Syntax Analysis

- Every *prog lang* has rules to provide syntactic structure of well-formed programs

- A program made of blocks
- A block made of statements
- A statement made of expressions
- An expression of tokens

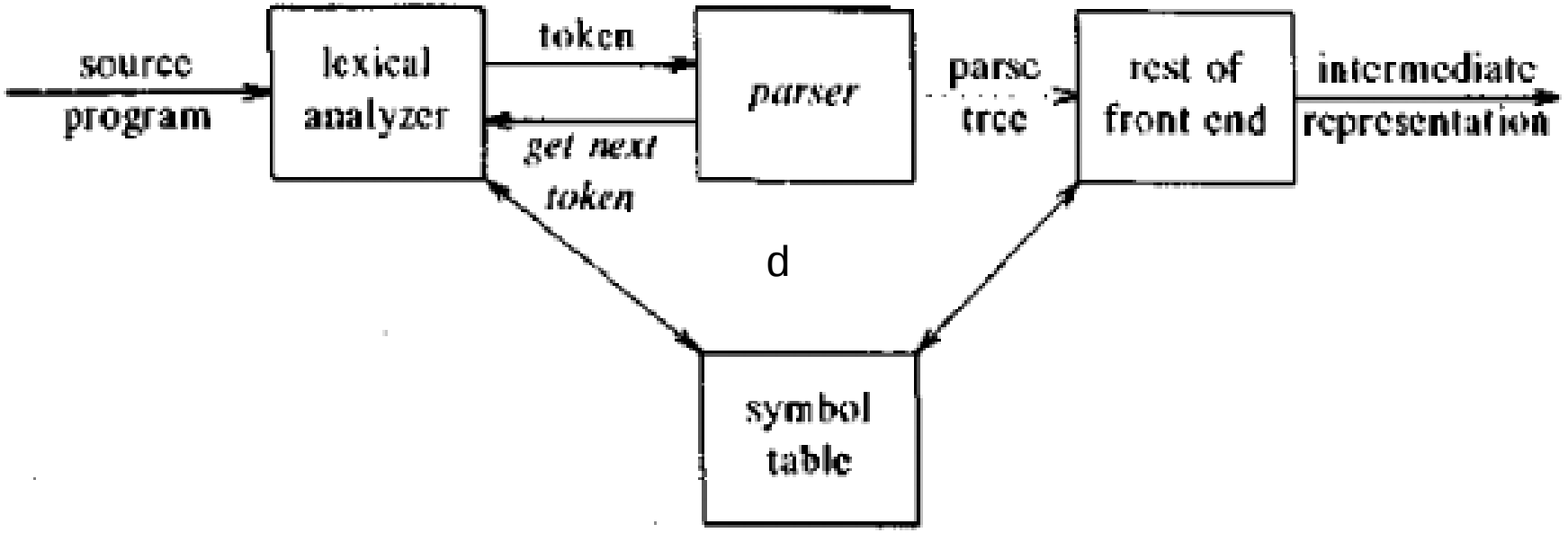
- Syntax of a prog lang described by context-free grammar or BNF

Why grammar?

Grammars offer significant advantages to language designers and compiler writers

- Precise, easy-to-understand, syntactic specification
- Easy means to construct parser to check whether source pgm is well-formed
- Detect errors and imparts a structure useful to translate *src* to *obj* code
- New constructs are easy to incorporate

Role of a parser



Parsing

- Output of parser is representation of parse tree
- Other tasks
  - collecting info about tokens in symbol table
  - type checking
  - semantic analysis *(later chapter)*
  - intermediate code *(later chapter)*

Compilation Errors

- **Lexical:** such as misspelling an identifier, keyword, or operator
- **Syntactic:** such as an arithmetic expression with unbalanced parentheses
- **Semantic:** such as an operator applied to an incompatible operand
- **Logical:** such as an infinitely recursive call

Goals of Error handlers

- report errors clearly and accurately
- recover from error quickly to detect subsequent errors
- *shd not* slow down processing of correct pgms

Error Recovery

- **panic mode:** discard symbols until a delimiter found
- **phrase level:** local correction on the rem. i/p

  - e.g. replace a comma by semi-colon

  - caution! (no infinite loop)
- **error productions:** if common errors known, can be simulated w/ grammars –> suitable action
- **global correction:** globally least-cost correction

  - For a string ***x*** and grammar ***G***, generate correct parse tree for a string ***y*** - w/ min. # insertions, deletions, changes of tokens

  s.t. ***x*** can be converted to ***y***

Context-Free Grammars

If S_1, and S_2 are statements and E an expression,

then

"**if** E **then** S_1 **else** S_2" is a statement

This statement can't be specified using RE!

Using syntactic var *stmt* to denote *class of statements* and *expr* to denote *class of expressions,* and following grammar

stmt → **if** *expr* **then** *stmt* **else** *stmt*

CFG & Parsing

- G = (T,V,S,P)
- T: finite set of terminals

  - tokens like **if, then, else, for, while** etc.
- V: finite set of non-terminals

  - syntactic vars denoting set of strings like *stmt* and *expr*
- S: Start symbol, a non-terminal denoting language defined by G or *L(G)*
- P: finite set of rules specifying how T and V can be combined to form strings

Example

$$
\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow \mathbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}
$$

Example

- $expr \rightarrow expr\ op\ expr$
- $expr \rightarrow (\ expr\ )$
- $expr \rightarrow -\ expr$
- $expr \rightarrow \mathbf{id}$
- $op \rightarrow +$
- $op \rightarrow -$
- $op \rightarrow *$
- $op \rightarrow /$
- $op \rightarrow \mathbf{t}$

- Terminals (T): + - * / ( )
- Non-terminals (V): *expr, op*
- Start sysmbol (S): *expr*

Notational Conventions

- Terminals (T)
- i) Lower-case letters early in the alphabet such as *a, b, c*.
- ii) Operator symbols such as +, - , etc .
- iii) Punctuation symbols such as (,),{,},[,], comma(,) etc.
- iv) The digits 0, 1 , . . . , 9.
- V) Boldface strings such as **id** or **if**.

Parsing: CFG Notations

Non-Terminals (V)

- Upper-case letters early in the alphabet such as A, B, C
- The letter S, which, when it appears, is usually the start symbol.
- Lower-case italic names such as *expr* or *stmt*

Notations

Grammar Symbols

- Upper-case letters late in the alphabet, such as X, Y, Z, represent grammar symbols, i.e. either nonterminals or terminals.
- Lower-case letters late in the alphabet, chiefly u, v , . . . , z, represent strings of terminals.
- Lower-case Greek letters, α, β, γ represent strings of grammar symbols.
- $A \rightarrow \alpha$
- means A (non-terminal) produces α (terminal and/or non-terminals)
- $A \rightarrow \alpha_1$,
- $A \rightarrow \alpha_2$,

  ….

  $A \rightarrow \alpha_k$

- Can be written as $A \rightarrow \alpha_1 | ... | \alpha_k$

Example

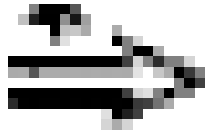
- E → E A E | (E) | -E | **id**
- A → +|-| * | / |↑

Derivation

- E → E + E | E * E | (E) | -E | **id**

  From the above grammar

  E => -E (we call E *derives* -E)
- E => -E => -(E) => -(id)

  '=>' means derive in one step
- The following symbol means 'zero or more steps'
- 

  $\alpha \overset{*}{\Rightarrow} \alpha$ for any string α, and

  If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.
- 

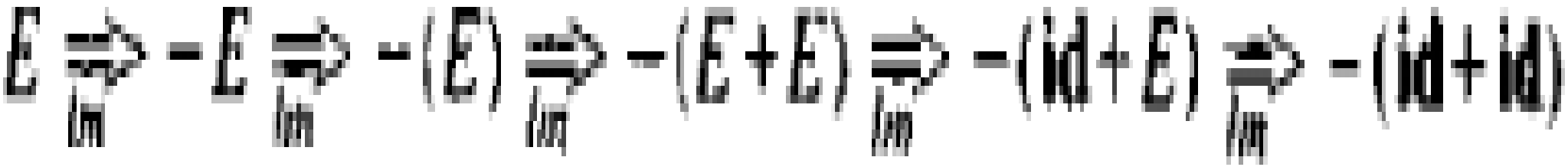
  to mean "derives in one or more steps.

CFG: CFL

- Strings in L(G) can contain only terminals of G
- A string of terminals *w* is in *L(G) iff* 
- *w* is called *sentence* of G
- If 
- Where α may contain non-terminals, we say α is sentential form of G.
- A sentence is a sentential form w/o non-terminals (i.e. having only terminals).

Leftmost derivation

- only the leftmost nonterminal in any sentential form is replaced at each step (by terminals/non-terminals)
-



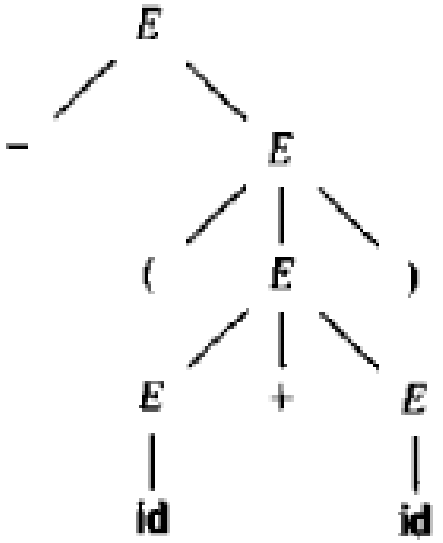
-
- Similarly, *rightmost* derivations (also known as canonical derivations)

Parse Trees and Derivations

- A parse tree is a *graphical representation* for a derivation that *filters out choice regarding replacement order*
- Interior nodes are non-terminals
- Leaves are nonterminals or terminals

  - read from L-to-R, give a sentential form (called *yield of a tree*)

