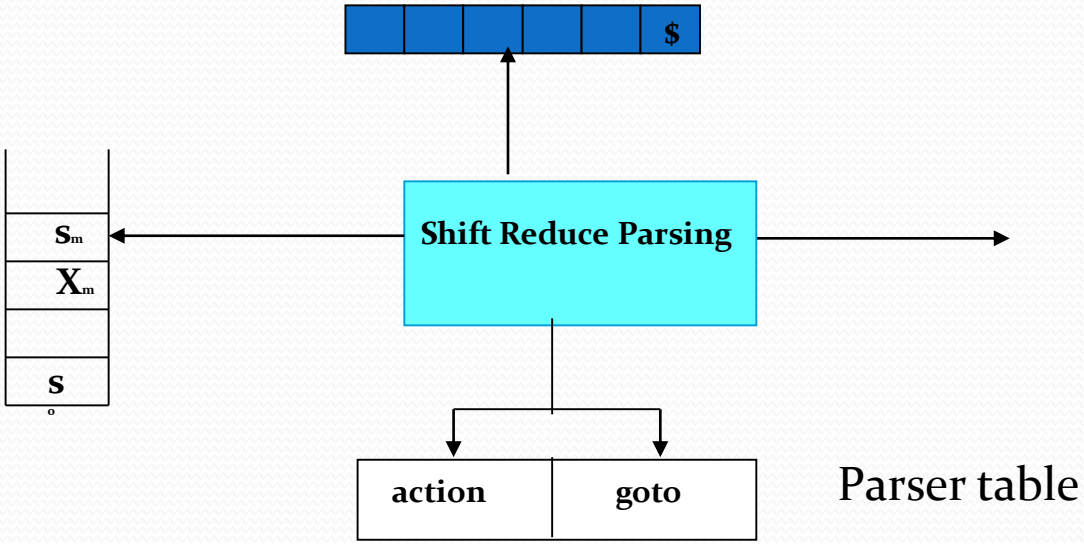
It is very hardwork to create manually LR parsing table.

LR parsing Methods

- There are three different methods to perform LR parsing.

1) SLR, stands for simple LR. It is easy to implement but less powerful than other parsing methods.
2) Canonical LR is the most general & powerful, but it is costly to implement. For a same grammar, the canonical LR parser has got much more number of states as compared to the simple LR parser.
3) LALR stands for lookahead LR. It is a mix of SLR and canonical LR, but can be implemented efficiently. It contains same number of states as the simple LR parser for the same grammar.

LR parsing algorithm



Parser table

Parsing table of LR parser consists of two parts a) action and b) goto

The stack of LR parser consists of grammar symbol and states.

Input tape contains of input string ended by $

LR parsing algorithm

LR parser works as follows:

a) It determines S_m, the state currently on top of the stack and the current input symbol a_i .

b) Then it consults action[S_m, a_i] in the LR parsing table, which can have one of four values

i) Shift S, where 'S' is a state. [Shift the current input in the stack and state number on top]

ii) Reduce by a grammar production $A \rightarrow \beta$

iii) accept

iv) error

Shift pushes the next input symbol and next state onto the stack.

Reduce matches the handle to some production $A \rightarrow \beta$. It then pops $r = 2*|\beta|$ elements off the stack and pushes A and S= goto[S_{m-r}, A].

Example of LR parsing

Parse the string id*id+id generated from the following grammar

1) E→ E+T
2) E→T
3) T→ T*F
4) T→F
5) F→(E)
6) F→id

	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			g1	g2	g3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			g8	g2	g3
5		r6	r6		r6	r6			
6	S5			S4				g9	g3
7	S5			S4					g10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example of LR parsing

	Stack	Input	Action
1.	0	id*id+id$	Shift[S5]
2.	0id5	*id+id$	reduce[r6] then g3 [0 & F has goto entry 3]
3.	0F3	*id+id$	reduce r4 then g2 [0 & T has goto entry 2]
4.	0T2	*id+id$	shift[S7]
5.	0T2*7	id+id$	shift[S5]
6.	0T2*7id5	+id$	reduce[r6] then g10
7.	0T2*7F10	+id$	reduce[r3] then g2
8.	0T2	+id$	reduce[r2] then g1
9.	0E1	+id$	shift[S6]
10.	0E1+6	id$	shift[S5]
11.	0E1+6id5	$	reduce[r6] then g3
12.	0E1+6F3	$	reduce[r4] then g9
13.	0E1+6T9	$	reduce[r1] then g1
14.	0E1	$	accept

Assignment

Grammar is

S →aABe
A →Abc
A →b
B →d

	Action						Goto		
	a	b	c	d	e	$	S	A	B
0	S2						1		
1						acc			
2		S4						3	May be error
3		S6		S7					5
4		r4		r4					
5					S8				
6			S9						
7					r5				
8						r2			
9		r3		r3					

Parse the string abbcbcde using the production rules.

