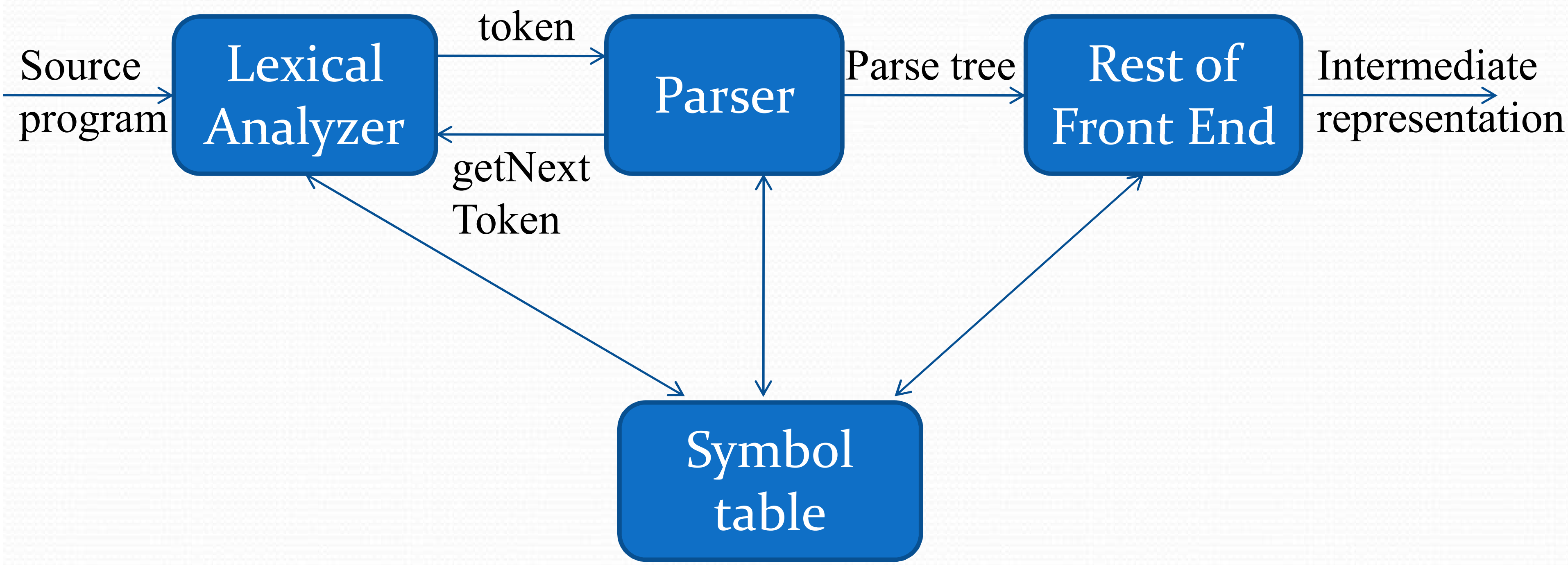
Chapter 4
Syntax Analysis

Sr. Nisha C D
Assistant Professor, Dept. of Computer Science
Little Flower College, Guruvayoor

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

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
  - Lexical 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

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

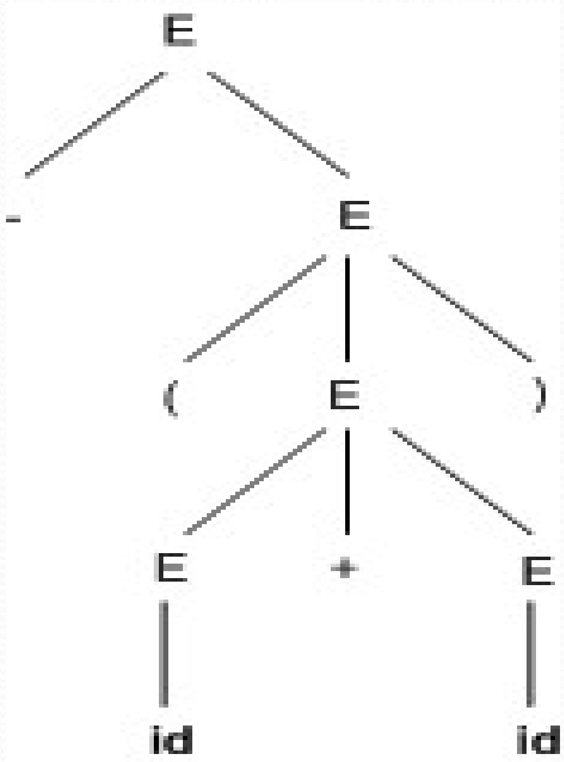
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

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**)