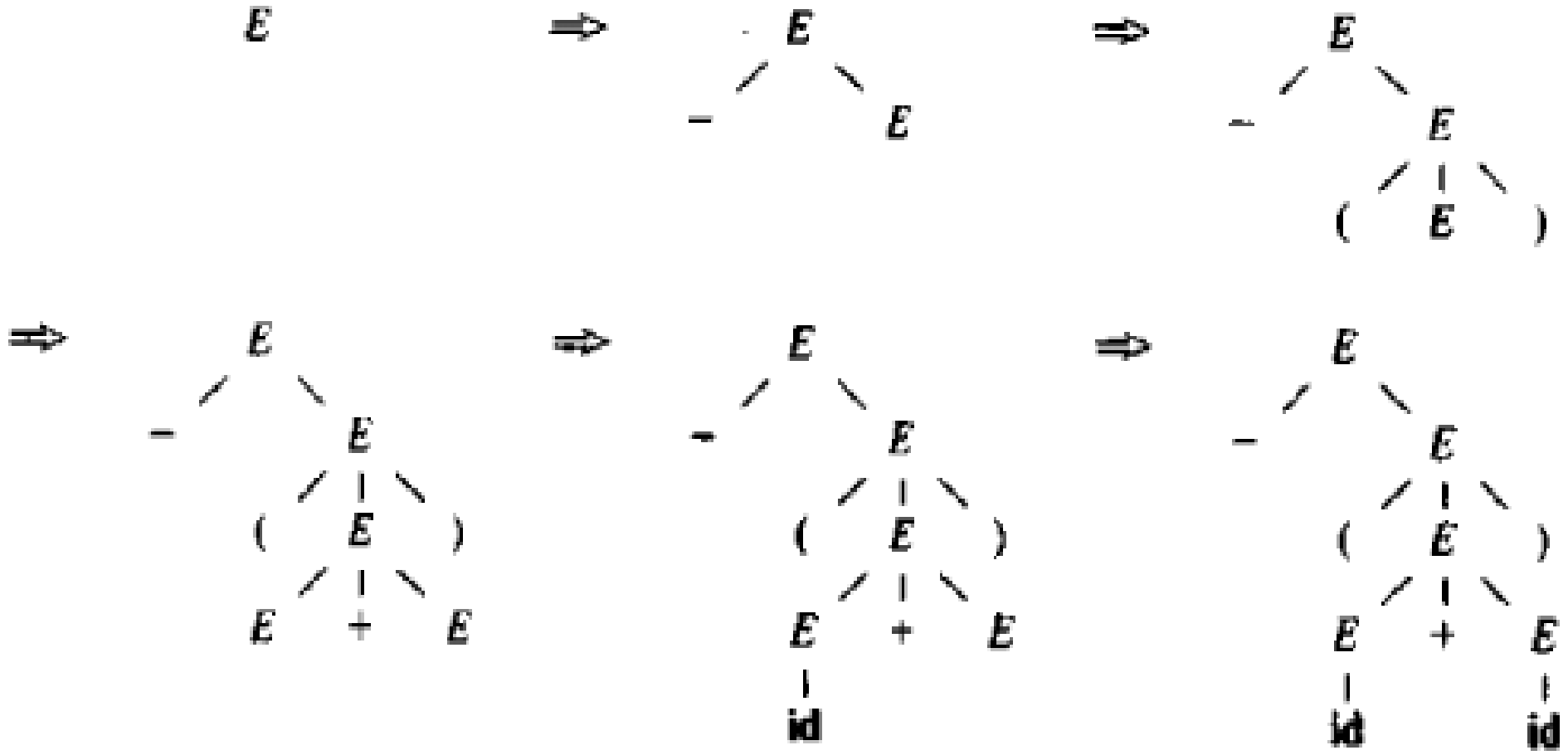
Examples



Parse tree for -(**id** + **id**)

Building Parse Tree from Derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$



Parse Tree: points to remember

- A parse tree ignores variations in order of replacement

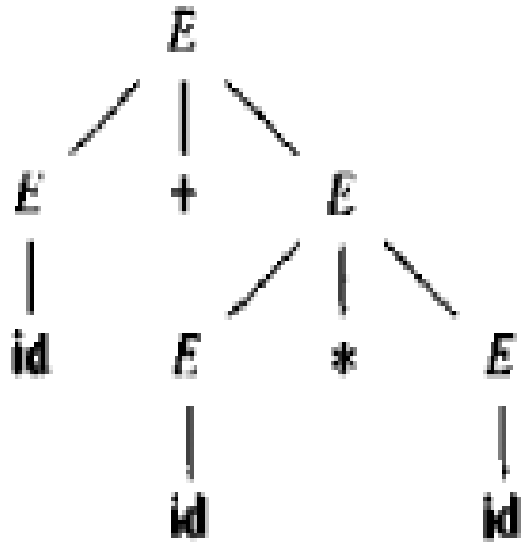
  [E=> -E =>-(E)=>-(E+E)=>-(E+**id**) => -(**id**+**id**)]

- But every parse tree is associated with a unique leftmost or unique rightmost derivation
- A sentence may have more than one leftmost or rightmost derivation

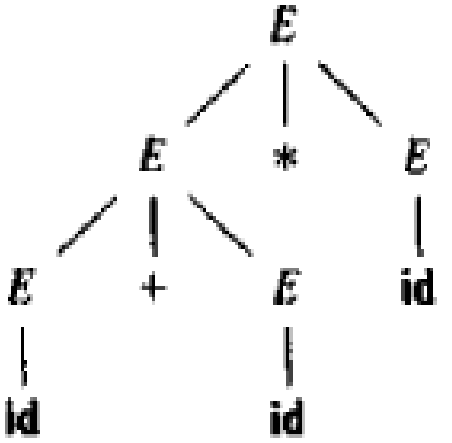
Ambiguity

- E → E+E | E*E| (E) | -E | **id**
- Both are leftmost derivations

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$



$$
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$



Parsing: Points to remember

- For most parsers, we need *unambiguous* grammar
- Even if we consider ambiguous grammar we need *disambiguating rules* to rule out undesirable parse trees

  ==> There should be ONLY ONE parse tree for each sentence.

- HOW TO REMOVE AMBIGUITY?

Why to remove ambiguity?

- Parsing determines whether a string of tokens can be generated by a grammar
- While scanning the I/p, parse tree is thought to be constructed
- Two broad types: 1. Top-down 2. Bottom-Up

Top-Down Parsing

- *type* → *simple*
- | ↑ **id**
- | **array** [*simple*] **of** *type*
- *simple* → **integer**
- | **char**
- | **num dotdot num**

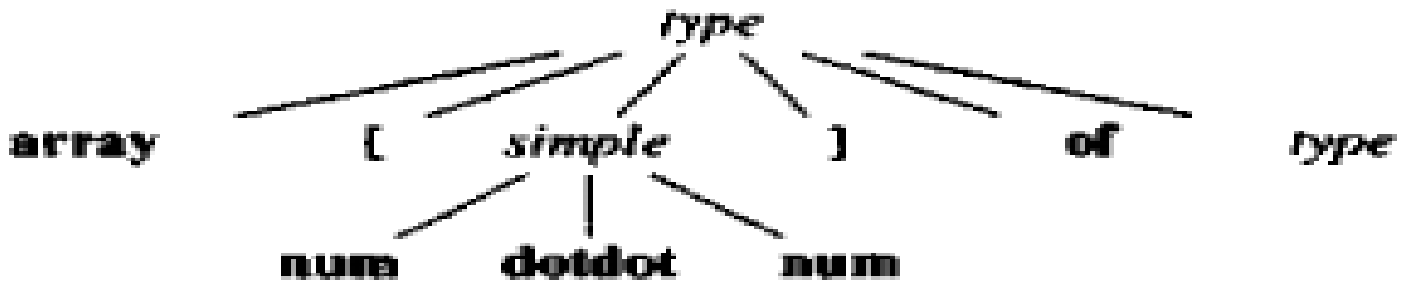
(a)


type

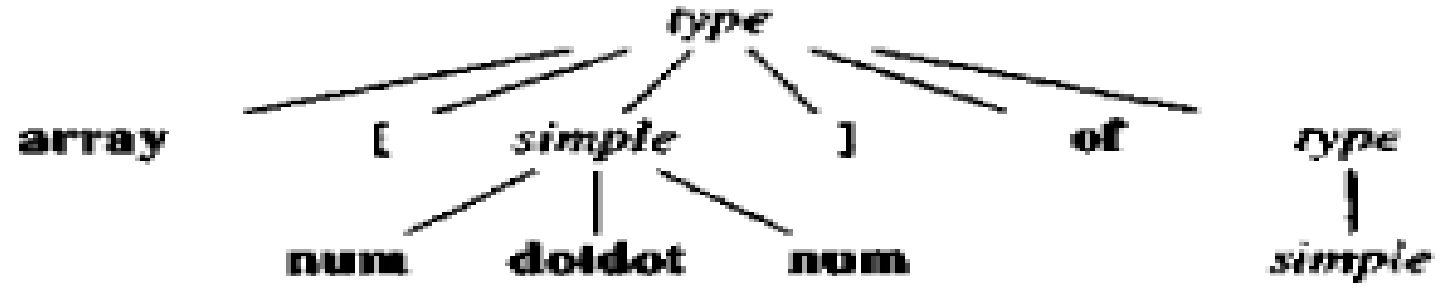
(b)


type
array
[
simple
]
of
type

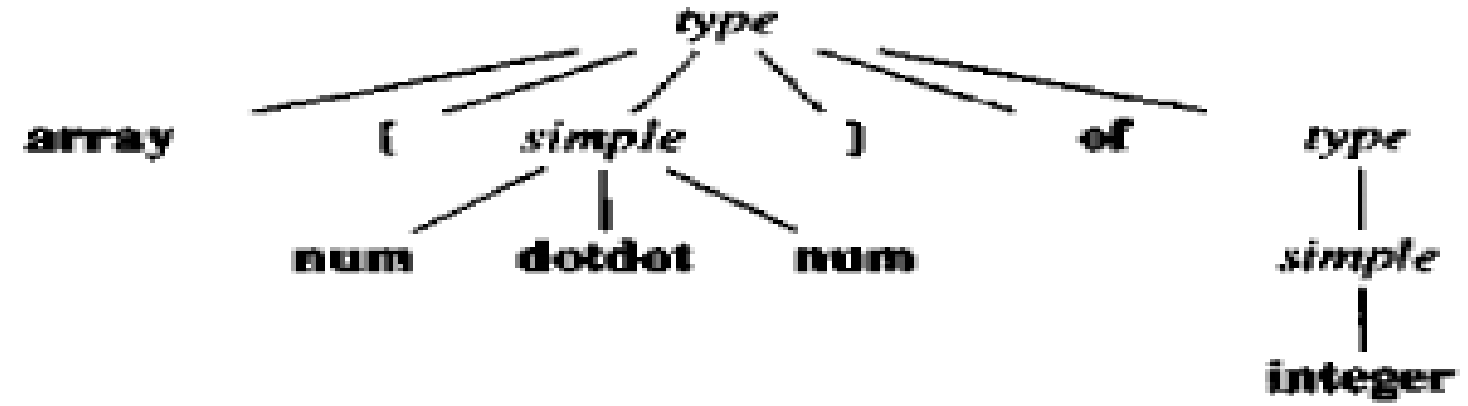
(c)