LR Parsing Table Construction

LR parsing table construction consists of

I. Augmented grammar formation

II. Construction the state of FA

State construction of FA using i) item grammar and ii) closure operation from the augmented grammar.

III. Shift and Goto part Construction

IV) Reduce part construction

I) **Augmented grammar formation:** Add a new non-terminal E/ as new start symbol with a production E/ → E [E was the original Non Terminal]

LR Parsing Table Construction

II. Construction the state of NFA

i) Item Grammar Construction:

- item of a grammar G is a production rule in G with a dot at some position of the right hand side of the production.
- Item (The presence of . [dot]) indicates how much of the production is parsed so far.

$E \rightarrow E + T$

Item of the Grammars are.

$E \rightarrow .E + T$

$E \rightarrow E. + T$

$E \rightarrow E + .T$

$E \rightarrow E + T.$

If there is a production $T \rightarrow \varepsilon$ in the grammar then $T \rightarrow .$ is an item of the grammar.

LR Parsing Table Construction

ii) Closure Operation: If I is the set of items of a grammar G then Closure of I is a set of item constructed by the following rules.

- Closure(I) contains every item in I.
- If $A \rightarrow \alpha . B \beta$ **exist in Closure(I) and** $B \rightarrow \gamma$ **is a production, then** $B \rightarrow .\gamma$ is also added in the Closure(I).

LR Parsing Table Construction

III. Shift and *Goto Operation:* If a set of item is I, and B is a grammar symbol, the operation

goto(I, B) = closure (A → α B. β) if A → α . B β is in I

In FA if goto(Ii, T) → Ij then Action[Ii, T] = Sj.

If goto(Ii, NT) → Ij then Goto[Ii, NT] = gj

IV. Reduce Part Construction

- formulate the FOLLOW of the non-terminals.
- If Ii contains a production in the form NT → α . , and FOLLOW(NT) = {Tj}, then Action [Ii, Tj] = r_k. Where k is the production number of NT → α .
- If Ii contains a production S → α . , where S is the new start symbol and FOLLOW(S) = $, then Action[Ii, $] = Accept.

Example

Construct SLR parsing table for the following grammar

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

Step I: The augmented grammar is

E′ → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F→ id

Example

Step II: Construction of the states of the NFA

i) & ii) Item Grammar Construction with Closure Operation

E′ → . E in I. The closure(I) is the set as follow.

{ E′ → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F→. id }

Mark it Io.

Example

State Io with all grammar symbols as Input.

goto(Io, E) = closure(E′ → E., E → E. + T) = { E′ → E., E → E. + T} = **I_1**

goto(Io, T) = closure(E → T., T → T. * F) = { E → T., T → T. * F} = **I_2**

goto(Io, F) = closure(T → F .) = {T → F.} = **I_3**

goto(Io, +) = ϕ [There is no production in the form NT → .+ in Io]

goto(Io, *) = ϕ [There is no production in the form NT → .* in Io]

goto(Io, ( ) = closure(F → (. E)) = {F → (.E), E → . E + T, E → .T, T → .T * F, T → .F, F → .(E)
F → .id} = **I_4**

goto(Io,)) = ϕ

goto(Io, id) = closure(F → id .) = {F → id.} = **I_5**

Example

I1={ E′ → E., E → E. + T}

State I1 with all grammar symbols as Input.

goto(I1, E) = ϕ

goto(I1, T) = ϕ

goto(I1, F) = ϕ

goto(I1, +) = closure(E → E + .T) = {E → E + .T, T → .T * F, T → .F, F → .(E), F → .id} = **I6**

goto(I1, *) = ϕ

goto(I1, ( ) = ϕ

goto(I1,)) = ϕ

goto(I1, id) = ϕ

Example

- **I_2**={ E → T., T → T. * F}

State I2 with all grammar symbols as Input.

goto(I2, E) = ϕ

goto(I2, T) = ϕ

goto(I2, F) = ϕ

goto(I2, +) = ϕ

goto(I2, *) = closure(T → T *.F) = { T → T * .F, F → .(E), F → .id} = **I_7**

goto(I2, ( ) = ϕ

goto(I2,)) = ϕ

goto(I2, id) = ϕ

Example

- I_3={T → F.}

State I_3 with all grammar symbols as Input.

goto(I_3, E) = ϕ
goto(I_3, T) = ϕ
goto(I_3, F) = ϕ
goto(I_3, +) = ϕ
goto(I_3, *) = ϕ
goto(I_3, ( ) = ϕ
goto(I_3,)) = ϕ
goto(I_3, id) = ϕ

I_3 = {T → F.} i.e. {T → id.}. So I_3 is the final state.

