- **Syntax Analysis: Introduction,**
- **Role Of Parsers, Context Free Grammars,**
- **Writing a grammar,**
- **Top Down Parsers,**
- **Bottom-Up Parsers,**
- **Operator-Precedence Parsing**

The role of parser

Uses of grammars

E -> E + T | T

T -> T * F | F

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

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

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

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Logical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct progrms

Context free grammars

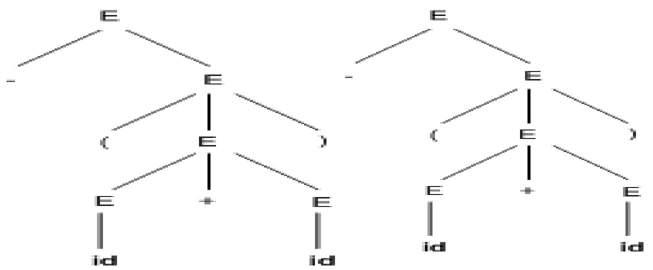
- Terminals
- Nonterminals
- Start symbol
- Productions

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

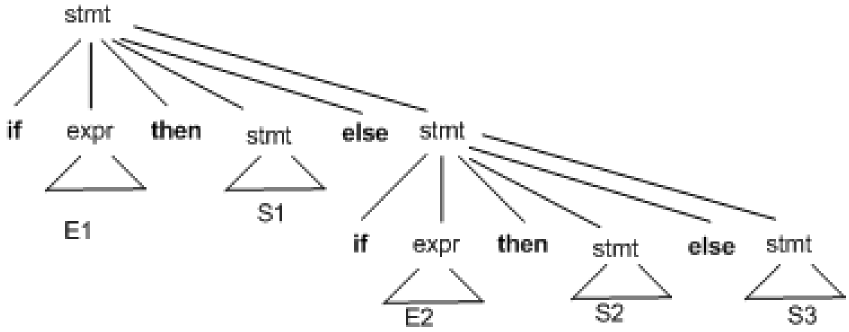
Parse trees

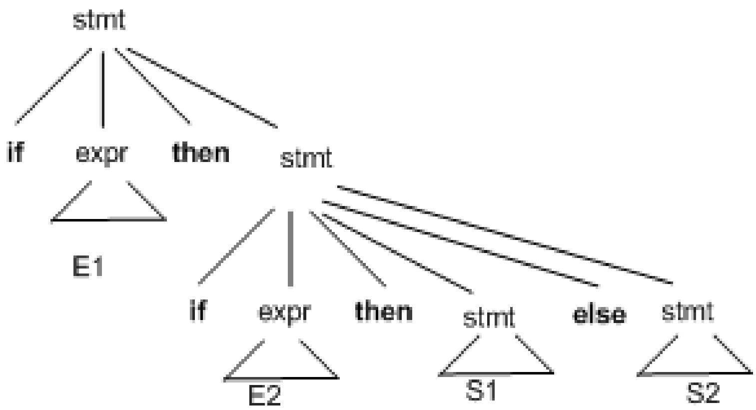
- **-(id+id)**
    - **E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)**

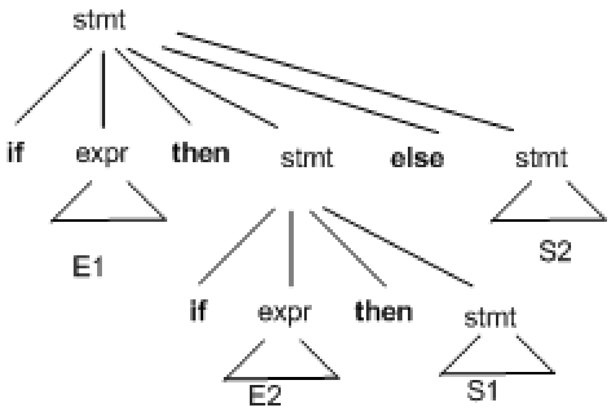


Elimination of ambiguity

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







Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation A=> Aα
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:

        - A -> A α|β
    - We may replace it with
        - A -> β A’
        - A’ -> α A’ | ε