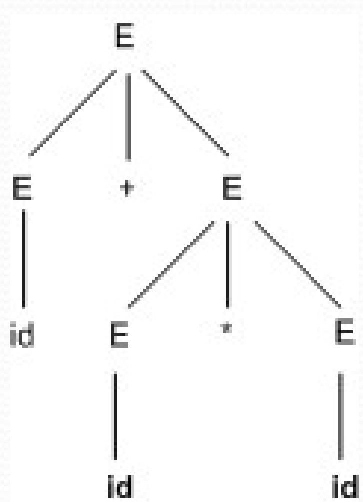
Parse trees

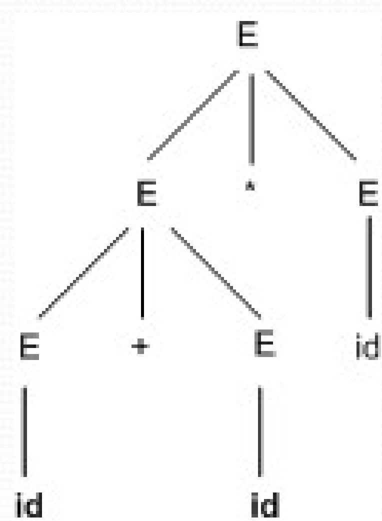
- **-(id+id)**
- E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)



Ambiguity

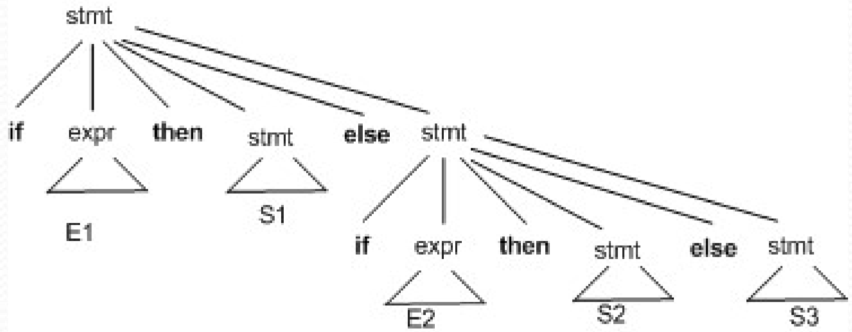
- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id

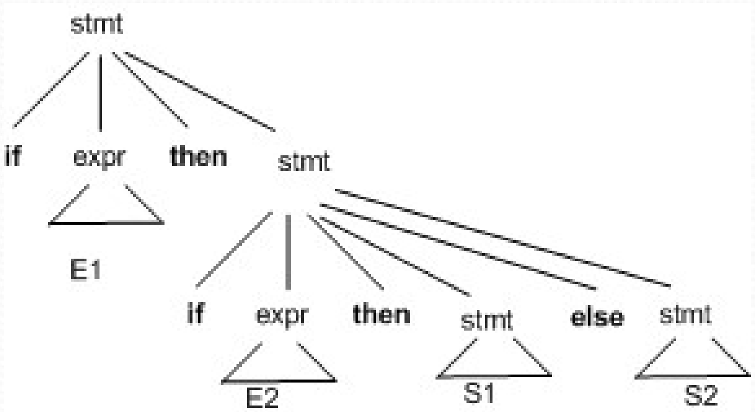


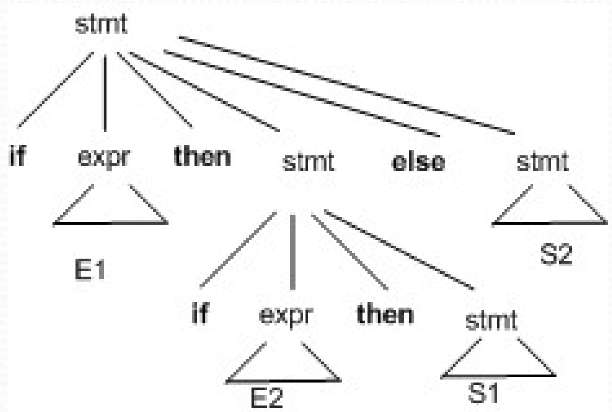


Elimination of ambiguity

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







Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched

stmt → matched_stmt
     | open_stmt

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

open_stmt → If expr **then** stmt
     | If expr **then** matched_stmt **else** open_stmt

Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{=}> A\ \alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \text{ -> } A\ \alpha \mid \beta$
  - We may replace it with
    - $A \text{ -> } \beta\ A'$
    - $A' \text{ -> } \alpha\ A' \mid \varepsilon$

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order A1,A2,…,An.
  - For (each i from 1 to n) {
    - For (each j from 1 to i-1) {
      - Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta_1 \gamma$ | $\delta_2 \gamma$ | … | $\delta_k \gamma$ where Aj-> δ_1 | δ_2 | … | δ_k are all current Aj productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

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-> $\alpha \beta 1 \mid \alpha \beta 2$ then we replace it with
    - A -> α A'
    - A' -> $\beta 1 \mid \beta 2$

Left factoring (cont.)