Example

$\mathbf{I_4}$={F → (.E), E → . E + T, E → .T, T → .T * F, T → .F, F → .(E), F → .id}

State I_4 with all grammar symbols as Input.

goto(I_4, E) = closure(F → (E.), E → E. + T) = {F → (E.), E → E. + T } = **I8**

goto(I_4, T) = closure(E → T., T → T. * F) = {E → T., T → T. * F} = I2

goto(I_4, F) = closure(T → F.) = {T → F.} = I_3

goto(I_4, +) = ϕ

goto(I_4, *) = ϕ

goto(I_4, ( ) =

closure(F → (.E)) = {F → (.E), E → .E + T, E → .T, T → .T * F, T → .F, F → .(E) F → .id} = I_4

goto(I_4,)) = ϕ

goto(I_4, id) = closure(F → id.) = {F → id.} = I_5

Example

$I_5 = \{F \rightarrow id.\}$

State I_5 with all grammar symbols as Input.

goto(I5, E) = ϕ
goto(I5, T) = ϕ
goto(I5, F) = ϕ
goto(I5, +) = ϕ
goto(I5, *) = ϕ
goto(I5, ( ) = ϕ
goto(I5,)) = ϕ
goto(I5, id) = ϕ

I5 is the final state.

Example

I6 ={E → E + .T, T → .T * F, T → .F, F → .(E), F → .id}

State I6 with all grammar symbols as Input.

goto(I6, E) = ϕ

goto(I6, T) = closure{E → E + T., T → T. * F} =I_9

goto(I6, F) = closure(T → F.) =I_3

goto(I6, +) = ϕ

goto(I6, *) = ϕ

goto(I6, ( ) = closure{F → (.E)}

={F → (.E), E → . E + T, E → .T, T → .T * F, T → .F, F → .(E), F → .id}=I_4

goto(I6,)) = ϕ

goto(I6, id) = {F → id.}= I_5

Example

- $\mathbf{I_7}$ ={T → T * .F, F → .(E), F → .id}

State I_7 with all grammar symbols as Input.

goto(I_7, E) = ϕ

goto(I_7, T) = ϕ

goto(I_7, F) = closure(T → T *F.) = {T → T * F.} = $\mathbf{I_{10}}$

goto(I_7, +) = ϕ

goto(I_7, *) = ϕ

goto(I_7, ( ) = closure (F → (.E))

= {F→ (.E), E → .E + T, E → .T, T → .T * F, T → .F, F → .(E), F→ .id} = I_4

goto(I_7,)) = ϕ

goto(I_7, id) = closure(F → id.) = {F → id.} = I_5

Example

$I_8 = \{F \rightarrow (E.),\ E \rightarrow E.\ +\ T\}$

State I_8 with all grammar symbols as Input.

goto(I8, E) = ϕ

goto(I8, T) = ϕ

goto(I8, F) = ϕ

goto(I8, +) = closure(E → E +. T) = {E → E + .T, T → . T * F, T → .F, F → .(E), F → .id } = I_6

goto(I8, *) = ϕ

goto(I8, ( ) = ϕ

goto(I8,)) = closure(F→ (E).) = {F → (E). } = **I_{11}**

Example

- $I_9 = \{E \rightarrow E + T., T \rightarrow T. * F\}$

State I_9 with all grammar symbols as Input.

goto(I8, id) = ϕ

goto(I9, E) = ϕ

goto(I9, T) = ϕ

goto(I9, F) = ϕ

goto(I9, +) = ϕ

goto(I9, *) = closure($T \rightarrow T * .F$) = $\{T \rightarrow T * .F, F \rightarrow .(E), F \rightarrow .id\} = I_7$

goto(I9, ( ) = ϕ

goto(I9,)) = ϕ

goto(I9, id) = ϕ

I_9 is the final state. As it contains $E \rightarrow E + T.$ [Parse end]

I_{10} is a final state as it contains $T \rightarrow T * F.$

I_{11} is a final state.

Example

The final FA is

State	Next State, Input							
	id	+	*	(	)	E	T	F
0	5			4		1	2	3
1		6						
2			7					
3								
4	5			4		8	2	3
5								
6	5			4			9	3
7	5			4				10
8		6			11			
9			7					
10								
11								

Example

III. Make the Shift and Goto part ($ is added)

	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			g1	g2	g3
1		S6							
2			S7						
3									
4	S5			S4			g8	g2	g3
5									
6	S5			S4				g9	g3
7	S5			S4					g10
8		S6			S11				
9			S7						
10									

Example

IV Construction of Reduce part

Equations are numbered

0	E′ → E
1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → (E)
6	F→ id

Follow of the Non-Terminals

FOLLOW(E′) = {$}

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

FOLLOW(T) = {+,), *, $}

FOLLOW(F) = {+,), *, $}

I1 contains a production E′ → E., and FOLLOW(E′) = {$}. Thus Action [I1, $] = Accept. As E′ is start symbol

I2 contains a production E → T. (production number 2), and FOLLOW(E) = {+,), $}.