type
array
[
simple
]
of
type
num
dotdot
num

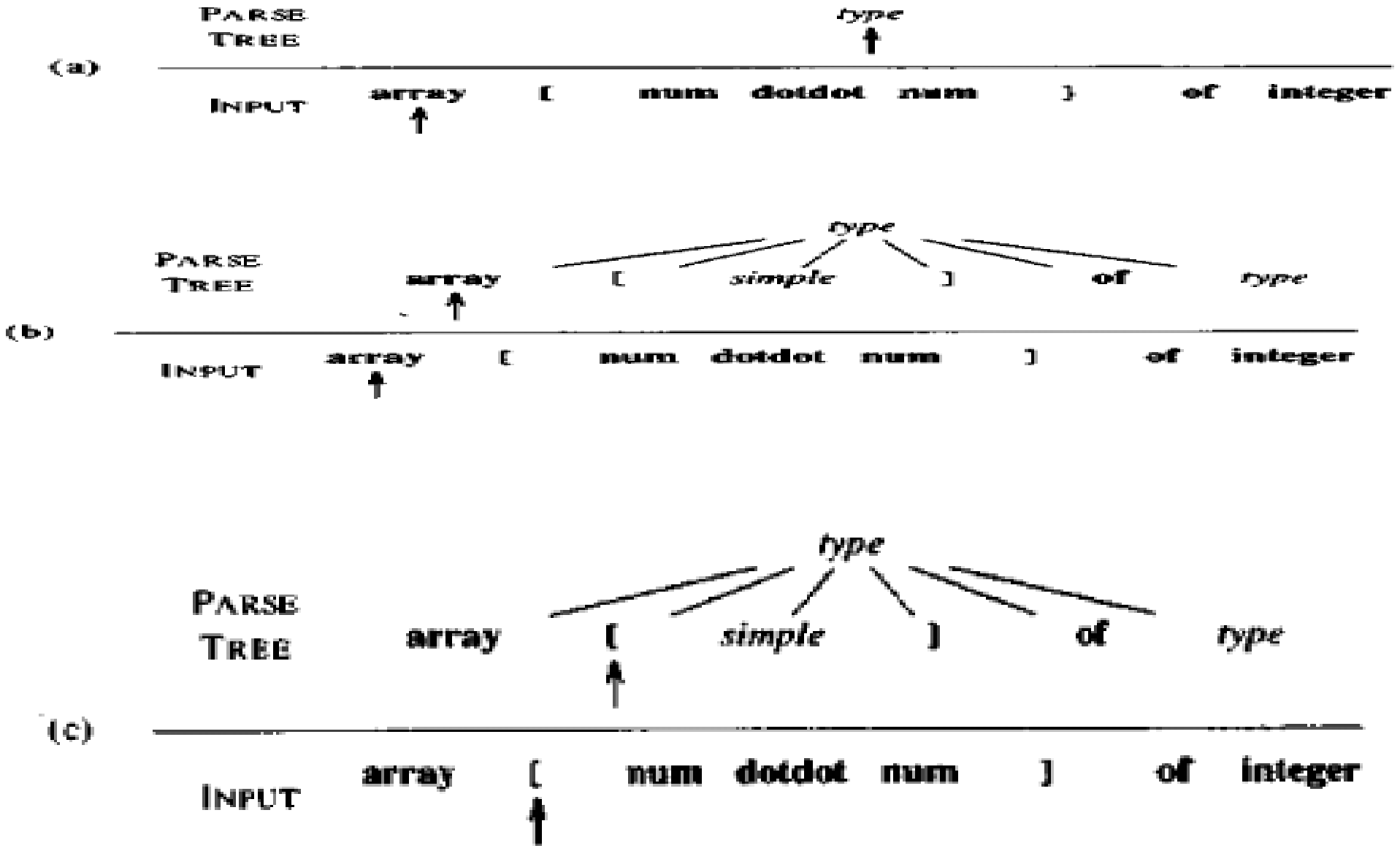
(d)


type
array
[
simple
]
of
type
num
dotdot
num
simple

(e)


type
array
[
simple
]
of
type
num
dotdot
num
simple
integer

array [num dotdot num] of integer



Why ambiguity is bad?

- The selection of a non-terminal involves trial-and-error
- Try a production, backtrack if not suitable, try another
- (Predictive parsing does not need backtracking ==> but needs UnAMBIGUOUS Grammar)

