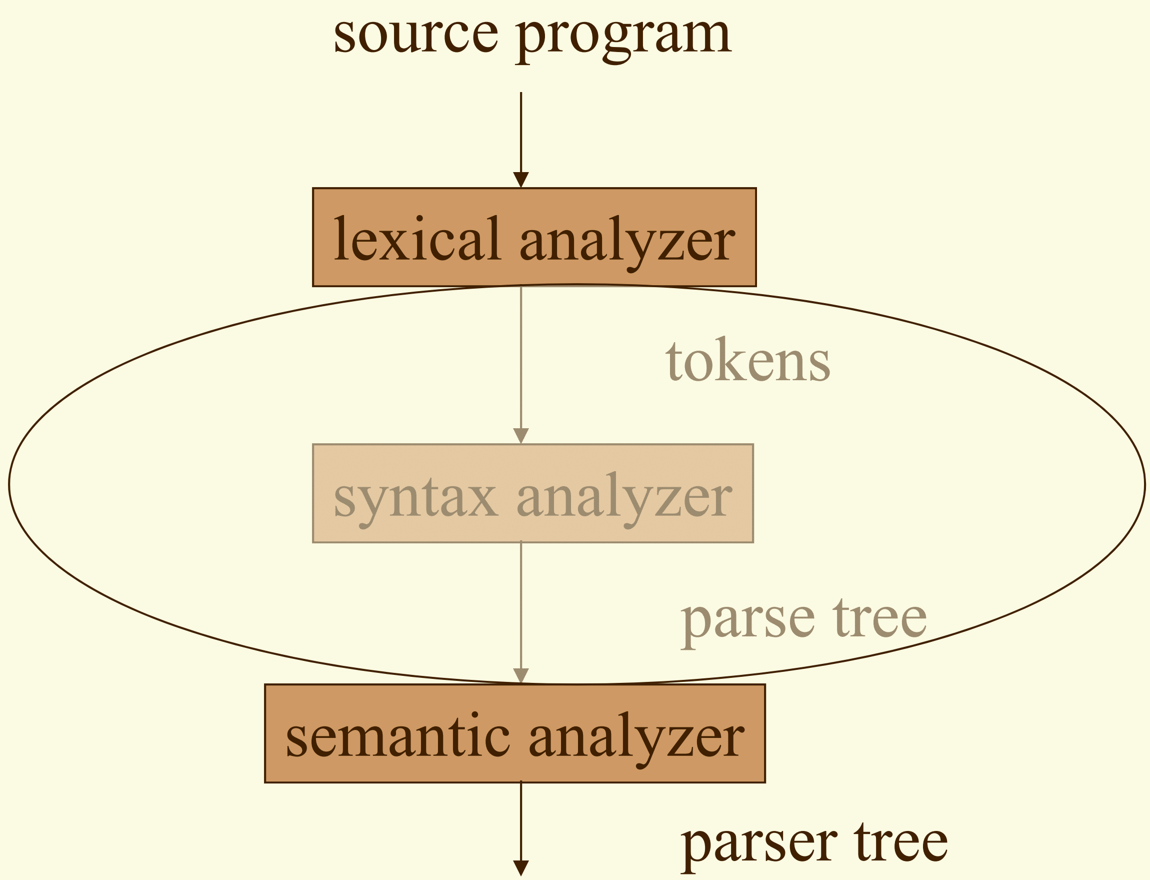
Syntax Analysis



by Neng-Fa Zhou

The Role of the Parser

- Construct a parse tree
- Report and recover from errors
- Collect information into symbol tables

Context-free Grammars

$G=(\Sigma ,N,P,S)$

- Σ is a finite set of terminals
- N is a finite set of non-terminals
- P is a finite subset of production rules
- S is the start symbol

CFG: Examples

- Arithmetic expressions

  E ::= T | E + T | E - T
  T ::= F | T * F |T / F
  F ::= id | (E)

- Statements

  IfStatement ::= if *E* then *Statement* else *Statement*

CFG vs. Regular Expressions

- CFG is more expressive than RE
  – Every language that can be described by regular expressions can also be described by a CFG
- Example languages that are CFG but not RE
  – if-then-else statement, $\{a^n b^n \mid n>=1\}$
- Non-CFG
  – $L1=\{wcw \mid w \text{ is in } (a|b)^*\}$
  – $L2=\{a^n b^m c^n d^m \mid n>=1 \text{ and } m>=1\}$

Derivations

$$\alpha A\beta \Rightarrow \alpha\gamma\beta \quad \text{if} \quad A ::= \gamma$$

$$\begin{cases} \alpha \overset{*}{\Rightarrow} \alpha \\ \alpha \overset{*}{\Rightarrow} \beta \text{ and } \beta \Rightarrow \gamma \text{ then } \alpha \overset{*}{\Rightarrow} \gamma \end{cases}$$

$$S \overset{*}{\Rightarrow} \alpha$$

- α is a sentential form
- α is a sentence if it contains only terminal symbols

Derivations

- leftmost derivation

  $\alpha A \beta \Longrightarrow \alpha \gamma \beta$ if α is a string of terminals