Action[I2, +] = r2, Action[I2,)] = r2, Action[I2, $] = r2

For I3: Action[I3, +] = r4, Action[I3,)] = r4, Action[I3, *] = r4, Action[I3, $] = r4,

By this process the final table is

Example

	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			g1	g2	g3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			g8	g2	g3
5		r6	r6		r6	r6			
6	S5			S4				g9	g3
7	S5			S4					g10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Assignment

1.Construct SLR parsing Table for the grammar

E →T + E

E →T

T →int * T

T →int

T →(E)

	int	*	+	(	)	$	E	T
1	s5			s4			s2	s3
2						acc		
3			s6		r2	r2		
4	s5			s4			s7	s3
5		s8	r4		r4	r4		
6	s5			s4			s9	s3
7					s10			
8	s5			s4				s11
9					r1	r1		
10			r5		r5	r5		
11			r3		r3	r3		

Assignment

2. Construct SLR parsing table for the following grammar

S →AA/bc

A →baA/c

	Action				goto	
	a	b	c	$	S	A
0		s4	s5		g2	g3
1				acc		
2		s7	s5			g6
3	s8		s9			
4	r4	r4	r4	r4		
5	r1	r1	r1	r1		
6	s8					
7		s7	s5			g10
8	r2	r2	r2	r2		
9	r3	r3	r3	r3		

S → AS |b

A →SA|a

S →Cb

C →bC | d

Source: D.Galles p: 104

Problem in SLR

Every SLR(1) grammar is unambiguous. But there are many unambiguous grammars that are not SLR(1).

Example:

Construct SLR parsing table for

S → L=R

S →R

L →*R

L→id

R →L

Sol. The augmented grammar is

S/ →S

S → L=R

S →R

L →*R

L→id

R →L

S/ →.S is in I. The closure(I) is the set as follow.

{ S/ →.S

S → .L=R

S →.R

L →.*R

L→.id

R →.L }

Mark it I_0.

- State Io with all grammar symbols as Input.

goto(I_o, S) = closure(S′ → S.) =(S′ → S.) Mark it as $\mathbf{I_1}$

goto(I_o, L) = closure(S → L.=R, R →L.) = (S → L.=R, R →L.) Mark it as $\mathbf{I_2}$

goto(I_o, R) = closure(S →R.) ={S →R.} Mark it as $\mathbf{I_3}$

goto(I_o, =) =ϕ

goto(I_o, *) =closure(L → *.R) = {L → *.R, R →.L, L →.*R, L→.id } Mark it as $\mathbf{I_4}$

goto(I_o, id) =closure(L→id.) ={L→id.} Mark it as $\mathbf{I_5}$

No transition from I_1 is possible.

goto(I_2, L) = closure(R →L.) = {R →L. } Mark it as $\mathbf{I_6}$

goto(I_2, =) = closure(S → L=.R) = {S → L=.R, L →.*R,L→.id,R →.L } Mark it as $\mathbf{I_7}$

goto(I_4, R) =closure(L → *R.) = {L → *R.} Mark it as I_8

goto(I_4, L)= {R →L. } it is $\mathbf{I_6}$

goto(I_4, id)= closure{L→id.} ={L→id.} It is $\mathbf{I_5}$

goto(I_4, *) =closure(L →*.R) = {L → *.R, R →.L, L →.*R, L→.id } it is $\mathbf{I_4}$

goto(I_7, R) =closure(S → L=R.) = {S → L=R.} Mark it as I_9

goto(I_7, *) =closure(L →*.R) = {L → *.R, R →.L, L →.*R, L→.id } it is $\mathbf{I_4}$

goto(I_7, id) = $\mathbf{I_5}$

goto(I_7, L) = $\mathbf{I_6}$

- The FA is

State	Next State, Input					
	id	=	*	S	L	R
0	5		4	1	2	3
1						
2		7			6	
3						
4	5		4		6	
5						
6						
7	5		4		6	9
8						
9						

- Shift and goto construction

State	Shift				goto		
	id	=	*	$	S	L	R
0	S5		S4		g1	g2	g3
1							
2		S7				g6	
3							
4	S5		S4			g6	
5							
6							
7	S5		S4			g6	g9
8							
9							

Follow of the non-terminals

Follow(S/)={$}

Follow(S)=

Follow(L)=

Follow(R)={=} S → L=R →*R=R

I_1 contains S′ → S.

Follow of S/is {$ }thus [1,$] will contain acc.

I_2 contains R →L.

Follow of R is {=} thus [2,=] will be r5.

0 S/ →S

1 S → L=R

2 S →R

3 L →*R

4 L→id

5 R →L

State	Shift				goto		
	id	=	*	$	S	L	R
0	S5		S4		g1	g2	g3
1				acc			
2		S7/r5				g6	
3							
4	S5		S4			g6	
5							
6							
7	S5		S4			g6	g9
8							
9							

State 2 suffers from shift reduce conflict.

Canonical & LALR Parsing

- Canonical LR is the most powerful LR parsing technique.

Canonical LR parsing table construction

- The construction process is same as LR parsing table construction. But difference is in Item Grammar Formation.

Here item is a pair [*P*, δ], where

- *P* is a production $A \rightarrow \beta$ with a • (dot) at some position in the *RHS*.
- δ is a terminal or $

Such items are known as LR(l) item.

l is the length of δ, known as look-ahead of the item.