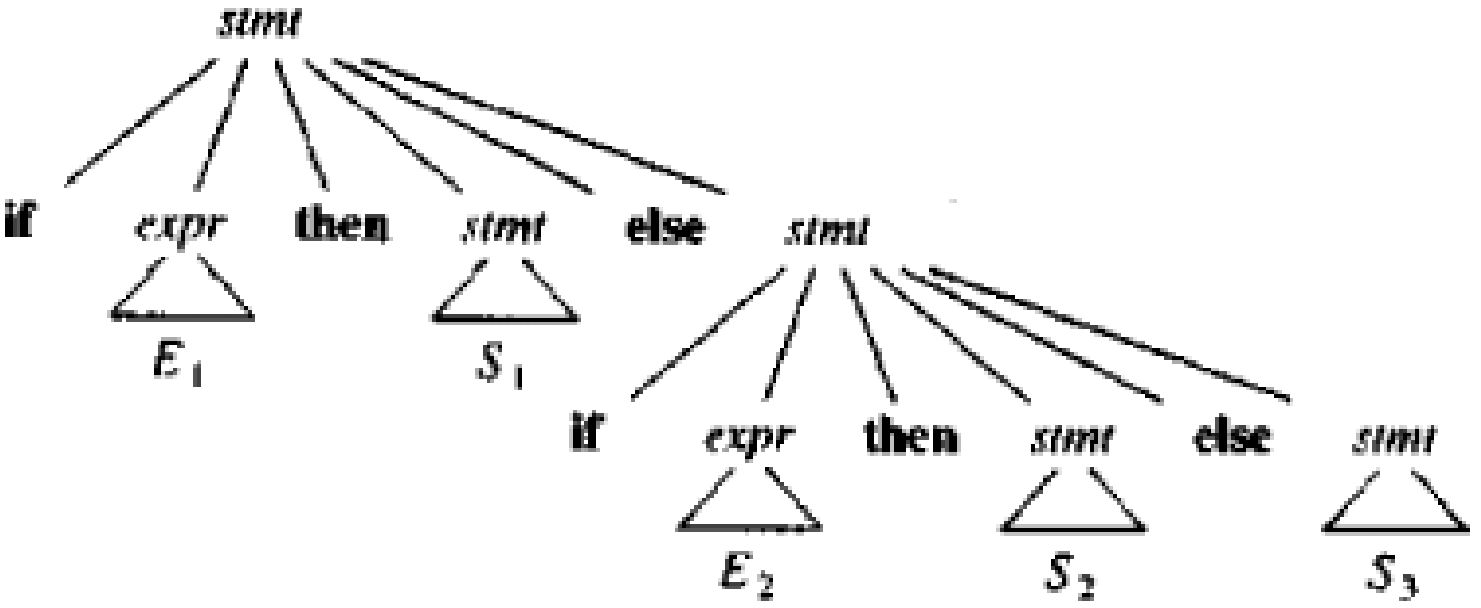
Eliminating Ambiguity

stmt → **if** *expr* **then** *stmt*

| ***if*** *expr* ***then*** *stmt* **else** *stmt*

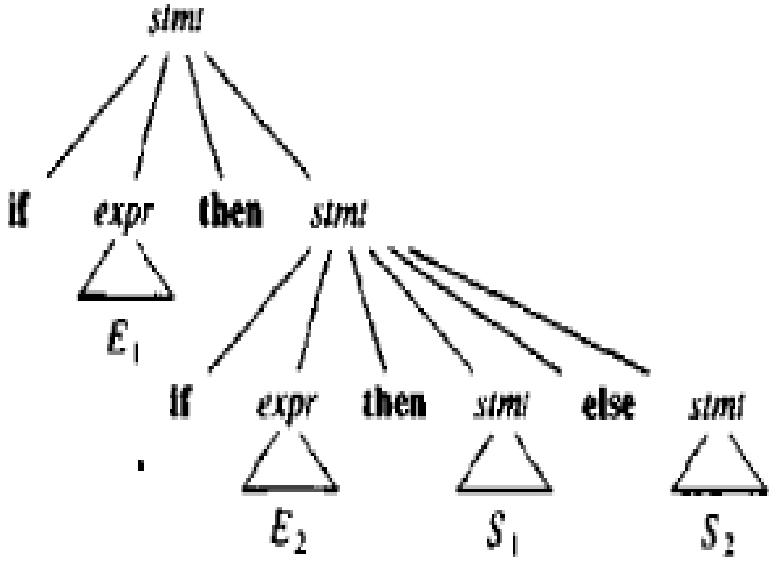
| **other**

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3



Ambiguity

if E_1 **then if** E_2 **then** S_1 **else** S_2



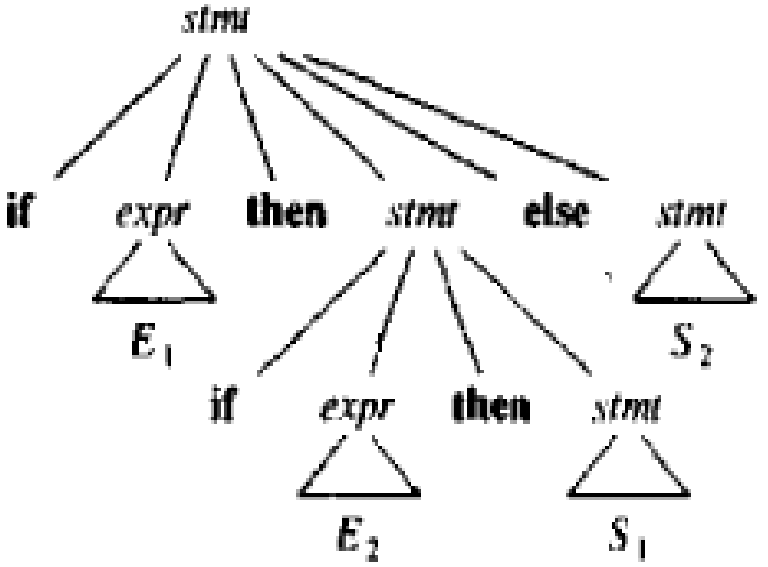


Fig 1

Fig 2

Both are ***correct,*** but which one is *acceptable?*

Disambiguating Rule

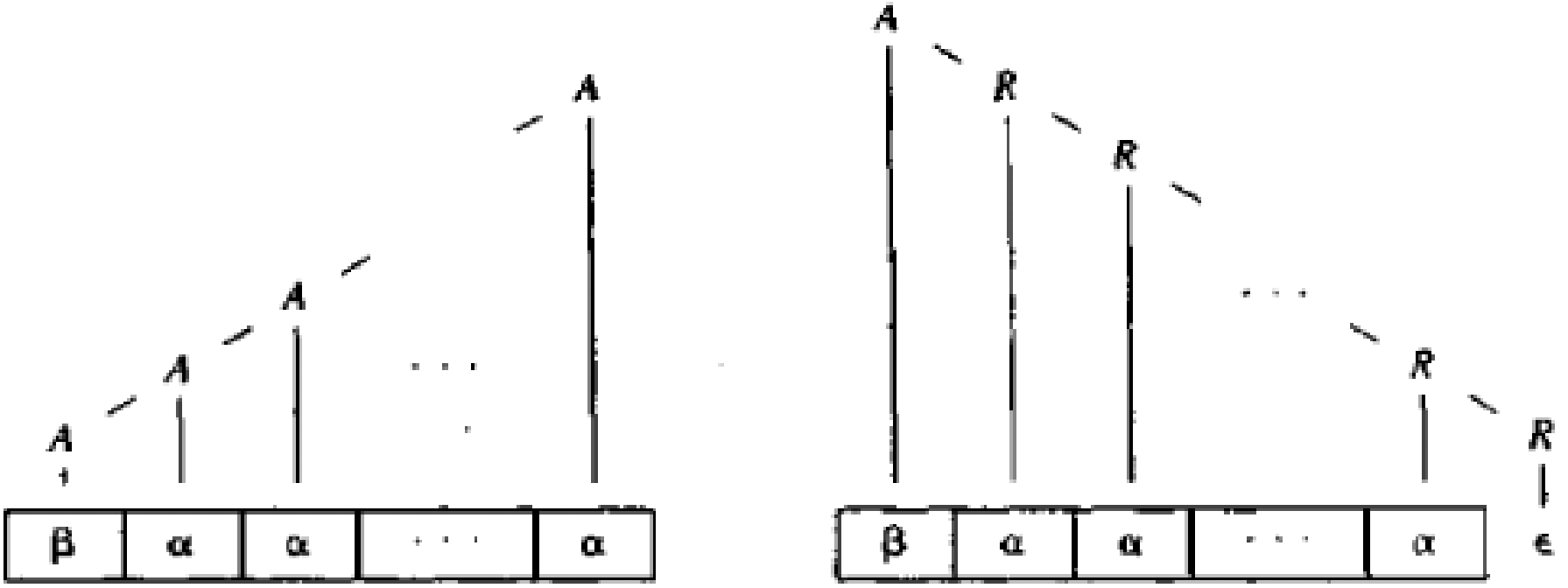
- Match each **else** with the closest previous unmatched **then**

stmt → *matched_stmt*
| *unmatched_stmt*
matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt*
| **other**
unmatched_stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Left Recursion

- A → Aα | β
- *expr → expr + term | term* *(or, E → E + T|T)*

  A = *expr*, α = +*term*, β = *term*



- A → βR
- R → αR| ε

Example

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

Soln

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

Removal of Left Recursion

- Left-recursive
- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid ... \mid \beta_1 \mid ... \mid \beta_n$

- Right-recursive
- $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$
- $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid .... \mid \alpha_m A' \mid \varepsilon$
-
- All immediate left-recursion eliminated from A and A' productions (provided no α_i is ε)

But....

- Derivations of two or more steps
- $S \rightarrow Aa \mid b$
- $A \rightarrow Ac \mid Sd \mid \varepsilon$

- $S \Rightarrow Aa \Rightarrow Sda$ (left recursive)

Eliminate Left Recursion

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. **for** i = 1 to n **do begin**

   for j= 1 to i - 1 **do begin**

   replace each $Ai \rightarrow Ajy$

   by $Ai \rightarrow \delta_1 y \mid \delta_2 y \mid \ldots \mid \delta_k y$.

   where $Aj \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current productions

   end

   eliminate immediate left recursion of A_i-productions

end

(Can remove for no-cycle $A \overset{+}{\Rightarrow} A$ or $A \rightarrow \varepsilon$)

Soln.

Problem

- *S* → *Aa* | *b*
- *A* → *Ac* | *Sd* | *ε*

- *S* =>*Aa* =>*Sda* (left-recursion)

- The algo not guaranteed to work as there is A --> *ε*
- i=1: j=0 ==> no immediate left recursion
- I=2, j=1: A → Sd gives A → Ac|Aad|bd|*ε*

- <u>Final soln</u>
- S → Aa | b
- A → bdA' | A
- A' → cA' | adA'| *ε*

Left Factoring

- A kind of grammar transformation (suitable for predictive parsing)
- If not clear which of 2 productions to expand A, defer the decision until enough is seen

  stmt → **if** *expr* **then** *stmt*

  | ***if*** *expr* ***then*** *stmt* **else** *stmt*

- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
- $A \rightarrow \alpha A'$
- $A' \rightarrow \beta_1 \mid \beta_2$.

Algorithm: Left factoring

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma \quad (\alpha \neq \varepsilon)$

  then
- $A \rightarrow \alpha A' \mid \gamma$
- $A' \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
- Repeatedly apply this until no 2 alternatives have a common prefix.

Left Factoring: Example

- $S \rightarrow iEtS \mid iEtSeS \mid a$ [*i*: **if,** *t*: **then,** *e*: **else]**
- $E \rightarrow b$ [*E*: expr, *S*: statement]

After left-factoring

- $S \rightarrow iEtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $E \rightarrow b$

On input *i*, expand iEtSS' and wait until iEtS is seen to decide S' → eS or S' → ε

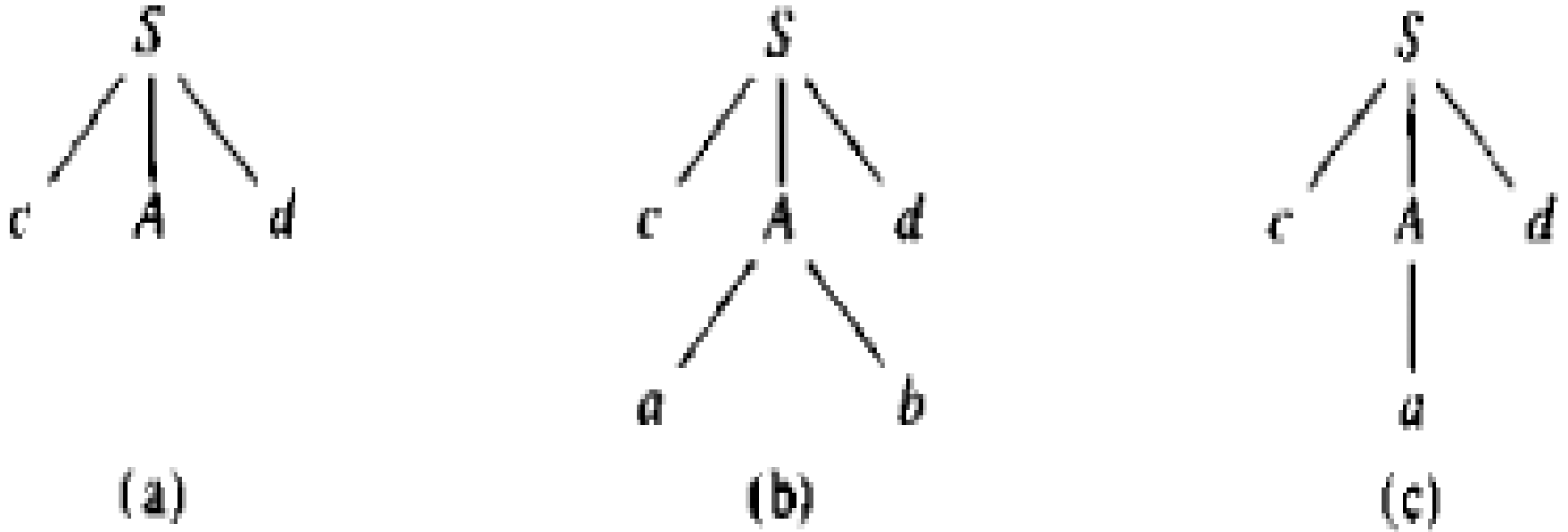
(Both grammars AMBIGUOUS, as on input ***e***, we are clueless!)

Top-Down Parsing

- Tries to find a leftmost derivation for an i/p string
- Builds a parse tree in pre-order
- General form: *Recursive Descent* (RD)
- May involve backtracking
- *Predictive Parsing* (a special type of RD) does not involve backtracking

An example (involves backtrack)

- $S \rightarrow cAd$
- $A \rightarrow ab \mid a$
- Input string: $w = cad$



Recursive Descent

void ***A()*** {

Choose an A-production, $A \rightarrow X_1 X_2 \ldots X_k$

for (i = 1 to k) {

if (X_i is a nonterminal)

call procedure $X_i()$;

else if (X_i equals the current input symbol ***a***)

advance the input to the next symbol;

else /* an error has occurred */;

}

}

Predictive Parsers

- Left-recursive grammar can cause a RD parser, (even w/ backtrack) to go into infinite loop
- Remove left recursion
- Apply left factoring
- Such grammar can be parsed by RD parser without backtracking ==> *predictive parser*
- Given a current symbol ***a*** and non-terminal *A* to be expanded ($A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$), there shd be a unique production that starts with ***a***.