- Rightmost derivation

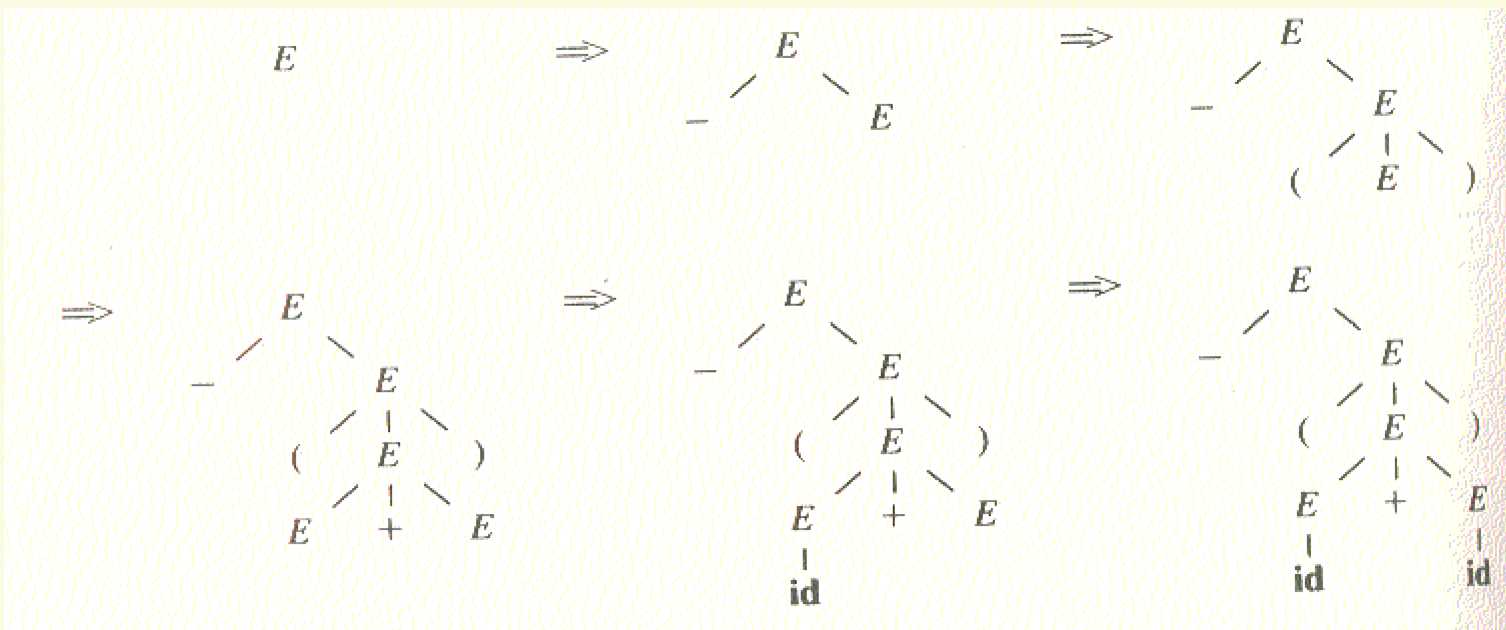
  $\alpha A \beta \Longrightarrow \alpha \gamma \beta$ if β is a string of terminals

Parse Trees

- A parse tree is any tree in which
  - The root is labeled with S
  - Each leaf is labeled with a token a or ε
  - Each interior node is labeled by a nonterminal
  - If an interior node is labeled A and has children labeled X1,.. Xn, then A ::= X1...Xn is a production.