Construction of LR(1) items
Closure(I)
If I is the set of items of a grammar G/ then Closure of I is a set of item constructed by the following rules.

Closure(I) contains every item in I.
Repeat
For each item [A →α.Bβ, a] in I,
each production B →γ in G/ ,
each terminal b in FIRST(βa)
Add {B→ .γ,b} in I.
Until no more items can be added to I

Shift and Goto part construction:

Construct the closure for all grammar symbol.

Construct finite automata from it.

For state transition with terminal place Shift with Item number.

with non-terminal place goto with Item number.

Reduce part construction:

If [A →α. , a] is in I_i , A ≠ S/ , then set action[i, a] to reduce A →α [Place r(production number of it)]

If [S/ →S. , $] is in I_i , then set action[i, $] to accept.

Example

Construct LR(1) item set for the grammar

S/ →S

S →CC

C→cC/d

Constructing Closure:

S/ →.S is in I. Start with closure(S/ →.S, $)

- Match it with [A →α.Bβ, a]

  (S/ → .S , $)

  [A →α.Bβ, a]

  →α is ε, B is S, β is ε, a is $
- There is production S →CC in G/
- FIRST(βa) is {$}.

Thus [S →.CC,$] is added to I

[S →.CC,$]

- Here α is ε, B is C, β is C, a is $
- There are productions C→cC/d in the grammar.
- FIRST(C$) = FIRST(C) = {c,d}

Thus [C→.cC, c], [C→.cC, d], [C→.d, c], [C→.d, d] are added to I

]

No other items can be added. Mark it as I_0

So I_0 is

{S/ →.S, $
S →.CC,$
C→.cC, c/d
C→.d, c/d
}

Example (Cont..)

goto(I_0, S) = closure(S/ →S., $) = {S/ →S., $} mark it as I_1

goto(I_0, C) = closure(S →C.C,$) = {[S →C.C,$], [C→.cC, $], [C→.d, $]} mark it as I_2

goto(I_0, c) = closure(C→c.C, c/d) = {[C→c.C, c/d], [C→.cC, c/d], [C→.d, c/d]} Mark it as I_3

goto(I_0 ,d) = closure(C→d., c/d) = {C→d., c/d} Mark it as I_4

goto(I_2 ,C) = closure(S →CC.,$) = {S →CC.,$} Mark it as I_5

goto(I_2 ,c) =closure(C→c.C, $) = {[C→c.C, $],[C→.cC, $],[C→.d,$]} Mark it as I_6

goto(I_2 ,d) =closure(C→d., $) ={C→d., $} Mark it as I_7

goto(I_3 ,C) = closure(C→cC., c/d)= {C →cC., c/d} Mark it as I_8

goto(I_3 ,c) =closure(C→c.C, c/d) = {[C→c.C, c/d], [C→.cC, c/d], [C→.d, c/d]} Same as I_3

goto(I_3 ,d) = closure(C→d., c/d) = {C→d., c/d} Same as I_4

goto(I_6 ,C) = closure(C→cC., $) = {C→cC., $} Mark it as I_9

goto(I_6,c) = closure(C→c.C, $) ={[C→c.C, $], [C→.cC, $], [C→.d, $]} Same as I_6

goto(I_6,d) = closure(C→d., $) = {C→d., $} Same as I_7

Example (Cont..)

Reduce part construction:

I_4 {C→d., c/d} has {[C→d., c], [C→d., d]} contains production in the form {A →α. , a} where , A ≠ S/ .
action[4, c] = r3 action[4,d] = r3 [C→d is production number 3]

I_8 {C →cC., c/d} has {[C→cC. , c], [C→cC. , d] contains production in the form {A →α. , a} where , A ≠ S/ .
action[8,c] = r2 action[8,d] = r2 [C→cC is production number 2]

I_5 { S →CC.,$} has S →CC. contains production in the form {A →α. , a} where , A ≠ S/ .
action[5, $] =r1

I_7 {C→d., $} has C→d. , contains production in the form {A →α. , a} where , A ≠ S/ .
action[7,$] = r3

I_9 {C→cC., $} has C→cC. , contains production in the form {A →α. , a} where , A ≠ S/ .
action[2,$] = r2

I_1 {S/ →S., $} contains production [S/ →S. , $] Thus action[1, $] = acc

Example (Cont..)