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
    - Stmt -> if expr then stmt else stmt
    -       | if expr then stmt
- On seeing input if it is not clear for the parser which production to use
- We can easily perform left factoring:
    - If we have A->αβ1 | αβ2   then we replace it with
        - A -> αA’
        - A’ -> β1 | β2

➢ TOP DOWN PARSING

A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right

It can be also viewed as finding a leftmost derivation for an input string

Example: id+id*id

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

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

}

Recursive descent parsing

Consists of a set of procedures, one for each nonterminal

Execution begins with the procedure for start symbol

A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
            if (Xi is a nonterminal
                call procedure Xi();
            else if (Xi equals the current input symbol a)
                advance the input to the next symbol;
            else /* an error has occurred */
        }
}
```

Example

S->cAd

A->ab | a

Input: cad

First and Follow

- First() is set of terminals that begins strings derived from
- If α=>ε then is also in First(ε)
    - In predictive parsing when we have A-> α|β, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
    - If we have S => αAaβ for some αand βthen a is in Follow(A)

If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
    1. If X is a terminal then First(X) = {X}.
    2. If X is a nonterminal and X->Y1Y2...Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y1),...,First(Yi-1) that is Y1...Yi-1 => ε. if ε is in First(Yj) for j=1,...,k then add ε to First(X).
    3. If X-> ε is a production then add ε to First(X)
- Example!

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
    1. Place $ in Follow(S) where S is the start symbol

2. If there is a production A-> αBβ then everything in First(β) except ε is in Follow(B).
3. If there is a production A->B or a production A->αBβ where First(β) contains ε, then everything in Follow(A) is in Follow(B)

- Example!

LL(1) Grammars

Predictive parsers are those recursive descent parsers needing no backtracking

Grammars for which we can create predictive parsers are called LL(1)

The first L means scanning input from left to right

The second L means leftmost derivation

And 1 stands for using one input symbol for lookahead

A grammar G is LL(1) if and only if whenever A-> α|βare two distinct productions of G, the following conditions hold:

For no terminal a do αandβ both derive strings beginning with a

At most one of α or βcan derive empty string

If α=> ε then βdoes not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

For each production A->α in grammar do the following:

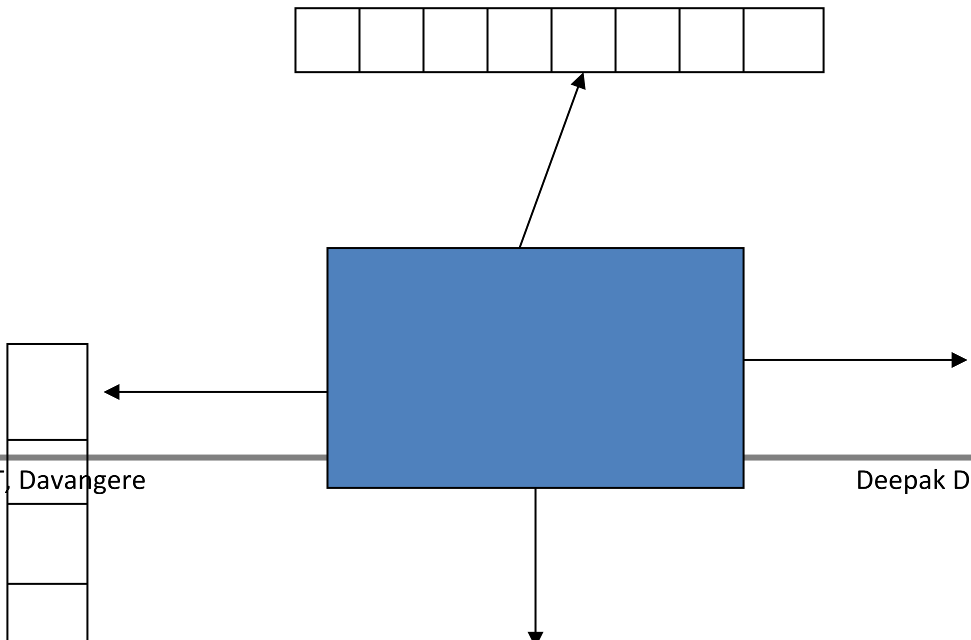
For each terminal a in First(α) add A-> in M[A,a]

If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well

If after performing the above, there is no production in M[A,a] then set M[A,a] to error .

Example

Non-recursive predicting parsing



Predictive parsing algorithm