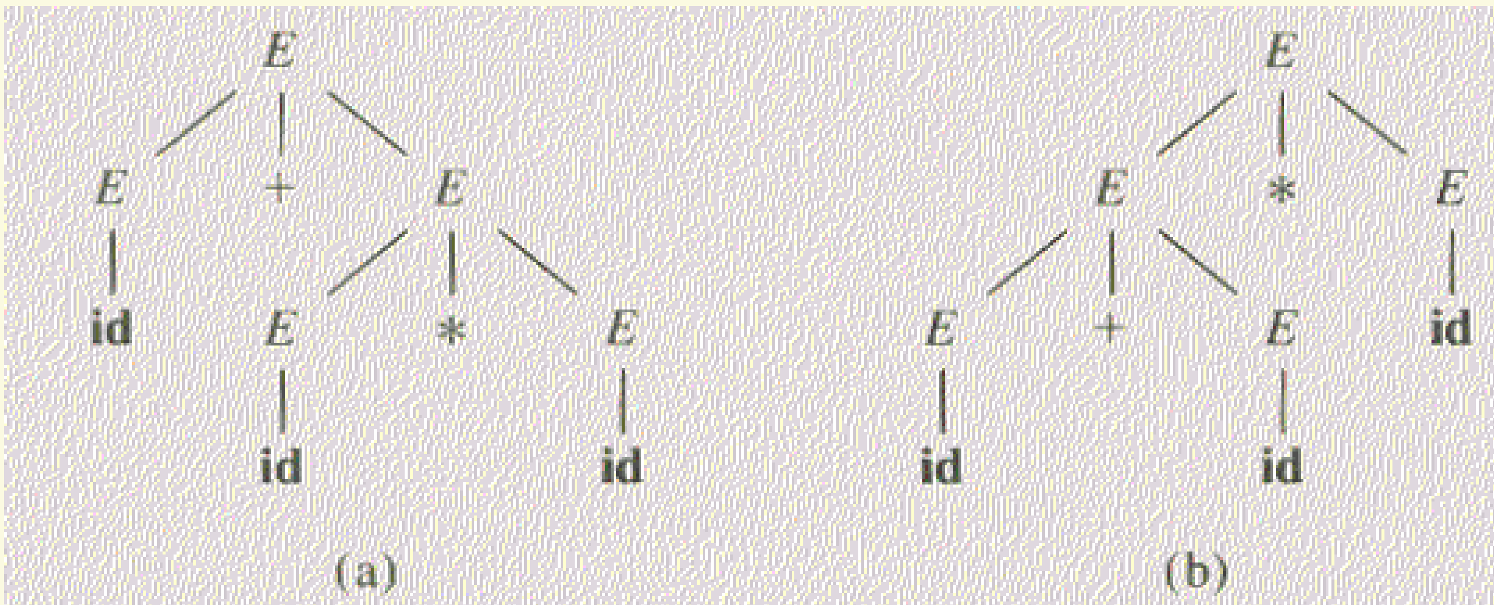
Parse Trees and Derivations

E ::= E + E | E * E | E - E | - E | (E) | id



Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.



Eliminating Ambiguity

- Rewrite productions to take the precedence of operators into account

stmt ::= *matched_stmt* |
unmatched_stmt
matched_stmt ::= if *E* then *matched_stmt* else *matched_stmt* |
other
unmatched_stmt ::= if *E* then *stmt* |
if *E* then *matched_stmt* else *unmatched_stmt*

Eliminating Left-Recursion

- Direct left-recursion

$$A ::= A\alpha \mid \beta$$

$$\Downarrow$$

$$A ::= \beta A'$$
$$A' ::= \alpha A' \mid \varepsilon$$

$$A ::= A\alpha 1 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \ldots \mid \beta n$$

$$\Downarrow$$

$$A ::= \beta 1 A' \mid \ldots \mid \beta n A'$$
$$A' ::= \alpha 1 A' \mid \ldots \mid \alpha n A' \mid \varepsilon$$

Eliminating Indirect Left-Recursion

- Indirect left-recursion

$$S ::= Aa \mid b$$
$$A ::= Ac \mid Sd \mid \varepsilon$$

- Algorithm

```
Arrange the nonterminals in some order A_1,...,A_n.
for (i in 1..n) {
    for (j in 1..i-1) {
        replace each production of the form A_i ::= A_j γ  by the
        productions A_i ::= δ_1 γ | δ_2 γ |... | δ_k γ where
            A_j ::= δ_1 | δ_2 |... | δ_k
    }
    eliminate the immediate left recursion among A_i productions
}
```

Left Factoring

$$A ::= \alpha\beta 1 \mid \ldots \mid \alpha\beta n \mid \gamma$$

↓

$$A ::= \alpha A' \mid \gamma$$
$$A' ::= \beta 1 \mid \ldots \mid \beta n$$

Top-Down Parsing

- Start from the start symbol and build the parse tree top-down
- Apply a production to a nonterminal. The right-hand of the production will be the children of the nonterminal
- Match terminal symbols with the input
- May require backtracking
- Some grammars are backtrack-free (predictive)

Construct Parse Trees Top-Down

- Start with the tree of one node labeled with the start symbol and repeat the following steps until the fringe of the parse tree matches the input string
  - 1. At a node labeled A, select a production with A on its LHS and for each symbol on its RHS, construct the appropriate child
  - 2. When a terminal is added to the fringe that doesn't match the input string, backtrack
  - 3. Find the next node to be expanded
- ! Minimize the number of backtracks

Example

Left-recursive

E ::=	T
	| E + T
	| E - T
T ::=	F
	| T * F
	| T / F
F ::=	id
	| number
	| (E)

Right-recursive

E ::=	T E'
E'::=	+ T E'
	| - T E'
	| e
T::=	F T'
T' ::=	* F T'
	| / F T'
	| e
F ::=	id
	| number
	| (E)

x - 2 * y

Control Top-Down Parsing

- Heuristics
  - Use input string to guide search
- Backtrack-free search
  - Lookahead is necessary
    - Predictive parsing