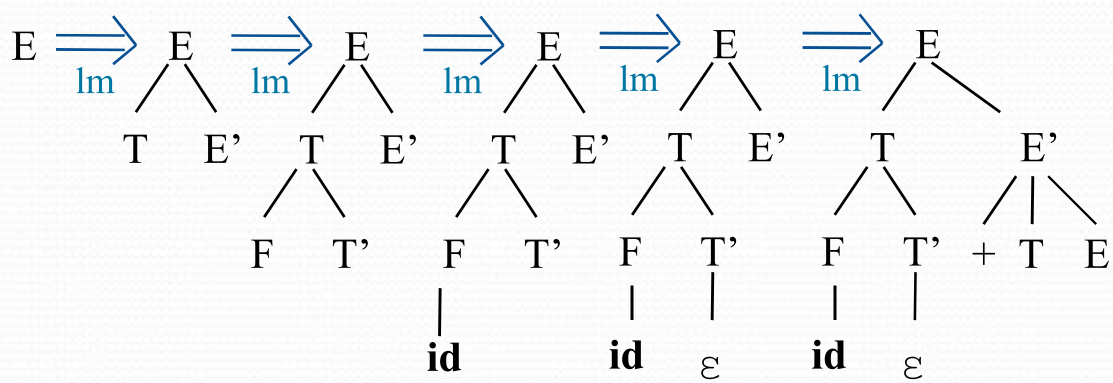
- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α <> ε, then replace all of A-productions A-> $\alpha\ \beta 1 \mid \alpha\ \beta 2 \mid \ldots \mid \alpha\ \beta n \mid \gamma$ by
    - A -> α A' | γ
    - A' -> $\beta 1 \mid \beta 2 \mid \ldots \mid \beta n$
- Example:
  - S -> I E t S | i E t S e S | a
  - E -> b