```
Set ip point to the first symbol of w;
Set X to the top stack symbol;
While (X<>$) { /* stack is not empty */
        if (X is a) pop the stack and advance ip;
        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;
        }
        set X to the top stack symbol;
}
```

BOTTOMUP PARSING

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 applyA 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

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**

Shift reduce parsing (cont.)

Basic operations:

Shift,Reduce,Accept, Error Example: id*id

LR Parsing

The most prevalent type of bottom-up parsers

LR(k), mostly interested on parsers with k<=1

Why LR parsers?

Table driven

Can be constructed to recognize all programming language constructs

Most general non-backtracking shift-reduce parsing method

Can detect a syntactic error as soon as it is possible to do so

Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

States represent set of items

An LR(0) item of G is a production of G with the dot at some position of the body:

For A->XYZ we have following items

A->.XYZ

A->X.YZ

A->XY.Z

A->XYZ.

In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.

What about A->X.YZ?

Constructing canonical LR(0) item sets

Augmented grammar:

G with addition of a production: S'->S

Closure of item sets:

If I is a set of items, closure(I) is a set of items constructed from I by the following rules:

Add every item in I to closure(I)

If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).

Example: E'->E

E -> E + T | T

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

Closure algorithm

```
SetOfItems CLOSURE(I) {
        J=I;
        repeat
                for (each item A-> α.Bβ in J)
                        for (each prodcution B->γ of G)
                                if (B->.γ is not in J)
                                        add B->.γ to J;
        until no more items are added to J on one round;
        return J;
```

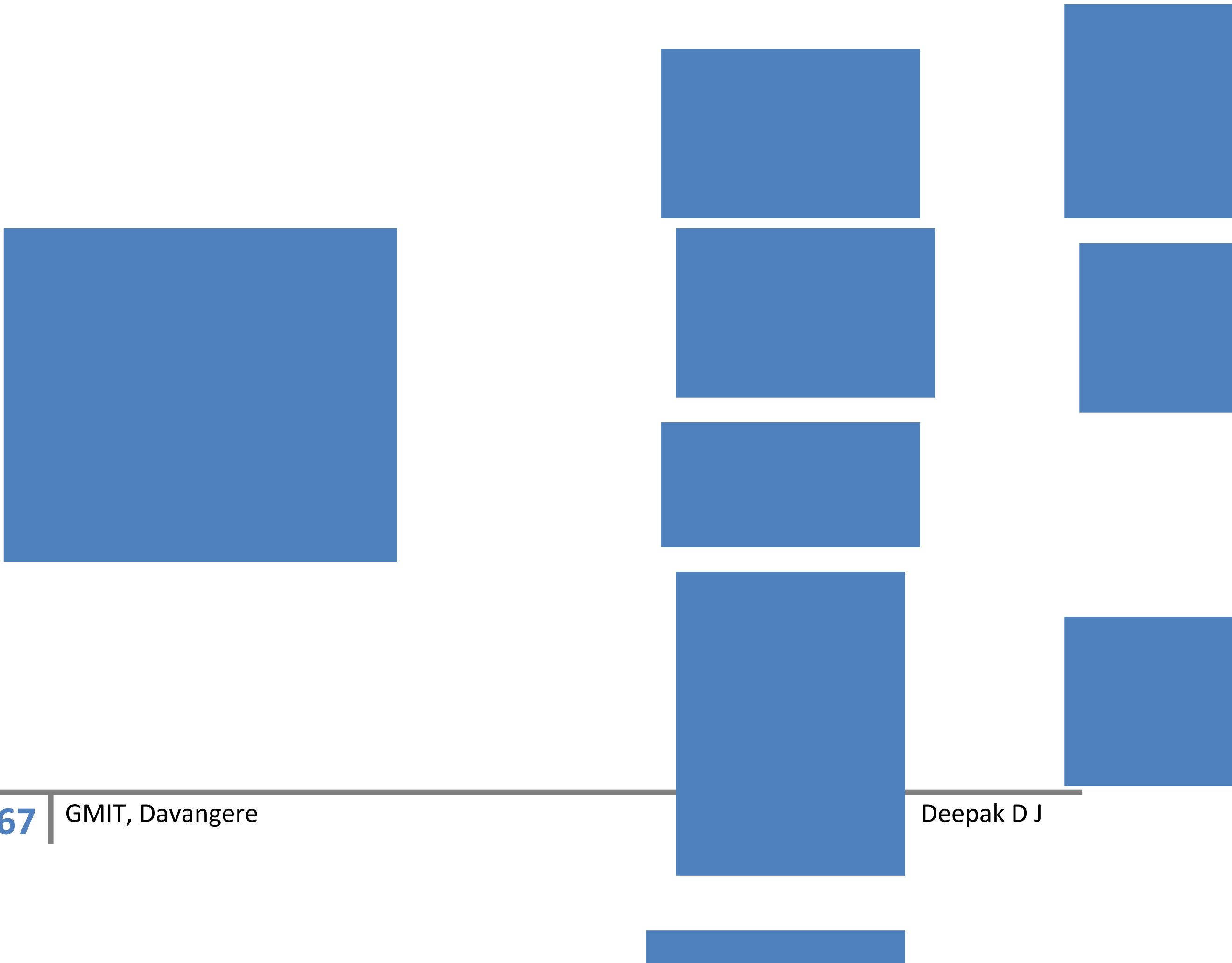
<u>GOTO Algorithm</u>