Predictive Parsing FIRST and FOLLOW

- FIRST(X)
  - If X is a terminal
    - FIRST(X)={X}
  - If X::= ε
    - Add ε to FIRST(X)
  - If $X ::= Y_1, Y_2, \ldots, Y_k$
    - Add FIRST(Y_i) to FIRST(X) if $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$
    - Add ε to FIRST(X) if $Y_1 \ldots Y_k \Rightarrow^* \varepsilon$

Predictive Parsing FIRST and FOLLOW

- FOLLOW(X)
  - Add \$ to FOLLOW(S)
  - If $A ::= \alpha B \beta$
    - Add everything in FIRST(β) except for ε to FOLLOW(B)
  - If $A ::= \alpha B \beta$ ($\beta \Rightarrow^* \varepsilon$) or $A ::= \alpha B$
    - Add everything in FOLLOW(A) to FOLLOW(B)

Recursive Descent Parsing

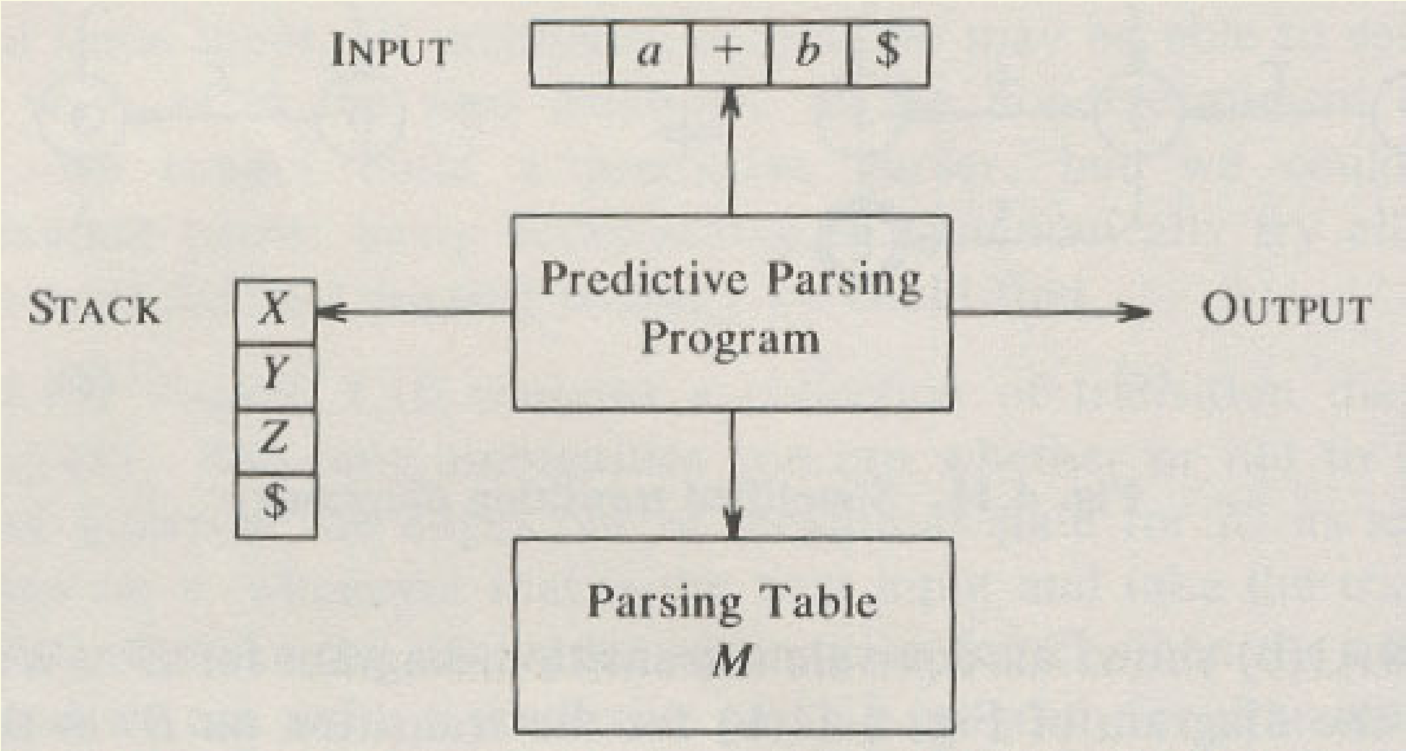
```
match(expected_token){
  if (input_token != expected_token)
    error();
  else
    input_token = next_token();
}

main(){
  input_token = next_token();
  exp();
  match(EOS);
}

exp(){
  switch (input_token) {
  case ID, NUM, L_PAREN:
    term();
    exp_prime();
    return;
  default:
    error();
  }
}
```

```
exp_prime(){
  switch (input_token){
  case PLUS:
    match(PLUS);
    term();
    exp_prime();
    break;
  case MINUS:
    match(MINUS);
    term();
    exp_prime();
    break;
  case R_PAREN,EOS:
    break;
  default:
    error();
  }
}
```