Top Down Parsing

Introduction

- 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' | ε
T -> FT'
T' -> *FT' | ε
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 */
        }
}
```

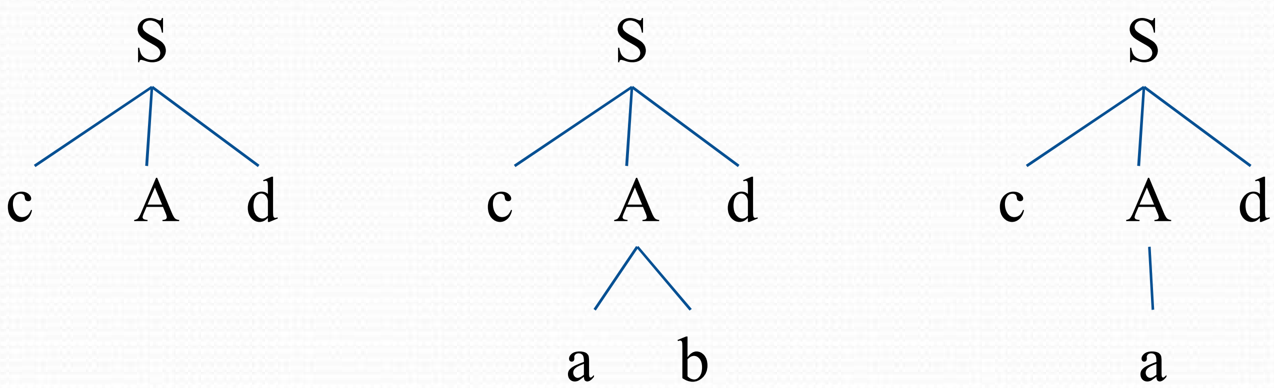
Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

Example

S->cAd
A->ab | a

Input: cad



First and Follow

- First() is set of terminals that begins strings derived from
- If $\alpha \overset{*}{=}> \varepsilon$ then is also in First(ε)
- In predictive parsing when we have A-> $\alpha \mid \beta$, 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 $\overset{*}{=}>$ α Aa β for some α and β then 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->$Y_1Y_2 \ldots Y_k$ is a production for some k>=1, then place a in First(X) if for some i a is in First(Y_i) and ε is in all of First(Y_1),…,First(Y_{i-1}) that is $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$. if ε is in First($Y_j$) 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-> $\alpha B \beta$ then everything in First(β) except ε is in Follow(B).
  3. If there is a production A->B or a production A-> $\alpha B \beta$ 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-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \overset{*}{=>} \varepsilon$ then β does 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:
  1. For each terminal a in First(α) add A-> in M[A,a]
  2. 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

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E'	{+,ε}	{), $}
T'	{*,ε}	{+,), $}

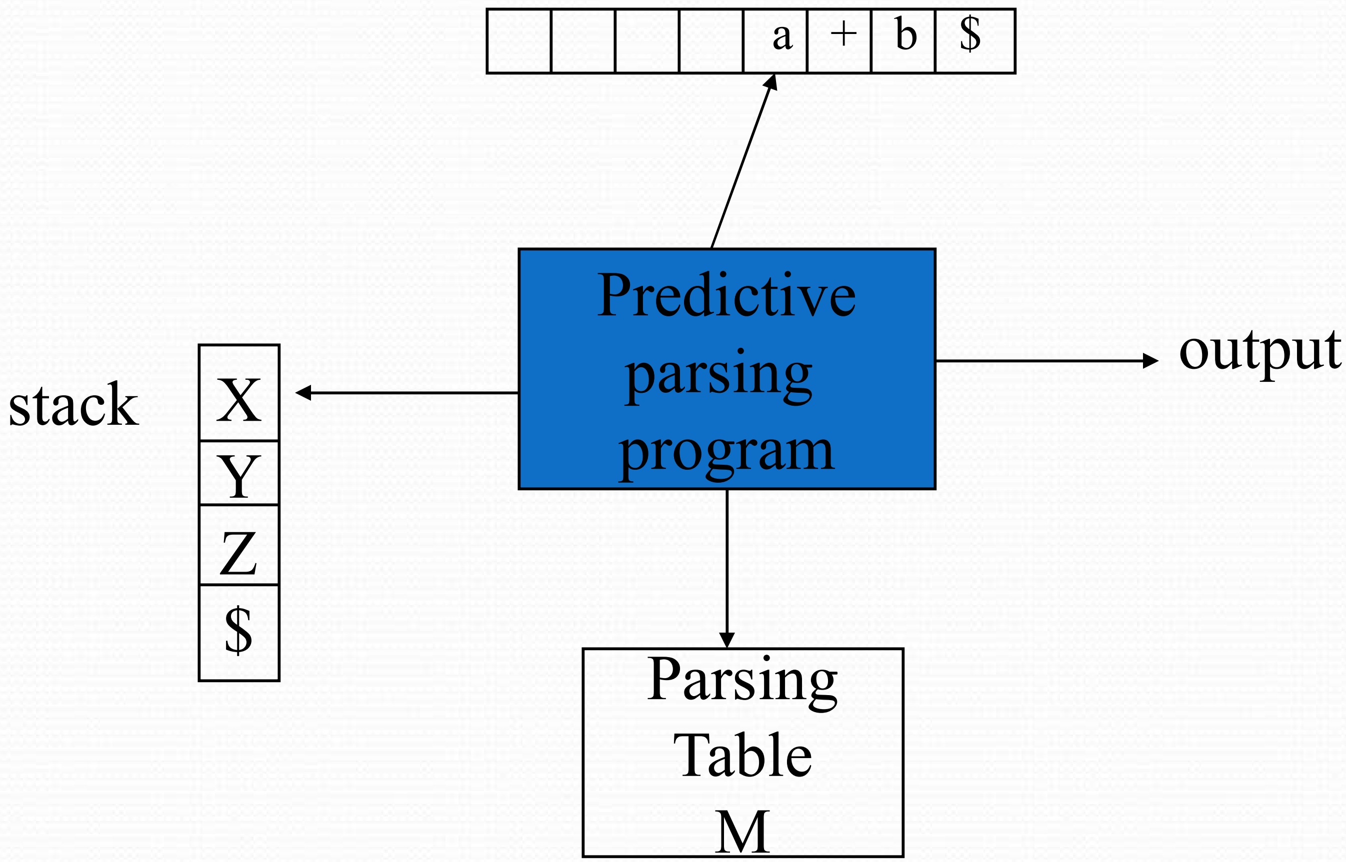
Non - terminal	Input Symbol id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | ε
E -> b

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

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;
}
```

Example

- id+id*id$

Matched	Stack	Input	Action
	E$	id+id*id$	

Error recovery in predictive parsing

- Panic mode
  - Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.
  - Add to the synchronization set of lower level construct the symbols that begin higher level constructs
  - Add symbols in First(A) to the synchronization set of nonterminal A
  - If a nonterminal can generate the empty string then the production deriving can be used as a default
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Example