```
SetOfItems  GOTO(I,X) {
  J=empty;
        if (A-> α.X β is in I)
                add CLOSURE(A-> αX. β ) to J;
```

```
        return J;
}
```

Canonical LR(0) items

```
Void items(G') {
        C= CLOSURE({[S'->.S]});
        repeat
                for (each set of items I in C)
                  for (each grammar symbol X)
                    if (GOTO(I,X) is not empty and not in C)
                        add GOTO(I,X) to C;
        until no new set of items are added to C on a round;
}
```



Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F->
(3)	03	$F	*id$	Reduce by T->
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F->
(7)	02710	$T*F	$	Reduce by T->
(8)	02	$T	$	Reduce by E->
(9)	01	$E	$	accept

a1	…	ai	…	an	$

Sm
Sm-1
…
$

LR parsing algorithm

let a be the first symbol of w$;

while(1) { /*repeat forever */

ACTION	GOTO

```
        let s be the state on top of the stack;
        if (ACTION[s,a] = shift t) {
                push t onto the stack;
                let a be the next input symbol;
        } else if (ACTION[s,a] = reduce A->β) {
                pop |β| symbols of the stack;
                let state t now be on top of the stack;
                push GOTO[t,A] onto the stack;
                output the production A->β;
        } else if (ACTION[s,a]=accept) break; /* parsing is done */
        else call error-recovery routine;
}
```

STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R 4	R7		R4	R4			
4	S5			S4			8	2	3
5		R 6	R 6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Line	Stack	Symbols
(1)	0	
(2)	05	id
(3)	03	F
(4)	02	T
(5)	027	T*
(6)	0275	T*id
(7)	02710	T*F
(8)	02	T
(9)	01	E
(10)	016	E+
(11)	0165	E+id
(12)	0163	E+F

Constructing SLR parsing table

Method

Construct C={I0,I1, … , In}, the collection of LR(0) items for G’

State i is constructed from state Ii:

> If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
>
> If [A->α.] is in Ii, then set ACTION[i,a] to “reduce A->α” for all a in follow(A)
>
> If {S’->.S] is in Ii, then set ACTION[I,$] to “Accept”

If any conflicts appears then we say that the grammar is not SLR(1).

If GOTO(Ii,A) = Ij then GOTO[i,A]=j

All entries not defined by above rules are made “error”

The initial state of the parser is the one constructed from the set of items containing [S’->.S]