Top-Down Parsing (Nonrecursive predictive parser)



Top-Down Parsing (Nonrecursive predictive parser)

```
set ip to point to the first symbol of w$;
repeat
      let X be the top stack symbol and a the symbol pointed to by ip;
      if X is a terminal or $ then
            if X = a then
                  pop X from the stack and advance ip
            else error()
      else        /* X is a nonterminal */
            if M[X, a] = X → Y1Y2 ··· Yk then begin
                  pop X from the stack;
                  push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
                  output the production X → Y1Y2 ··· Yk
            end
            else error()
until X = $    /* stack is empty */
```

parsing table (pointing to M[X, a])

Example

STACK	INPUT	OUTPUT
$\$E$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
$\$E'T$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E \to TE'$
$\$E'T'F$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$+ \mathbf{id} * \mathbf{id}\$$	
$\$E'$	$+ \mathbf{id} * \mathbf{id}\$$	$T' \to \epsilon$
$\$E'T+$	$+ \mathbf{id} * \mathbf{id}\$$	$E' \to +TE'$
$\$E'T$	$\mathbf{id} * \mathbf{id}\$$	
$\$E'T'F$	$\mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$* \mathbf{id}\$$	
$\$E'T'F*$	$* \mathbf{id}\$$	$T' \to *FT'$
$\$E'T'F$	$\mathbf{id}\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \to \epsilon$
$\$$	$\$$	$E' \to \epsilon$

Parsing Table Construction

for each production p = (A::=α) {

for each terminal a in FIRST(α), add p to M[A,a];

if ε is in FIRST(α)

for each terminal b (including $) in FOLLOW(A)

add p to M[A,b];

}

Example

E ::= E+T | T
T ::= T*F | F
F ::= (E) | id

E ::= TE'
E' ::= +TE' | e
T ::= FT'
T' ::= *FT' | e
F ::= (E) | id

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

LL(1) Grammar

A grammar is said to be LL(1) if |M[A,a]|<=1 for each nonterminal A and terminal a.

- Example (non-LL(1) grammar)

S ::=iEtS | iEtSeS | a
E :: = b

S ::= iEtSS' | a
S' ::= eS | ε
E ::= b

Bottom-Up Parsing

- Start from the input sequence of tokens
- Apply a production to the sentential form and rewrite it to a new one
- Keep track of the current sentential form

Construct Parse Trees Bottom-Up

1. α = the given string of tokens;
2. Repeat (reduction)

   2.1 Matches the RHS of a production
   with a substring of α

   2.2 Replace RHS with the LHS of the production

until no production rule applies (backtrack) or
α becomes the start symbol (success);

Example

S ::= aABe
A ::= Abc | b
B ::= d

abbcde
↓
aAbcde
↓
aAde
↓
aABe
↓
S

Control Bottom-Up Parsing

- Handle
  - A substring that matches the right side of a production
  - Applying the production to the substring results in a *right-sentential form, i.e.,* a sentential form occurring in a right-most derivation
- Example

E ::= E+E
E ::= E*E
E ::= (E)
E ::= id

<u>id</u> + id * id
E + <u>id</u> * id
E + E * <u>id</u>
<u>E + E</u> * E

Bottom-Up Parsing
Shift-Reduce Parsing

```
push '$' onto the stack;
token = nextToken();
repeat
        if (there is a handle  A::=β on top of the stack){
                reduce β  to  A; /* reduce */
                pop β off the stack;
                push A onto the stack;
        } else {/* shift */
                shift token onto the stack;
                token = nextToken();
        }
until (top of stack is S and token is eof)
```

Example

	Stack	Input	Action
(1)	$\$$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(2)	$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	reduce by $E \rightarrow \mathbf{id}$
(3)	$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(4)	$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	shift
(5)	$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	reduce by $E \rightarrow \mathbf{id}$
(6)	$\$E + E$	$* \mathbf{id}_3\$$	shift
(7)	$\$E + E *$	$\mathbf{id}_3\$$	shift
(8)	$\$E + E * \mathbf{id}_3$	$\$$	reduce by $E \rightarrow \mathbf{id}$
(9)	$\$E + E * E$	$\$$	reduce by $E \rightarrow E * E$
(10)	$\$E + E$	$\$$	reduce by $E \rightarrow E + E$
(11)	$\$E$	$\$$	accept