Example

- *stmt* → **if** *expr* **then** *stmt* **else** *stmt*
  | **while** *expr* **do** *stmt*
  | **for** (*opt_expr; opt_expr;opt_expr*)

TDs for Predictive Parsers

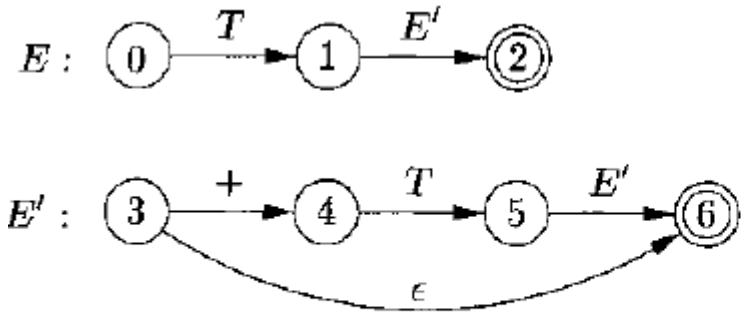
- Unlike LA (one TD for each non-terminal)
- Edges of TD are tokens/non-terminals (NT)
- On a token ==> take a transition if lookahead is a matching token
- On a NT (A) ==> call procedure for A

Construct a TD from Grammar

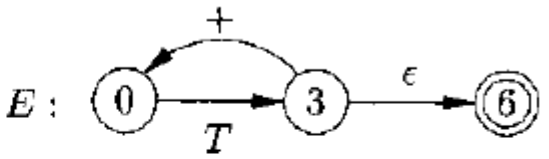
- Step 0: Remove LR, apply left factoring(LF)
- For each NT (A),
  - create an init-state and fin-state
  - For each production $A \rightarrow X_1X_2...X_n$
    - create a path from init-state to fin-state w/ edges $X_1, X_2, ..., X_n$.

- E → E+T | T

  modified to E → TE’ , E’ → +TE’ | ε



(a)



(b)

Working of a Predictive Parser (PP)

- Begin with init-state for start symbol
- If at state *s, and* there is an edge

  -labeled by ***a*** to state *t* and lookahead is ***a*** ==> move to *t,* move ahead i/p cursor

  - edge labeled by NT (A), ==> do not move i/p cursor, go to init-state of A

    - if fin-state of A is reached w/ i/p ==> jump to state *t*

  - labeled by ε ==> no advancing of I/p, jump to state t

Predictive Parsing

- Whenever there is a NT ==> potentially recursive procedure calls
- A non-recursive implementation ==> Apply STACK (push states s for an NT, pop when its fin-state is found)

- Applicable when there is no non-determinism (no two rules for same input)
- When non-determinism inevitable ==> RD with backtrack (No PP).

Example (Grammar)

E → E + T | T

T → T * F | F

F → (E) | **id**

After LR removal

E → TE'

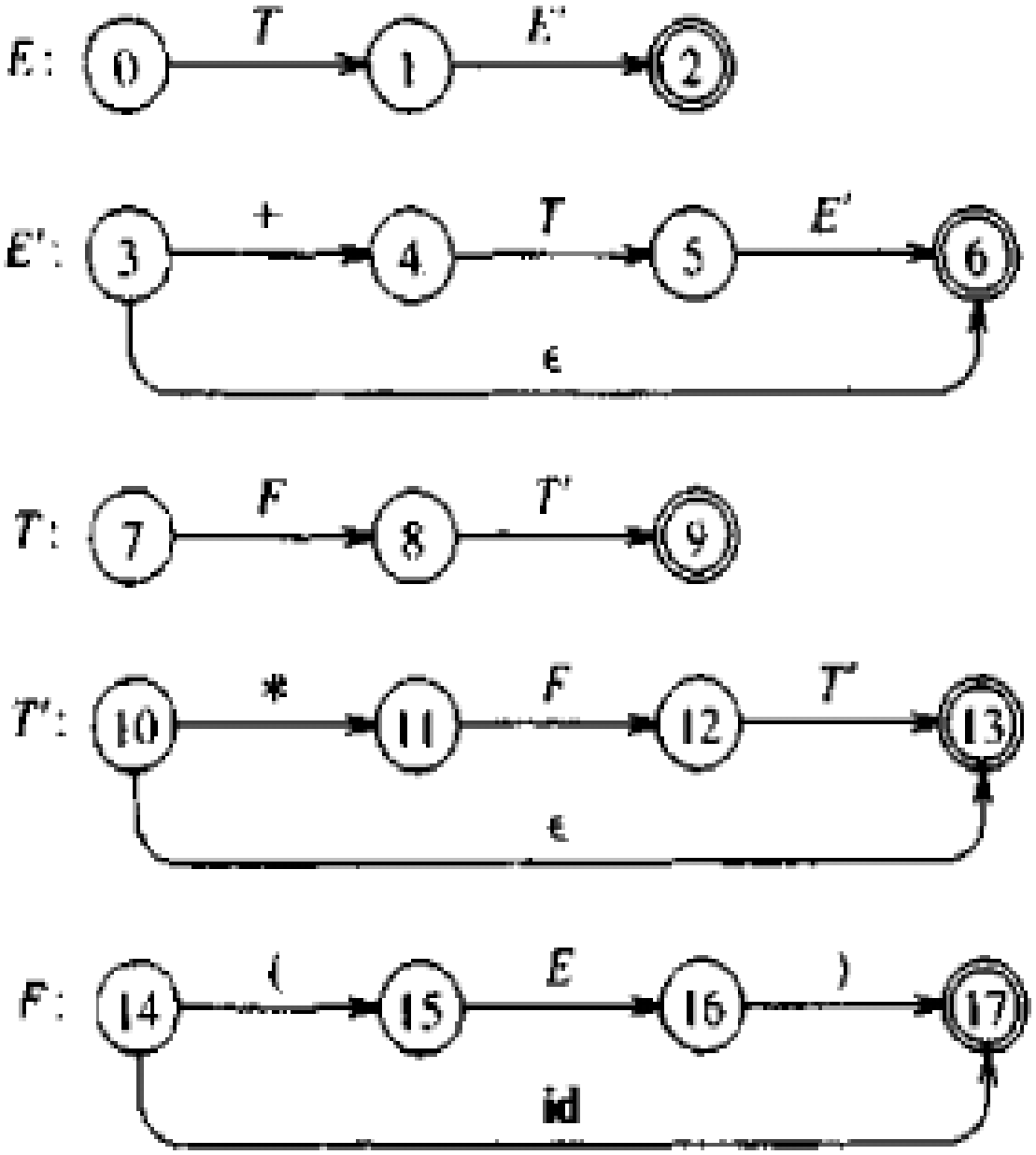
E' → +TE' | ε

T → FT'