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

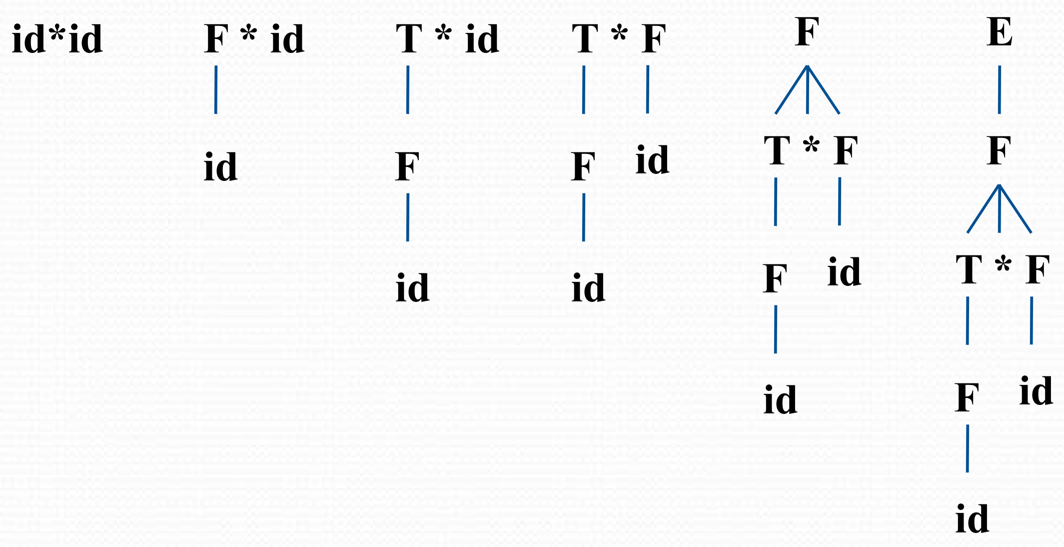
Stack	Input	Action
E$	)id*+id$	Error, Skip)
E$	id*+id$	id is in First(E)
TE'$	id*+id$	
FT'E'$	id*+id$	
idT'E'$	id*+id$	
T'E'$	*+id$	
*FT'E'$	*+id$	
FT'E'$	+id$	Error, M[F,+]=synch
T'E'$	+id$	F has been poped

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

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

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

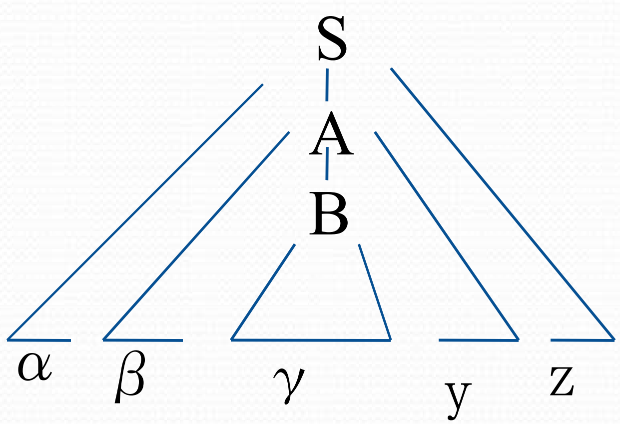
Stack	Input
$S	$

Shift reduce parsing (cont.)

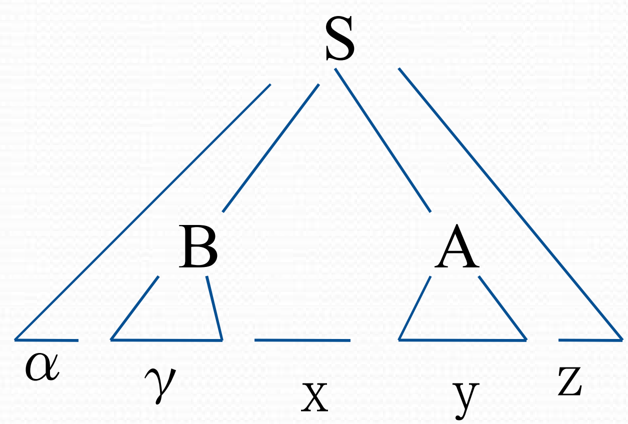
- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error
- Example: id*id

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



Stack	Input
$\$ \alpha \; \beta \; \gamma$	yz$
$\$ \alpha \; \beta B$	yz$
$\$ \alpha \; \beta By$	z$



Stack	Input
$\$ \alpha \; \gamma$	xyz$
$\$ \alpha Bxy$	z$

Conflicts during shit 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	Input
… if expr then stmt	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) …$

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(o) 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