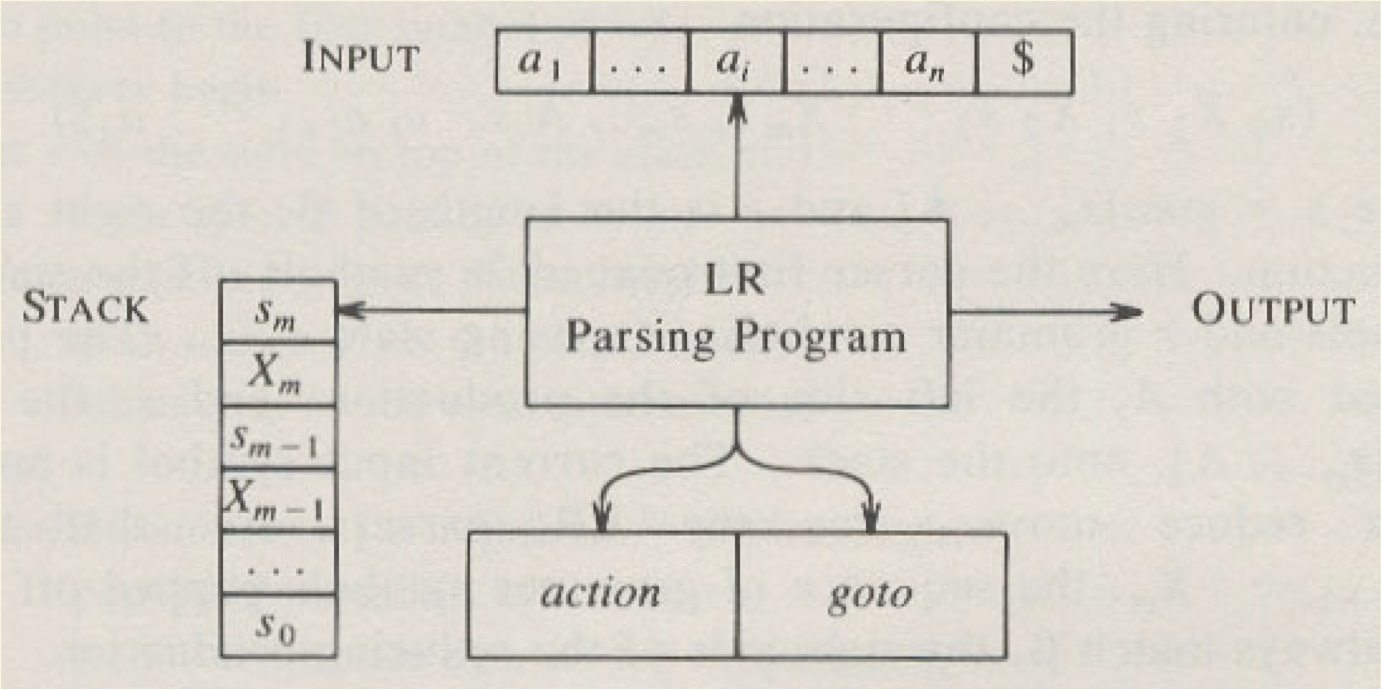
A Problem in Shift-Reduce Parser

The stack has to be scaned to see whether a handle appears on it.

↓

Use a state to uniquely identify a part of a handle (*viable prefix*) so that stack scanning becomes unnecessary

LR Parser



INPUT
a_1 . . . a_i . . . a_n $
STACK
s_m
X_m
s_{m-1}
X_{m-1}
. . .
s_0
LR
Parsing Program
OUTPUT
action
goto

LR Parsing Program

```
set ip to point to the first symbol of w$;
repeat forever begin
      let s be the state on top of the stack and
            a the symbol pointed to by ip;
      if action[s, a] = shift s' then begin
            push a then s' on top of the stack;
            advance ip to the next input symbol
      end
      else if action[s, a] = reduce A → β then begin
            pop 2*|β| symbols off the stack;
            let s' be the state now on top of the stack;
            push A then goto[s', A] on top of the stack;
            output the production A → β
      end
      else if action[s, a] = accept then
            return
      else error()
end
```

Example

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

id * id + id

STATE	action						goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		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			

LR Grammars

- LR grammar
  - A grammar is said to be an LR grammar if we can construct a parsing table for it.
- LR(k) grammar
  - lookahead of up to k input symbols
- SLR(1), LR(1), and LALR(1) grammars

SLR Parsing Tables

- LR(0) item
  - A production with a dot at some position of the RHS

A ::= •XYZ	we are expecting XYZ
A ::= X •YZ	
A ::= XY•Z	
A ::= XYZ•	we have seen XYZ

Closure of a Set of Items I

```
function closure ( I );
begin
    J := I;
    repeat
        for each item A → α·Bβ in J and each production
            B → γ of G such that B → ·γ is not in J do
                add B → ·γ to J
    until no more items can be added to J;
    return J
end
```

Closure of a Set of Items I Example

E' ::= E
E ::= E+T | T
T ::= T*F | F
F ::= (E) | id

closure({E'::= • E}) = ?