T' → *FT' | ε

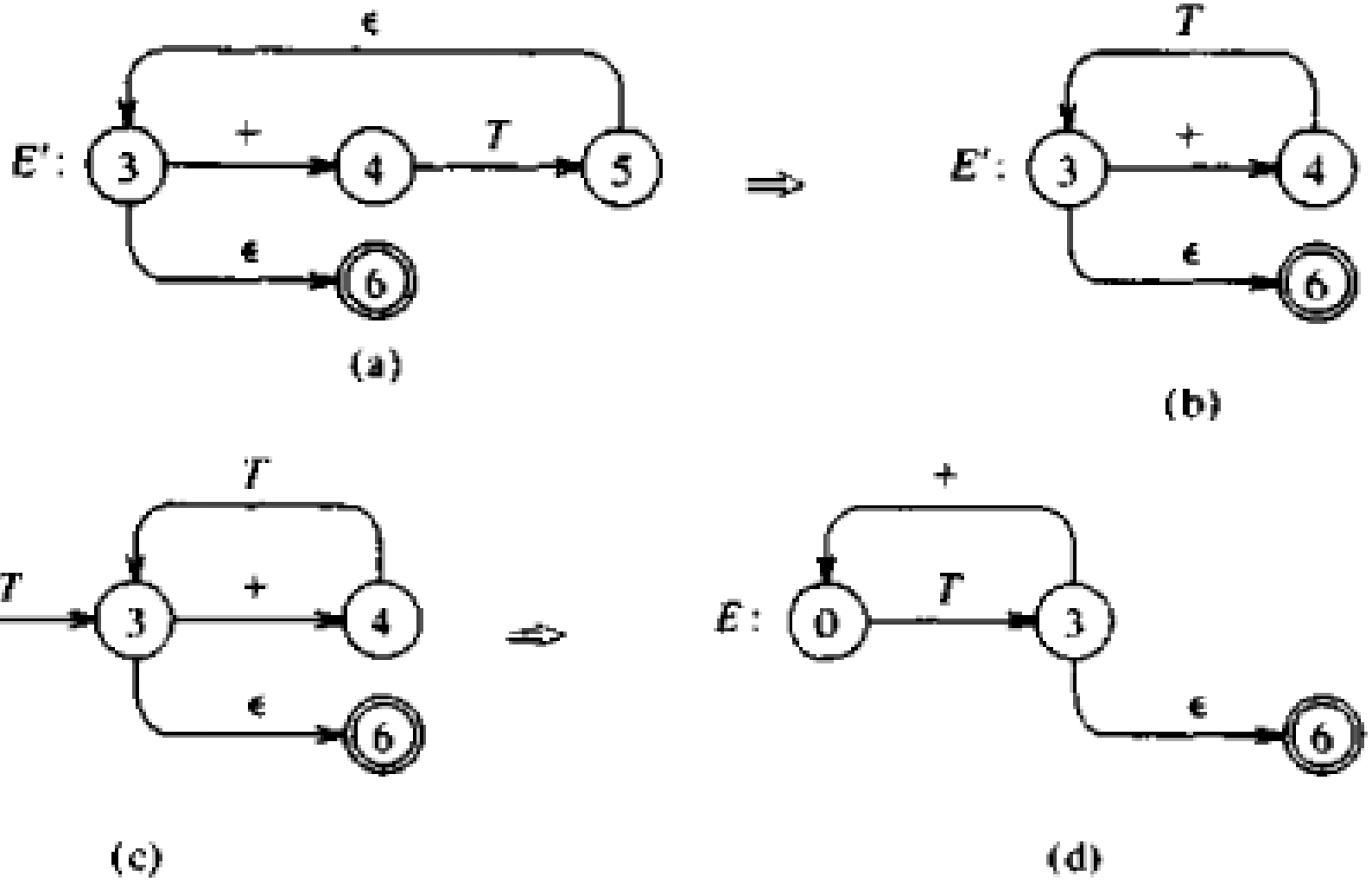
F → (E) | **id**

TDs for modified grammar



$$E \rightarrow TE',$$
$$E' \rightarrow +TE' | \varepsilon$$

Simplified TDs



Simplified TDs

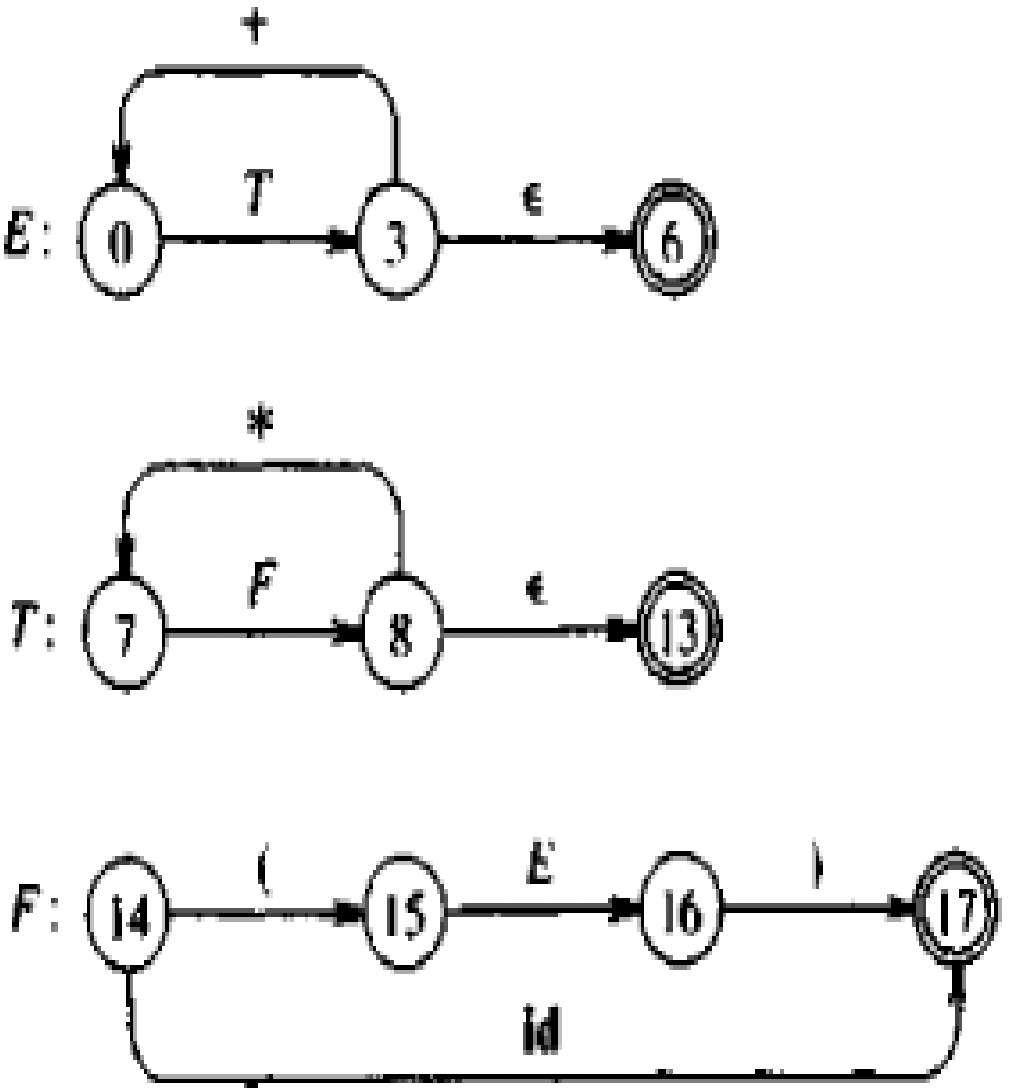
- *E → TE'*

  E' → +TE' | ε

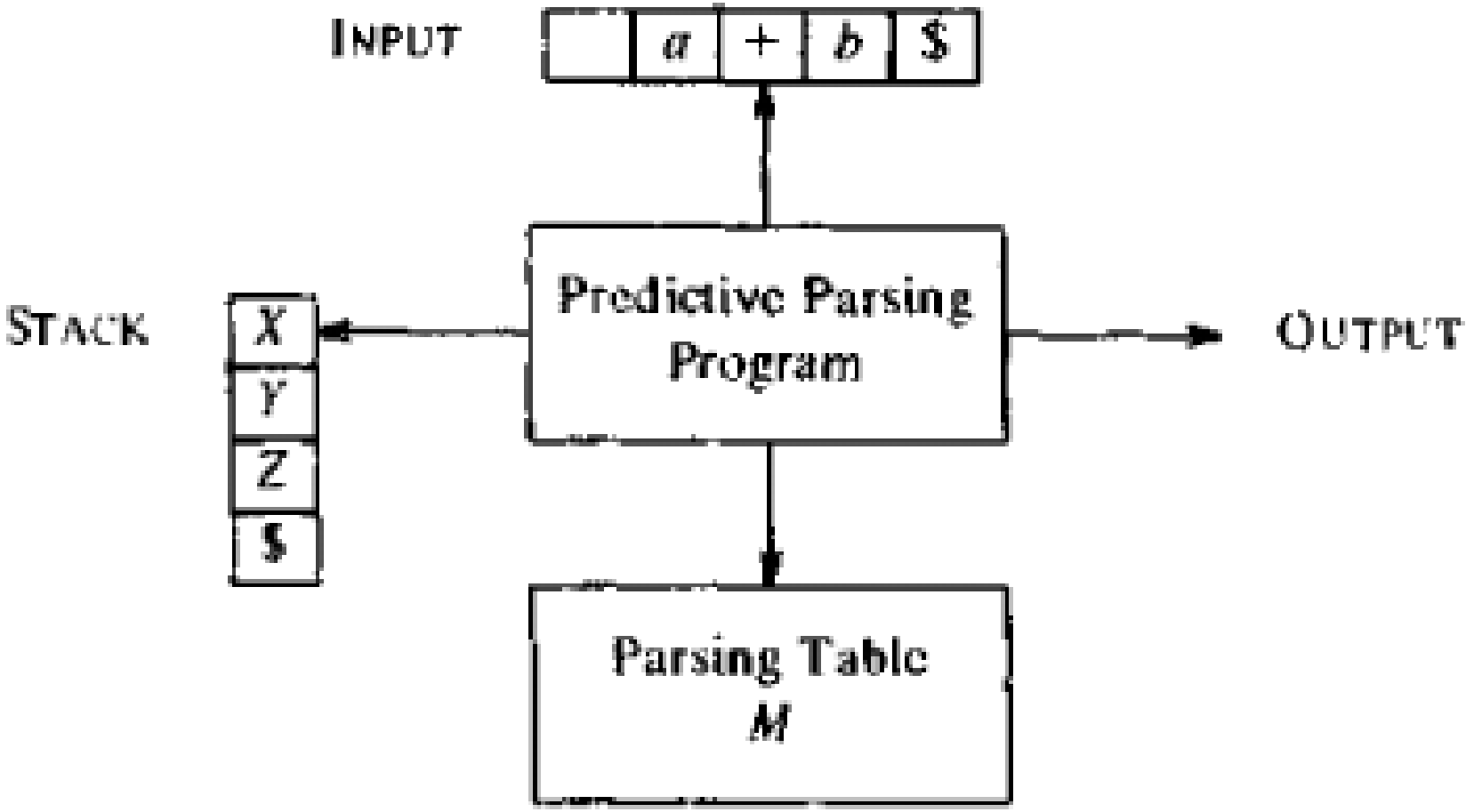
  T → FT'

  T' → *FT' | ε
- F → (E) | **id**

  (coding 20-25% faster)



Non-recursive Predictive Parsing



Non-recursive PP

- An I/P buffer

  - *string to be parsed, (end $)*

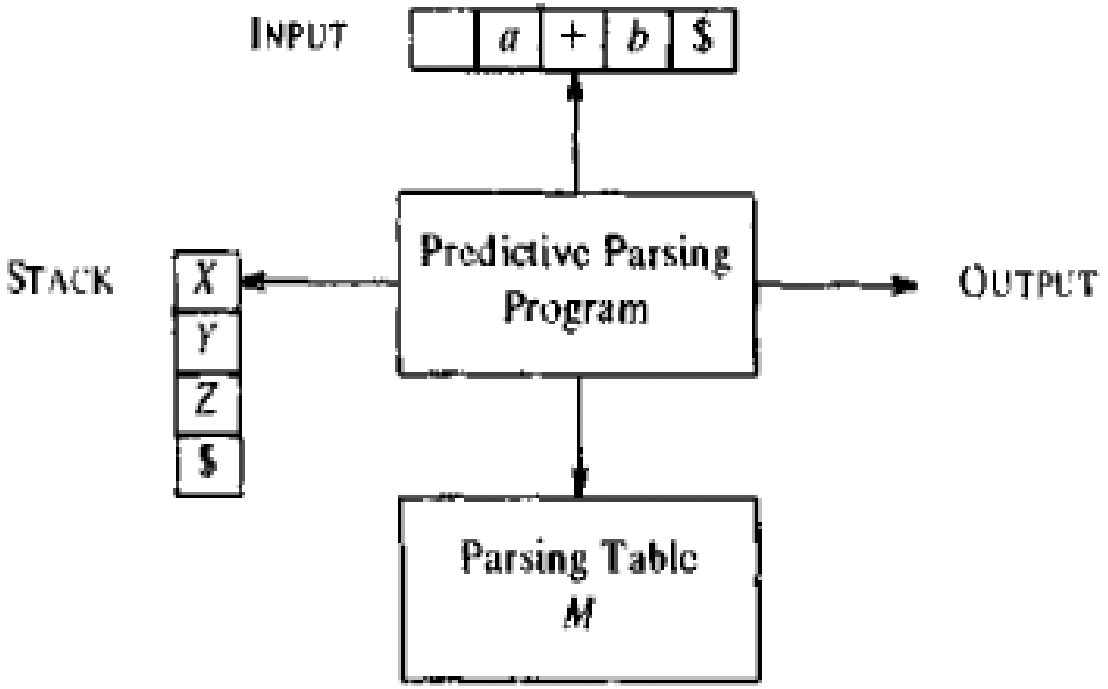
- A stack

  - *grammar symbols (bottom $)*

- A parsing table

  - *2-D array M[A,a]*

- An output stream



Non-recursive PP: Algo

I/P: string *w.* O/P: if w is in L(G), its leftmost derivation, else error

set i/p to point to first symbol of *w$*

repeat

let **X** = top of stack, **a** = symbol pointed to by i/p

if X = a terminal or $ **then**

if *X = a* **then**

pop X from stack and advance i/p pointer

else *error()*

else */* X is a NT */*

if $M[X,a] = X \rightarrow Y_1Y_2...Y_k$ **then begin**

Pop *X* from stack

Push $Y_k, Y_{k-1}, ...Y_1$ on to stack with Y_1 on top of stack

Output the production $X \rightarrow Y_1Y_2...Y_k$

end

else *error()*

until *X = $* */* stack is empty */*

Non-recursive Predictive Parsing

```
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 → Y1 Y2 ··· Yk then begin
            pop X from the stack;
            push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
            output the production X → Y1 Y2 ··· Yk
        end
        else error()
until X = $    /* stack is empty */
```