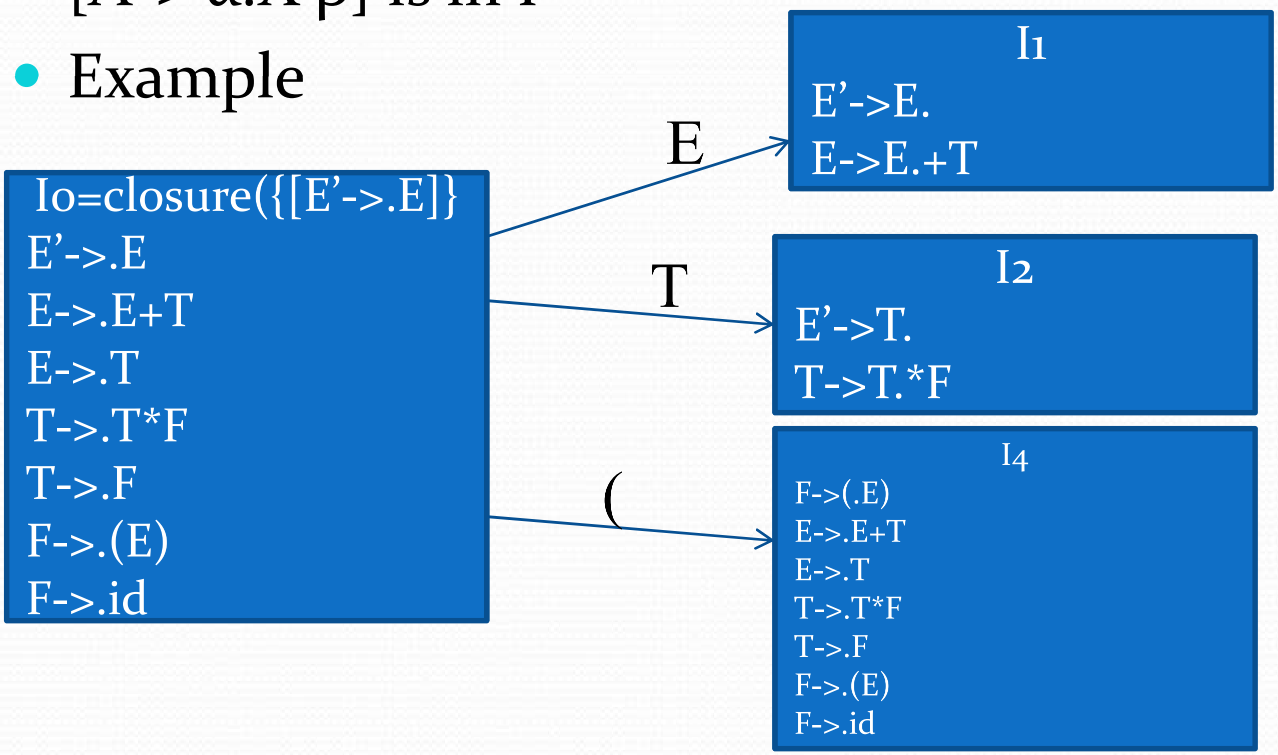
- Augmented grammar:
  - G with addition of a production: S’->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).
- Example:

E’->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

Io=closure({[E’->.E]}
E’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.X β] is in I
- Example



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;
```

GOTO algorithm

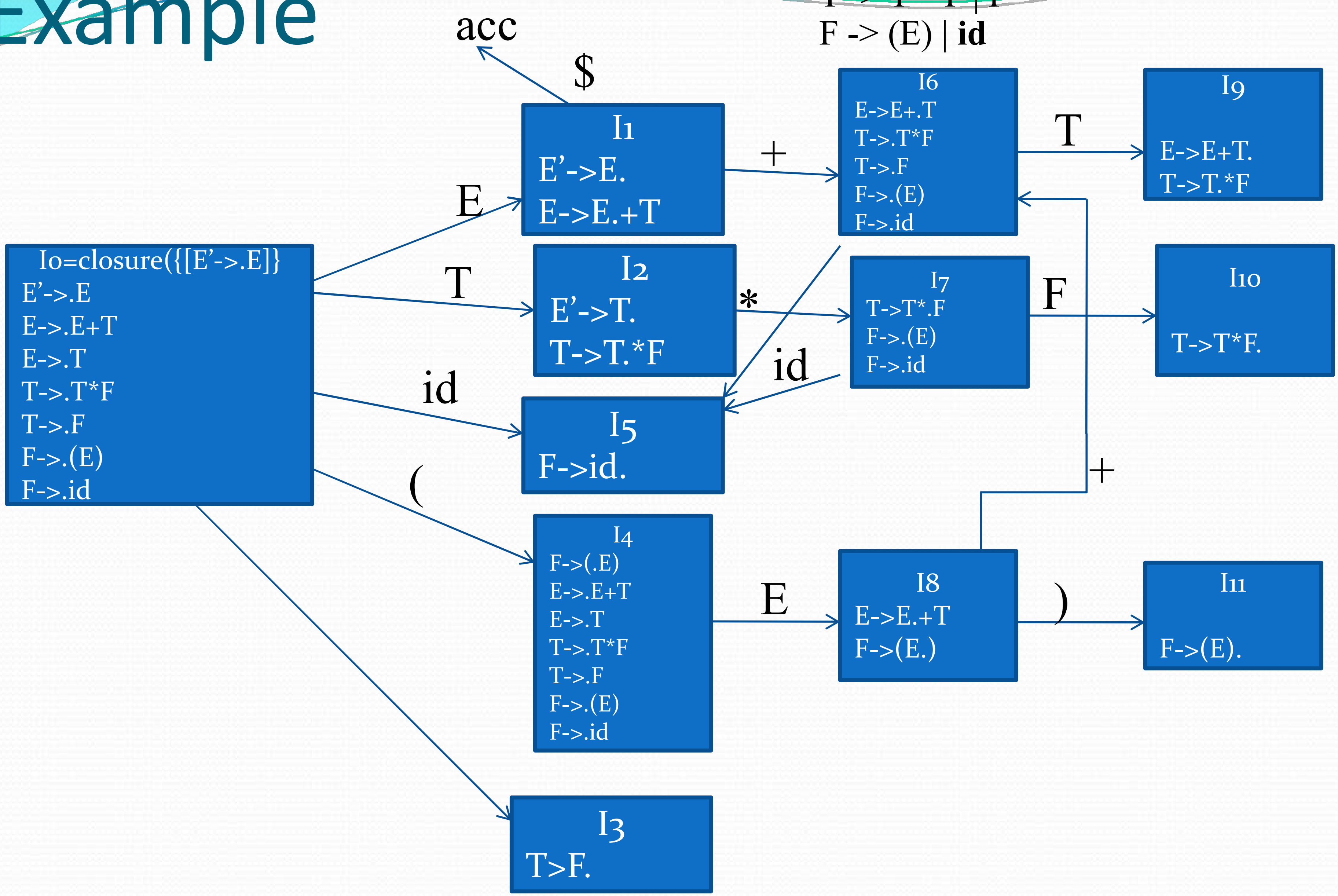
```
SetOfItems  GOTO(I,X) {
  J=empty;
  if (A-> α.X β is in I)
        add CLOSURE(A-> αX. β ) to J;
  return J;
}
```

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;
}
```