State	Shift			goto	
	c	d	$	S	C
0	S3	S4		g1	g2
1			acc		
2	S6	S7			g5
3	S3	S4			g6
4	r3	r3			
5			r1		
6	S6	S7			g9
7			r3		
8	r2	r2			
9			r2		

Canonical Parsing Table

0 $S' \rightarrow S$
1 $S \rightarrow CC$
2 $C \rightarrow cC$
3 $C \rightarrow d$

Example

- S' → S $
  S → S ; A
  S → A
  A → E
  A → id := E
  E → E + id
  E → id

Terminals = { $, ;, id, := ,+ }

- Io = Closure([S' → . S, $]) =
- { [S' → . S, $],
  [S → . A, $],
  [S → . S ; A, $],
  [A → . E, $],
  [A → . id := E, $],
  [S → . A, ;],
  [S → . S ; A, ;],
  [E → . E + id, $],
  [E → . id, $],
  [A → . E, ;],
  [A → . id := E, ;],
  [E → . E + id, +],
  [E → . id, +],
  [E → . E + id, ;],
  [E → . id, ;] }

State	Action					Goto			
	id	;	+	:=	$	S	A	E	
0	S4					g1	g2	g3	
1		S5			acc				
2		r3			r3				
3		r4	S6		r4				
4		r7	r7	S7	r7				
5	S4						g8	g3	
6	S9								
7	S11							g10	
8		r2			r2				
9		r6	r6		r6				
10		r5	S6		r5				
11		r7	r7		r7				

- Construct cannonical LR(1) parsing table for

S→SS+/ SS*/ a

Show that the following grammar
S → Aa|bA c|dc|bda
A→ d
is LALR(1) but not SLR(1)

Show that the following grammar
S → Aa |bAc |Bc|bBa
A → d
B →d
is LR(l) but not LALR(l)

for brilliant students.....

Further reading

- LR(0) parsing
- Ambiguous grammar and LR parsing

LALR Parsing Table

- Lookahead-LR.
- The method is most used in practise because the table obtained by it are considerably smaller than canonical LR table.
- This is more practical because the table obtained by it is considerably smaller than canonical LR table.
- Common syntactic construct of programming language can be expressed conveniently by LALR grammar.
- This is also true for SLR, but few construct as discussed before cannot be conveniently handled by SLR technique.
- In comparison of parser size both SLR and LALR for a grammar have the same number of states. [Example: For PASCAL it is several hundreds, For canonical LR it is several thousand.].

LALR parsing table construction

- With LALR the number of states of LR (1) parser is reduced by merging states differing only in their lookahead set.

Consider the previous grammar

S →CC

C→cC

C→d

The set of items are

- I_0 = {[S/ →.S, $], [S →.CC,$], [C→.cC, c/d], [C→.d, c/d]}
- I_1 ={[S/ →S., $]}
- I_2 ={ [S →C.C,$], [C→.cC, $], [C→.d, $]}
- I_3 = {[C→c.C, c/d], [C→.cC, c/d], [C→.d, c/d]}
- I_4 ={[C→d., c/d]}
- I_5 ={[S →CC.,$]}
- I_6 ={[C→c.C, $],[C→.cC, $],[C→.d,$]}
- I_7 ={[C→d., $]}
- I_8 ={[C →cC., c/d]}
- I_9 ={C→cC., $}

LALR parsing table construction

In the previous item sets (I_3 I_6), (I_4 I_7) and (I_8 I_9) can be merged as they differ only in their lookahead sets.

Merging the states the new states are constructed as.

I_{36} = {[C→c.C, c/d/$], [C→.cC, c/d/$], [C→.d, c/d/$]}

I_{47} ={[C→d., c/d/$]}

I_{89} ={[S →CC., c/d/$]}

State	Shift			goto	
	c	d	$	S	C
0	S36	S47		g1	g2
1			acc		
2	S36	S47			g5
36	S36	S47			g8
47	r3	r3			
5			r1		
8	r2	r2	r2		

Assignment

- Construct the LALR(1) parsing table for the following grammar.

S → BC
S → b
B → bB
B → a
C → ϵ
C → cC

	Action				goto		
	a	b	c	$	S	B	C
0	S5	S3			g2	g4	
1				acc			
2	S5	S7		r2		g6	
3			S9	r5			g8
4		r4	r4				
5		r3	r3				
6	S5	S7				g6	
7				r1			
8			S9				g10
9				r6			