The goto Operation

goto(I,X) =closure({A ::= αX• β | A ::= α •X β is in I})

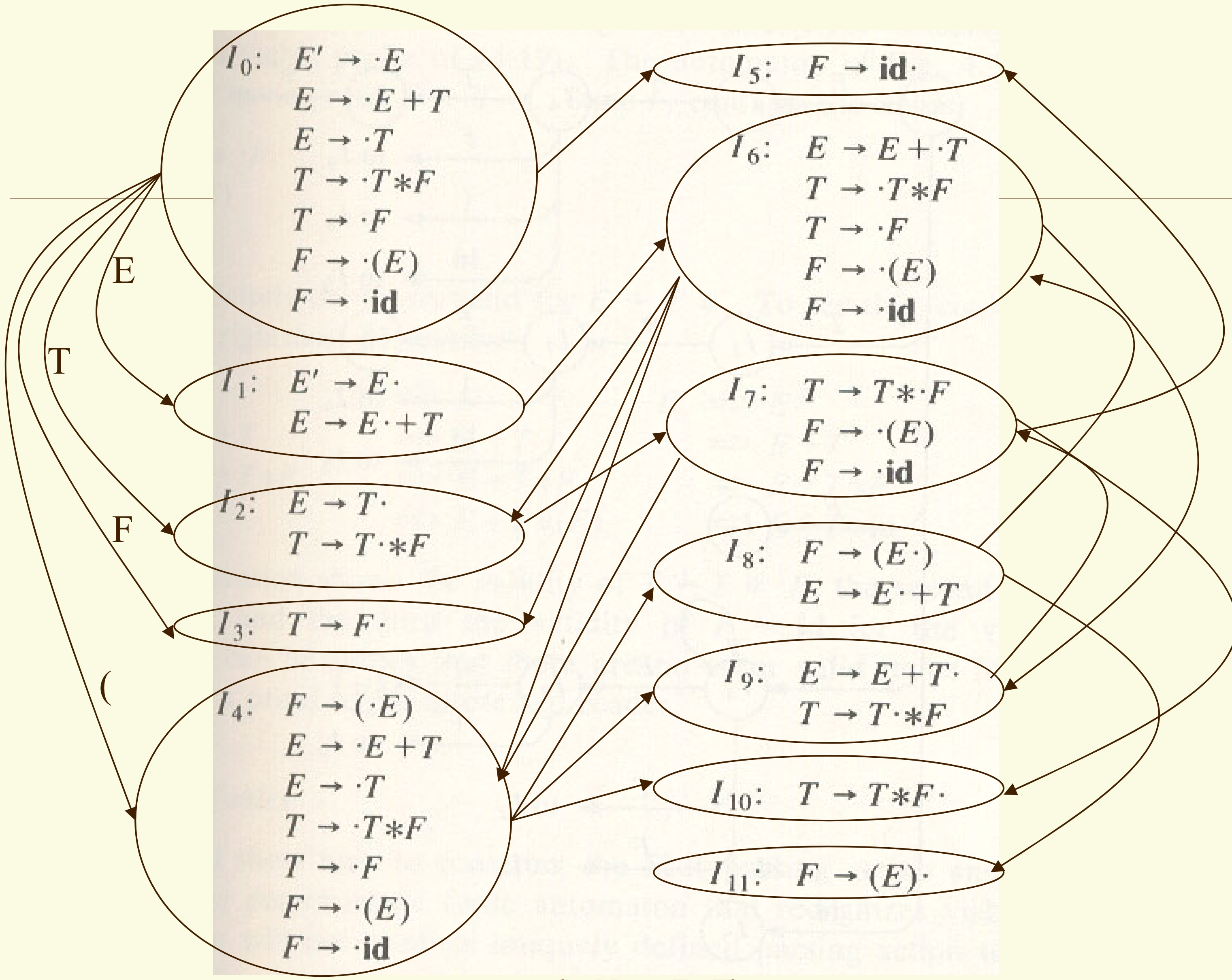
- Example

  I = {E' ::= E• , E ::= E•+ T}

  goto(I,+) = ?