Example

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

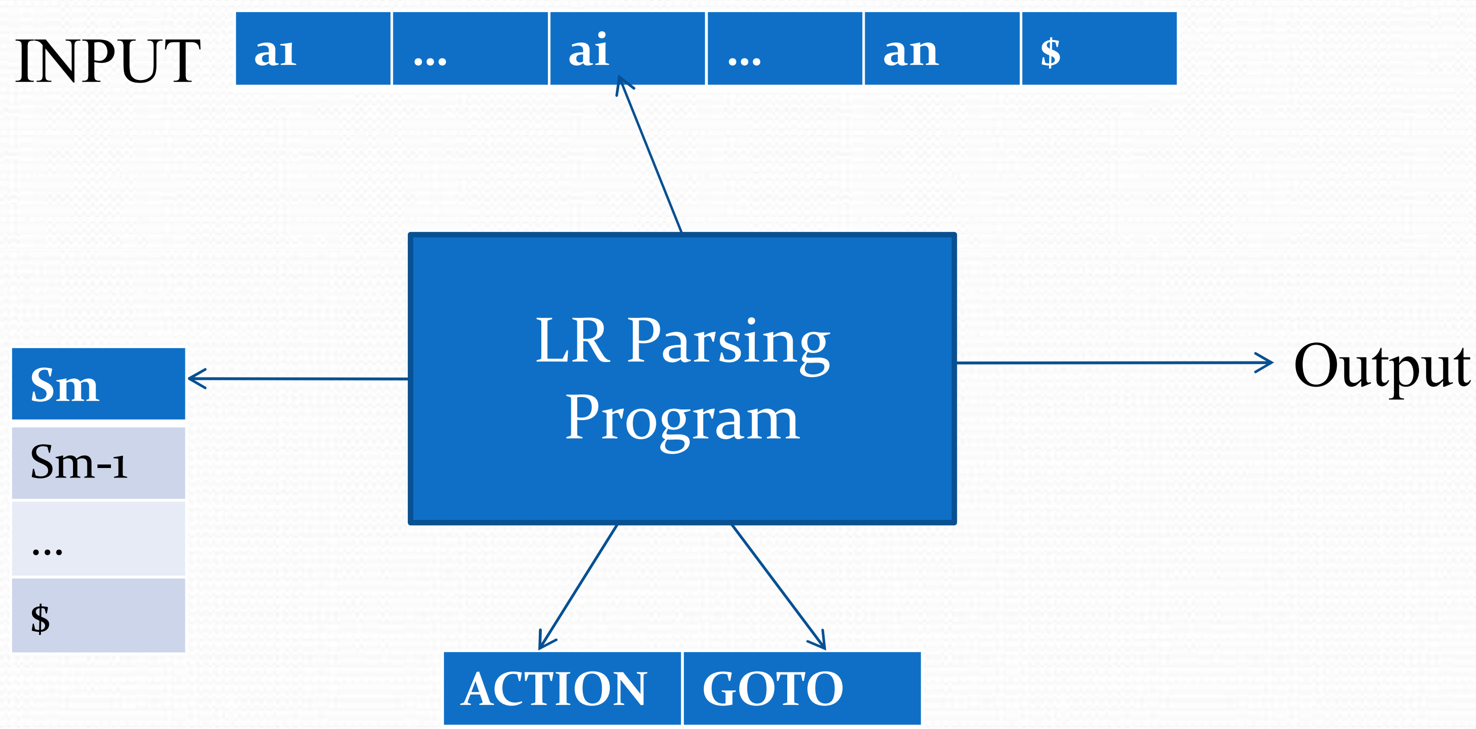


Use of LR(0) automaton

- Example: id*id

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

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;
while(1) { /*repeat forever */
    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;
}
```

Example

(0) E'->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F->**id**

id*id+id?

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	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id
(12)	0163	E+F	$	Reduce by T->F
(13)	0169	E+T`	$	Reduce by E->E+T
(14)	01	E	$	accept

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]

Example grammar which is not SLR(1)

S -> L=R | R
L -> *R | id
R -> L

I0
S'->.S
S -> .L=R
S->.R
L -> .*R |
L->.id
R ->. L

I1
S'->S.

I2
S ->L.=R
R ->L.

I3
S ->R.

I4
L->*.R
R->.L
L->.*R
L->.id

I5
L -> id.

I6
S->L=.R
R->.L
L->.*R
L->.id

I7
L -> *R.

I8
R -> L.

I9
S -> L=R.

	Action =
2	Shift 6 Reduce R->L

More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items

Canonical LR(1) items

- In LR(1) items each item is in the form: [A->α.β,a]
- An LR(1) item [A->α.β,a] is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where
  - Γ= δα
  - Either a is the first symbol of w, or w is ε and a is $
- Example:
  - S->BB
  - B->aB|b

$S \overset{*}{\Rightarrow} aaBab \underset{rm}{\Rightarrow} aaaBab$

Item [B->a.B,a] is valid for γ=aaa and w=ab

Constructing LR(1) sets of items

```
SetOfItems Closure(I) {
    repeat
        for (each item [A->α.Bβ,a] in I)
            for (each production B->γ in G')
                for (each terminal b in First(βa))
                    add [B->.γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems Goto(I,X) {
    initialize J to be the empty set;
    for (each item [A->α.Xβ,a] in I)
        add item [A->αX.β,a] to set J;
    return closure(J);
}

void items(G'){
    initialize C to 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 sets of items are added to C;
}
```

Example

S’->S

S->CC

C->cC

C->d

Canonical LR(1) parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(1) items for G’