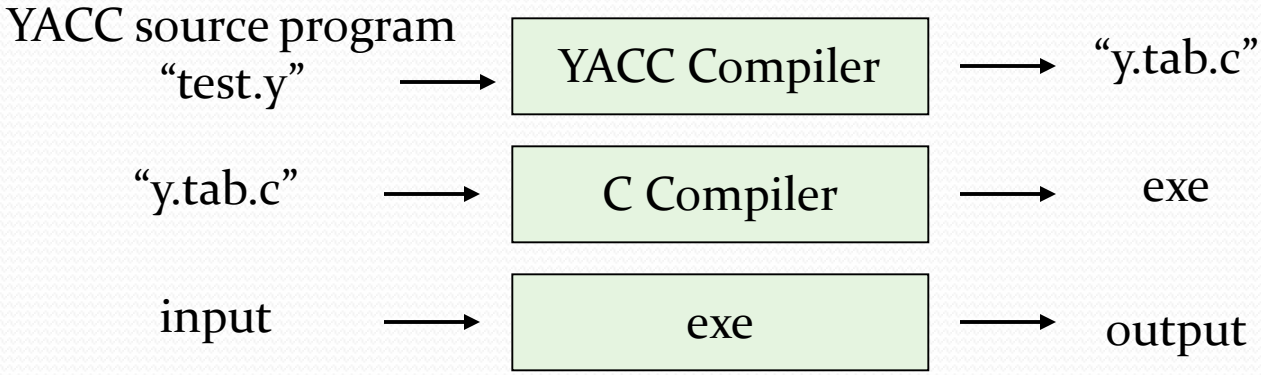
YACC

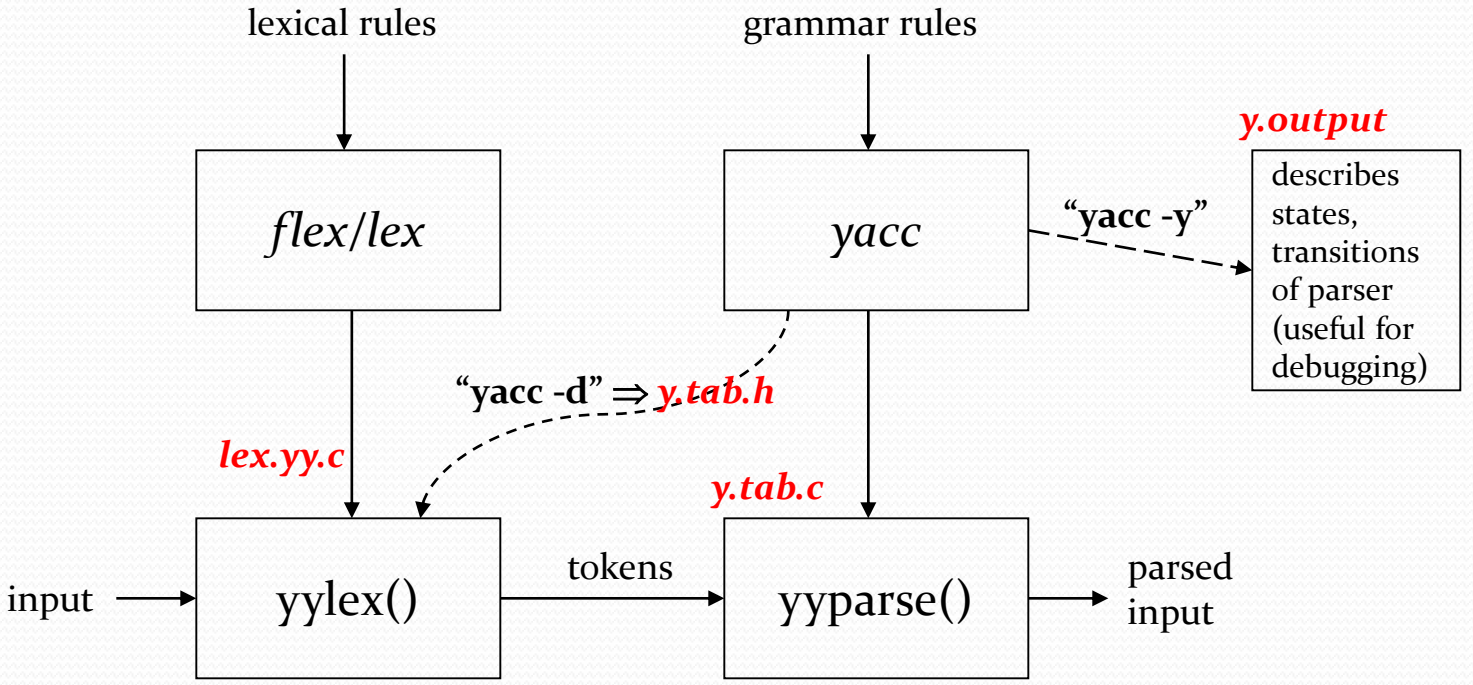
- Stands for **Y**et **A**nother **C**ompiler **C**ompiler.
- developed by **Stephen C. Johnson** at AT&T for the Unix operating system(1970).
- It generates an LALR(1) compiler based on a grammar written in a notation similar to Backus-Naur Form.
- Work with lex. YACC calls yylex() to get the next token.
- Invokes yylex() automatically.
- Generate *y.tab.h* file through the -d option.
- The lex input file must contains y.tab.h
- For each token that lex recognized, a number is returned (from *yylex()* function.)

YACC generates

- Tables – according to the grammar rules.
- Driver routines – in C programming language.
- y.output – a report file.

How does YACC works?





YACC file format

YACC file format:

```
declarations          /* specify tokens, and non-terminals */
%%
translation rules     /* specify grammar here */
%%
supporting C-routines
```

- Extension of YACC file is .y.

- Command "yacc yaccfile" produces y.tab.c, which contains a routine yyparse().
  - yyparse() calls yylex() to get tokens.

- yyparse() returns o if the program is grammatically correct, non-zero otherwise.

YACC file format

Declaration section consists of

C code:

- Any code between %{and%} is copied to the C file.
- typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

definitions:

The definitions section of a lex file was concerned with characters; in yacc this is tokens. These token definitions are written to a .h file when yacc compiles this file.