Example: Parsing Table (M[A,a])

$E \rightarrow TE'$

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | **id**

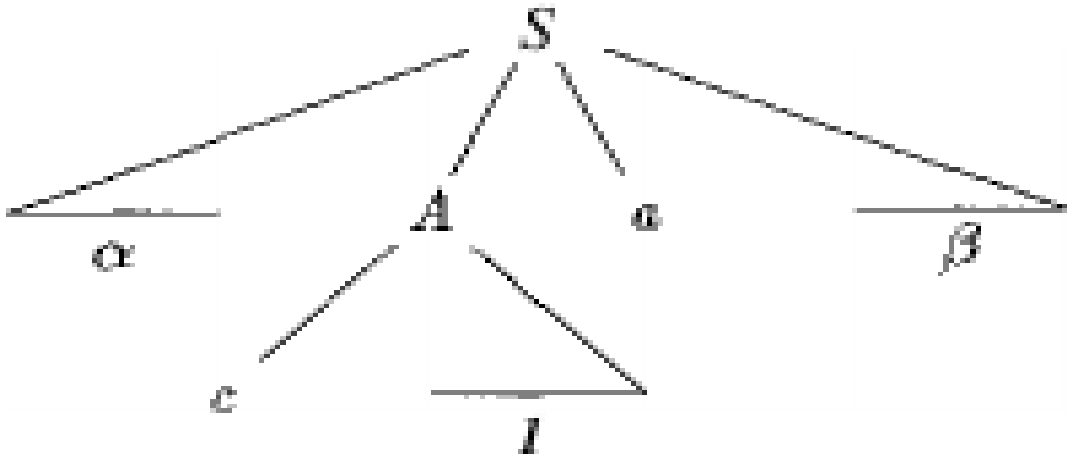
NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Moves for I/P: **id + id * id**

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \rightarrow TE'$
$E'T'F	id + id * id$	$T \rightarrow FT'$
$E'T'id	id + id * id$	$F \rightarrow id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \epsilon$
$E'T+	+ id * id$	$E' \rightarrow +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \rightarrow FT'$
$E'T'id	id * id$	$F \rightarrow id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

FIRST and FOLLOW

- Two functions aid Top-Down and Bottom-Up parsers
- Associated w/ grammar G
- FIRST & FOLLOW help choose a production
- Help fill up entries in parsing table *M[A,a]*
- *FIRST(A) ={c}*
- *FOLLOW(A)= {a}*



FIRST(X)

- 1. If X is terminal, $FIRST(X) = \{X\}$.
- 2. If $X \rightarrow \varepsilon$ is a production, then add ε to $FIRST(X)$.
- 3. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is production, then place a in $FIRST(X)$ if a is in $FIRST(Y_i)$ and

  $FIRST(Y_1) = FIRST(Y_2) = \ldots = FIRST(Y_{i-1})$ contains ε

- For example,
- $FIRST(Y_j) = \{\varepsilon\}$ for all j = 1, 2, . . . , k, then add ε to $FIRST(X)$.
- everything in $FIRST(Y_1)$ is surely in $FIRST(X)$.

FOLLOW(S)

- 1. Place *$* in *FOLLOW(S)*, where S is the start symbol and $ is the input right end-marker.
- 2. If there is a production $A \rightarrow \alpha B \beta$, then everything in *FIRST(β)* except for *ε* is placed in *FOLLOW(B)*,
- 3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where *FIRST(β)* contains *ε,* then everything in *FOLLOW(A)* is in *FOLLOW(B)*.

Example

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

- What are FIRST and FOLLOW sets for the non-terminals?

Soln.

- FIRST(E) = FIRST(T) = FIRST(F) = {**(**, **id** }
- FIRST(E') = {+, ε}
- FIRST (T') = {*,ε}
-
- FOLLOW(E) = {**)**, **$**} = FOLLOW(E')
- FOLLOW(T) = FOLLOW (T') = {**+,),$**}
- FOLLOW(F) = {**+, *,), $**}

Parsing Table Construction

- FIRST and FOLLOW aid in construction
- I/P: Grammar *G,* O/P: Parsing table *M*
- Rules
- 1. For each production **A** → **α** of the grammar, do steps 2 and 3.
- 2. For each terminal **a** in *FIRST(**α**)*, add **A** → **α** to

  M[A, a].
- 3. If ε is in *FIRST(**α**)*, add **A** → **α** to *M[A, b]* for each terminal b in *FOLLOW(A)*,

  (If ε is in *FIRST(**α**)* and $ is in *FOLLOW(A)*, add **A** → **α** to *M[A, $])*
- 4. Make each undefined entry of *M* to be **error**.

Construction of Parsing Table (Example)

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

Example: Parsing Table (M[A,a])

$E \rightarrow TE'$

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | **id**

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Build Parsing Table for another grammar

- $S \rightarrow iEtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $E \rightarrow b$

Parsing Table: multiply-defined

- $S \rightarrow iEtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $E \rightarrow b$

NONTER-MINAL	INPUT SYMBOL					
	a	*b*	*e*	*i*	*t*	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

- *M[S', e]* contains both S' → eS and S' → *ε*
- (As *FOLLOW(S') = {e, $}*)
- The grammar is ambiguous
- Resolve ambiguity by choosing *S′ → eS* when *e* (**else**) is seen
- Associate **else** with closest previous *t* (**then**)

LL(1) Grammar

- A grammar whose parsing table has no multiply-defined entries
- First 'L': scanning I/P L-to-R
- Second 'L': leftmost derivation
- '1': One symbol (terminal or NT) lookahead
- No ambiguity in G
- No left-recursion in G

LL(1) Grammar

- A grammar G is LL(1) *iff* (if and only if) whenever $A \rightarrow \alpha|\beta$ are two distinct productions of G
- 1. For no terminal *a,* both α and β generate strings that starts w/ *a.*
- *2. At most* one of α and β can derive ε
- 3. If β generates ε, then α does not derive any string that starts with a terminal in *FOLLOW(A)*

Which one is LL(1)?

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

OR

- *S* → *iEtSS'* |*a*
- *S'* → *eS* |*ε*
- *E* → *b*
- **USE Predictive Parser for control constructs!**

Error Recovery in PP

- Table-driven predictive parsing
- when terminal at top-of-stack (TOS) does not match lookahead symbol

  - When A (NT) is TOS and ***a*** is next I/P symbol, and *M[A,a]* is **error**

Panic Mode Error Recovery (cf. Aho et al. 4.4)

- Skip symbols in I/P until a selected set of *synchronizing tokens* appears
- Some heuristics

  1. Put all symbols in FOLLOW (A) in a *synch set* for *A*

     - skip until one in FOLLOW (A) seen, pop A

  2. Add keywords that begin stmt for a NT generating expn

  3. Add symbols in FIRST (A) in *synch set*

  4. Adding ε-production and postpone error detection

  5. For a non-matching terminal at TOS, pop the terminal, issue a msg that a matching terminal is inserted

Adding synch tokens

$E \rightarrow TE'$
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | **id**

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

Panic Mode Error Recovery

- Look up entry for *M[A,a]*
- *Blank* – skip *a*
- *Synch* – pop NT (A) to resume parsing
- *Non-matching terminal* – pop token from stack

Example

I/P: **)id * + id**

STACK	INPUT	REMARK
$E	) id * + id $	error, skip)
$E	id * + id $	id is in FIRST(E)
$E'T	id * + id $	
$E'T'F	id * + id $	
$E'T'id	id * + id $	
$E'T'	* + id $	
$E'T'F*	* + id $	
$E'T'F	+ id $	error, M[F, +] = synch
$E'T'	+ id $	F has been popped
$E'	+ id $	
$E'T +	+ id $	
$E'T	id $	
$E'T'F	id $	
$E'T'id	id $	
$E'T'	$	
$E'	$	
$	$	

****Suitable error msg to be generated**

Phrase Level Recovery

- Fill the blank entries w/ pointers to error routines
- The error routines can modify, insert, or delete characters in the I/P and generate error msgs
- They can pop from stack

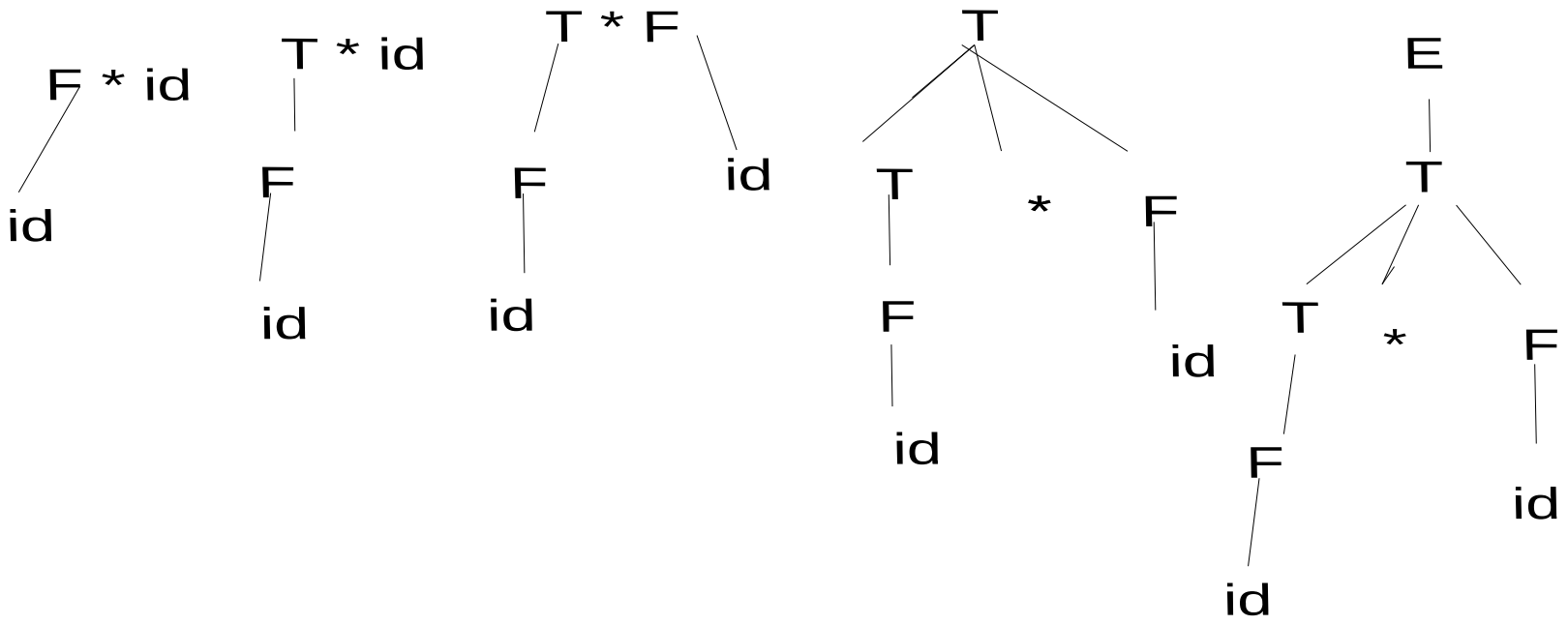
  - Can cause problems
- There shd not be any infinite loop
- CHECK: recovery shd consume I/P symbol or reduce stack-length