Canonical LR(0) Collection of Set of Items

```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```


I0: E' → ·E
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I1: E' → E·
E → E·+T
I2: E → T·
T → T·*F
I3: T → F·
I4: F → (·E)
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I5: F → id·
I6: E → E+·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I7: T → T*·F
F → ·(E)
F → ·id
I8: F → (E·)
E → E·+T
I9: E → E+T·
T → T·*F
I10: T → T*F·
I11: F → (E)·
E
T
F
(

Constructing SLR Parsing Table

1. Construct C={I0,I1,…,In}, the collection of sets of LR(0) items for G' (augmented grammar).

2. If [A→α•aβ] is in Ii where a is a terminal and goto(Ij,a)=Ij, the set action[i,a] to "shift j".

3. If [S'→S•] is in Ii, then set action[i,$] to "accept".

4. If [A→α•] is in Ii, then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).

Example

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

id * id + id

STATE	action						goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		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			

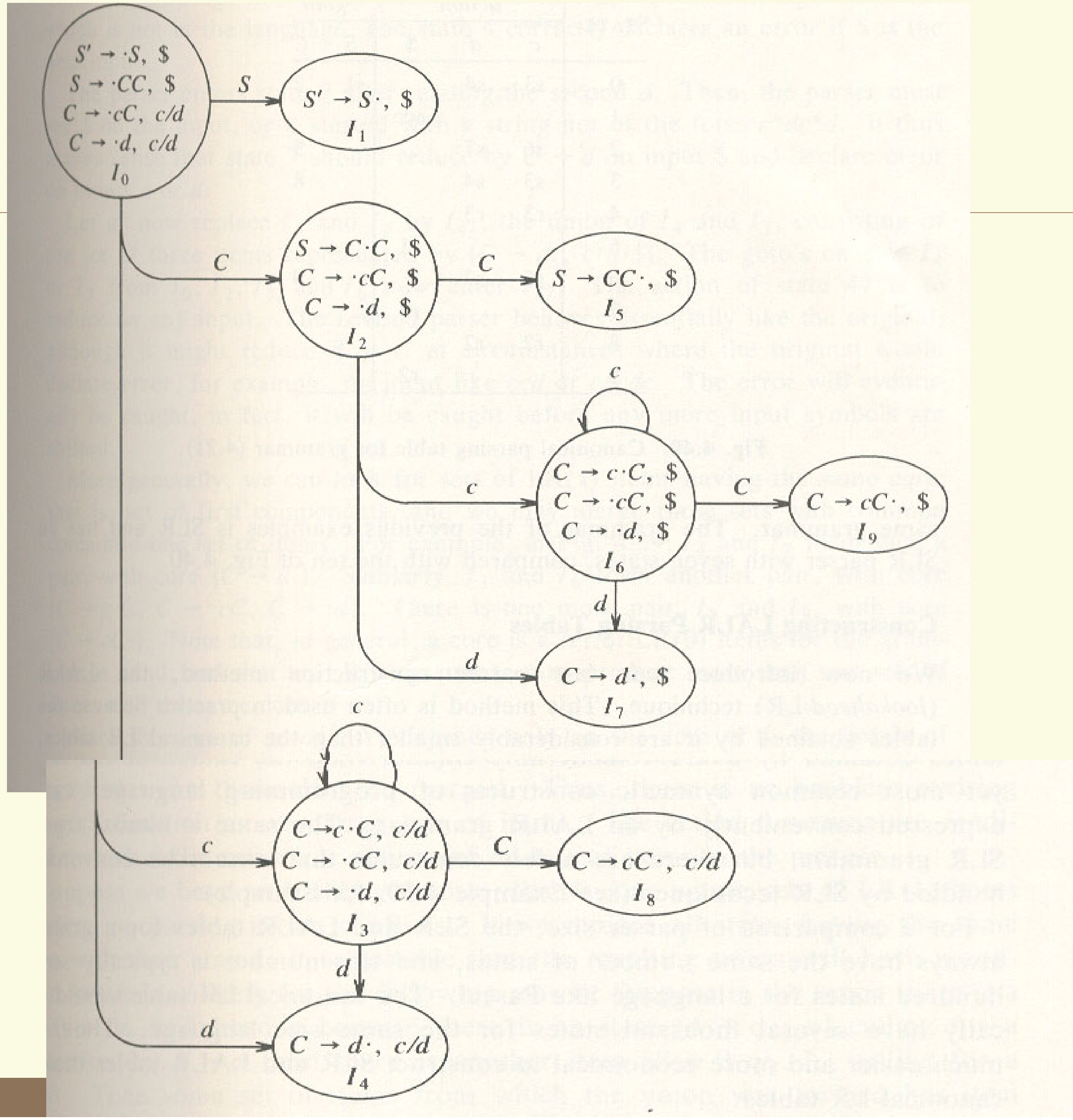
Unambiguous Grammars that are not SLR(1)

S ::= L = R
S ::= R
L ::= * R
L ::= id
R ::= L

LR(1) Parsing Tables

- LR(1) item
  - LR(0) item + one look ahead terminal

- [A::=α• ,a]
  - reduce α to A only if the next symbol is a


S' → ·S, $
S → ·CC, $
C → ·cC, c/d
C → ·d, c/d
I0
S
S' → S·, $
I1
C
S → C·C, $
C → ·cC, $
C → ·d, $
I2
C
S → CC·, $
I5
c
C → c·C, $
C → ·cC, $
C → ·d, $
I6
C
C → cC·, $
I9
d
C → d·, $
I7
c
C → c·C, c/d
C → ·cC, c/d
C → ·d, c/d
I3
C
C → cC·, c/d
I8
d
C → d·, c/d
I4

LALR(1)

- Treat item closures Ii and Ij as one state if Ii and Ij differs from each other only in look ahead terminals.

Descriptive Power of Different Grammars

LR(1) > LALR(1) > SLR(1)