- Token are declared using '**%token'.** (To specify token AAA BBB %token AAA BBB)
- Start symbol is declared as **%start expr** (expr is the start symbol.)

[The first non-terminal specified in the grammar specification section.

To overwrite it with %start declaraction. %start non-terminal]

- Single-character tokens don't have to be declared
- Any name not declared as a token is assumed to be a non-terminal.

associatively rules: These handle associatively and priority of operators.

- '%right' Declare a terminal symbol (token type name) that is right-associative
- '%left' Declare a terminal symbol (token type name) that is left-associative

YACC file format

Example of **Declaration section** of a YACC file:

```
%{
#include <stdio.h>
int base;
int yyerror(char *msg);
%}

%start list
%token  NUM  ID
%left   '|'
%left  '&'
%left   '+'  '-'
%left    '*'    '/'   '%'          /*Operators declared lower have higher precedence*/
%left    UMINUS
```

YACC file format

- **Rule section: A** colon is used between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule:
- Unlike lex, yacc pays no attention to line boundaries in the rules section, *that is a lot of whitespace makes grammars easier to read*.

Let the grammar is

expr → expr + term | term;

term → term * factor | factor;

factor → (expr) | ID | NUM;

%%

The rule in .y file are written as

```
expr   :    expr    '+'    term
       |    term
       ;
term   :    term    '*'    factor
       |    factor
       ;
factor :    '(' expr ')'
       |    ID
       |    NUM
%%
```

YACC file format

The position of rules:

- Whenever the parser reduces a rule, it executes user C code associated with the rule, known as the rule's *action*.
- *The action appears in braces after the* end of the rule, before the semicolon or vertical bar.
- The action code can refer to the values of the right-hand side symbols as $1, $2, . . . , and can set the value of the left-hand side by setting $$.

```
expr   : expr '+' term   { $$ = $1 + $3; }
       | term            { $$ = $1; }
       ;
term : term '*' factor { $$ = $1 * $3; }
       | factor          { $$ = $1; }
       ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

•In the expression building rules, 'expr' is the first attribute at the right side and 'term' is the third attribute at the right side.

•The operator's value would be *$2, although in* grammar the operators do not have interchanging values.

•The action on the last two rule are not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value $1 to $$.

A complete YACC file

This is a YACC file to implement simple calculator

```
%{
#include<stdio.h>
#include<stdlib.h>
extern int yylex();
int yyerror(char *msg);
%}
%union {
   float f;}

%token <f> NUM
%type <f> E T F

%%
S : E          {printf("%f\n",$1);}
  ;
E : E '+' T    {$$ = $1 + $3;}
  | E '-' T    {$$ = $1 - $3;}
  | T          {$$ = $1;}
  ;
T : T '*' F    {$$ = $1 * $3;}
  | T '/' F    {$$ = $1 / $3;}
  | F          {$$ = $1;}
  ;
F : '(' E ')'   {$$ = $2;}
  | '-' F       {$$ = -$2;}
  | NUM         {$$ = $1;}
  ;

%%

int yyerror(char *msg)
{
   printf("Invalid Expression");
   exit(0);
}

int main()
{
  yyparse();
  return 0;
}
```

let the name is “calc.y”

A concrete YACC file

```
%{
#include<stdio.h>
#include<stdlib.h>
#include "y.tab.h"   //generated via yacc -d
%}

%%

[0-9]+(\.[0-9]+)?([eE][0-9]+)?  {yylval.f = atoi(yytext); return NUM;}
[-+()*/]                    {return yytext[0];}
[ \t\f\v\n]                  { ; }

%%


int yywrap()
{
   return -1;
}
```

A lex file is written to exatract the tokens

Let the name of the lex file is calc.l

Compile and Run

compile yacc file as
$ yacc calc.y

It will produce a file y.tab.c.

rather compile it as
$ yacc -d calc.y

it will produce y.tab.c and y.tab.h. The .h file is included in lex file.

compile the lex file as
$ lex calc.l
It generates lex.yy.c which generates tokens from the given input.

To run it finally
$ cc lex.yy.c y.tab.c which generates a.out.

Assignment

- Write a YACC program evaluate an arithmetic expression involving operating +, -,* and /. Please take care for division by ‘o’ error.
- Write a YACC program which will determine whether a string is accepted by the following grammar or not

  S → aSb|ab.
- Write a yacc program to recognize nested if statements and display the number of levels of nested if.

References

- Aho A, Sethi R, Ullman J. D., “Compilers Principle, Techniques, and Tools ” Pearson Education.
- Chattopadhyay Santanu, “Compiler Design”, PHI
- Gales David, “Modern Compiler Design” Pearson Education.
- Niemann Thomas, “A Guide To Lex & Yacc” O’Reilly & Associates
- Johnson Maggie , “SLR and LR(1) Parsing”, http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf
- Wayne Cochran “Introduction to Yacc” video lectures available in https://www.youtube.com.