  - State i is constructed from state Ii:
    - If [A->α.aβ, b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
    - If [A->α., a] is in Ii, then set ACTION[i,a] to “reduce A->α”
    - If {S’->.S,$] is in Ii, then set ACTION[I,$] to “Accept”
  - If any conflicts appears then we say that the grammar is not LR(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,$]

Example

S'->S

S->CC

C->cC

C->d

LALR Parsing Table

- For the previous example we had:

I4
C->d. , c/d

I47
C->d. , c/d/$

I7
C->d. , $

- State merges cant produce Shift-Reduce conflicts. Why?
- But it may produce reduce-reduce conflict

Example of RR conflict in state merging

S'->S

S -> aAd | bBd | aBe | bAe

A -> c

B -> c

An easy but space-consuming LALR table construction

- Method:
  1. Construct C={I0,I1,…,In} the collection of LR(1) items.
  2. For each core among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  3. Let C'={J0,J1,…,Jm} be the resulting sets. The parsing actions for state i, is constructed from Ji as before. If there is a conflict grammar is not LALR(1).
  4. If J is the union of one or more sets of LR(1) items, that is J = I1 UI2…IIk then the cores of Goto(I1,X), …, Goto(Ik,X) are the same and is a state like K, then we set Goto(J,X) =k.
- This method is not efficient, a more efficient one is discussed in the book

Compaction of LR parsing table

- Many rows of action tables are identical
  - Store those rows separately and have pointers to them from different states
  - Make lists of (terminal-symbol, action) for each state
  - Implement Goto table by having a link list for each nonterinal in the form (current state, next state)

Using ambiguous grammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						GO TO
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Acc	
2	S3		S2				6
3		R4	R4		R4	R4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5				
7		R1	S5		R1	R1	
8		R2	R2		R2	R2	
9		R3	R3		R3	R3	

I0: E'->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E'->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->.id

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

Readings

- Chapter 4 of the book