BOTTOM-UP Parsing

- Parse tree is generated bottom-up
- General style: *shift-reduce* (SR) parsing
  - LR parsing
  - Operator precedence parsing
- SR scans input from L to R
  - reduce a str *w* to root *S*
  - *a* substr on RHS replaced by LHS NT

Bottom-Up Parsing

- id * id



Example

Grammar

- $S \rightarrow aABe$
- $A \rightarrow Abc \mid b$
- $B \rightarrow d$

String *abbcde*

abbcde => *aAbcde* => *aAde* => *aABe* => *S*

- **Can we try other reductions? And reach S?**

What we did: *rm* derivation

S ==> aABe ==> aAde ==> aAbcde ==> abbcde

All are *rightmost (rm)* derivations

(even w/ *lm* derivation, we get the same string! -agree)

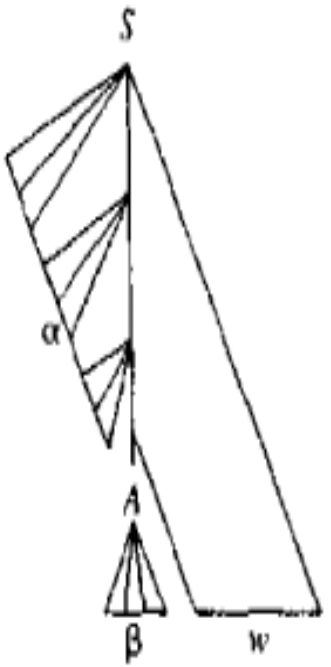
Questions:

- When to reduce?
- what to reduce?

Handles

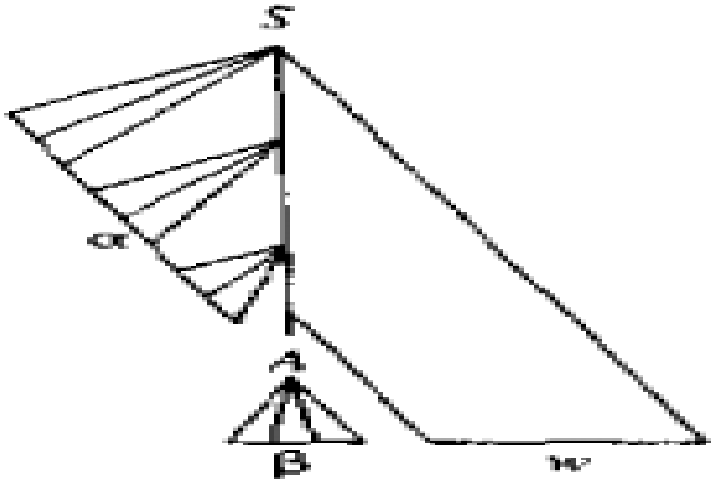
- Informally, a **handle** of a str is a **substr** that matches RHS of a production and whose **reduction** to LHS represents one step along the reverse of a *rm*–derivation
- Often $A \rightarrow \beta$ is not a handle because a reduction always cannot lead to *S.*

  *(*e.g. if *aAbcde* is reduced to *aAAcde* by *applying* $A \rightarrow b$, we can't reach *S)*
-



Handles

- Formally, a **handle** of a (*) right-sentential form *y* is a *production* A → β and a *position* of *y* where β can be replacd by *A* to produce the previous right-sentential form in a rightmost derivation of *y.*



- *S* =*=> *αAw* =*rm*=> *αβw* then **A → β** is “a”
- handle of *αβw after* position *α*
- *“the”* handle, if grammar is unambiguous.

- E.g. *abbcde, A → b* is a handle at position 2
- *aAbcde is a rt*-sentential form whose handle is ***A → Abc*** at pos 2
- (We also call β is a handle of αβw if pos of β and prod A → β clear)

---

*Any sentential form obtained through *rm*-derivation

Example

- E → E + E | E * E|(E)|**id**

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E + E} \\
&\underset{rm}{\Rightarrow} E + \underline{E * E} \\
&\underset{rm}{\Rightarrow} E + \underline{E} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E * E} \\
&\underset{rm}{\Rightarrow} E * \underline{\mathbf{id}_3} \\
&\underset{rm}{\Rightarrow} \underline{E + E} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

Handle Pruning

- Rightmost derivation in reverse obtained by “handle pruning”
- If *w* is the sentence at hand

  $S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \gamma_n = w$

- To reconstruct, we locate β_n in γ_n and replace β_n by A_n (for $A_n \rightarrow \beta_n$) to get (n-1)st rt-sentential form γ_{n-1}
- ….Continuing we get S
- But how handles βn are found?

Example

- I/P: **$id_1 + id_2 * id_3$**

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

SR parsing: Stack Implementation

- **a stack**: hold grammar symbols
- **a buffer**: hold input string *w*

	STACK	INPUT
Init:	$	w$
Final:	$S	$ (success)

else error

SR parser using stack

	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

SR Parser: Operations

- ***Shift:***

  - push next i/p symbol into stack

- ***Reduce:***

  - rt-end of the handle is at TOS, locate the left-end, and decide NT to replace handle

- *Accept:*

  - notify successful completion of parsing

- *Error:*

  - discovers syntax error and calls error for error-recovery

Why stack?

(1) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Az \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta Byz \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta\gamma yz$

(2) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha BxAz \overset{*}{\underset{rm}{\Rightarrow}} \alpha Bxyz \overset{*}{\underset{rm}{\Rightarrow}} \alpha\gamma xyz$

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$

STACK	INPUT
$\$\alpha\beta B$	$yz\$$

STACK	INPUT
$\$\alpha\gamma$	$xyz\$$

STACK	INPUT
$\$\alpha Bxy$	$z\$$

Conflicts in Shift/Reduce Parsing

- Shift-reduce conflict
- Reduce/reduce Conflict

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

- Using SR parser (Shift-reduce)
-

STACK	INPUT
· · · **if** *expr* **then** *stmt*	**else** · · · $

R-R Conflicts in SR parsing

- A statement beginning with *A(I,J)*

(1) *stmt* → **id** (*parameter_list*)
(2) *stmt* → *expr* := *expr*
(3) *parameter_list* → *parameter_list* , *parameter*
(4) *parameter_list* → *parameter*
(5) *parameter* → **id**
(6) *expr* → **id** (*expr_list*)
(7) *expr* → **id**
(8) *expr_list* → *expr_list* , *expr*
(9) *expr_list* → *expr*

STACK	INPUT
· · · **id** (**id**	, **id**) · · ·

Operator Precedence Parsing

- A small but important class of grammar w/ following 2 properties

  - No ε on RHS

  - No two adjacent NT on RHS

- The above is called **Operator Grammar**

  $E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$

  A → + | - |* | / |↑ (Is it Op-Grammar?)

Operator Precedence Parsing

- But the following is Operator Grammar
- $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$

- Advantages
  - simplicity
- Problems
  - unary operators (NOT considered now)
  - only a small class of grammars

Precedence relations

- Relations b/w a pair of terminals
- Based on associativity and traditional notions

RELATION	MEANING
$a \lessdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \gtrdot b$	a "takes precedence over" b

Using Op-Prec relations

- Delimit the handle of a rt sential form
- $<_o$ on the left
- = in the interior
- $_o>$ on the right

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

- E.g. **id + id * id**
- **$ < . id . > + < . id . > * < . id . > $**

Procedure

- 1. Scan from L-to-R until first **. >**
- 2. Scan back from there until a **< .**
- 3. Handle contains every thing

  to left of first **. >** and to the right of **.<** in Step 2

  (incl any intervening or surrounding NT)

  Acc to the rule ==> **E + id * id** =rm=> **E + E * E**

- New expression is

  $ < . + < . * . > $

- The handle is **E * E**

Op-Prec Parsing Algo

- I/P: string *w*

```
(1)  set ip to point to the first symbol of w$;
(2)  repeat forever
(3)       if $ is on top of the stack and ip points to $ then
(4)            return
           else begin
(5)            let a be the topmost terminal symbol on the stack
                    and let b be the symbol pointed to by ip;
(6)            if a <· b or a ≐ b then begin
(7)                 push b onto the stack;
(8)                 advance ip to the next input symbol;
               end;
(9)            else if a ·> b then          /* reduce */
(10)                repeat
(11)                     pop the stack
(12)                until the top stack terminal is related by <·
                         to the terminal most recently popped
(13)           else error()
          end
```

Creating table

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

General rules for Operator Table

1. If **a** has higher prec than **b** then

   a . > b and **b <. a**

2. If **a** and **b** have equal precedence then

   a . > b and **b . > a** (both **a** and **b** are left associative)

   else

   a . < b and **b .< a** (both **a** and **b** are right associative)

3. Any terminal **a**, **a .> $**

4. **a <. id, id .> a, a <. (, (<. a,) . > a, a .>),**

   a . > $, $ <. a for any operator **a.**

Example

- **id * (id ↑ id) – id / id**

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Handling Unary Operators

Case I: ! (logical NOT) unary but not binary

- for any other operator, **a < . !** (if **!** is unary, **a** unary or binary)
- **! . > a** if **!** has higher precedence over **a**

  else

  ! <. a (Try with *E & !E & E* for *! . > &* and *& lt ass)*

  Case II: - (minus) unary (prefix) and binary (infix)

- when both unary and binary same prec. (try **id *-id**)

  Soln: differentiate b/w unary and binary **minus**

  (FORTRAN, minus unary if prev token: operator, '(',',', assign)

Op-Prec: Error Recovery

Error occurs when

1. **no precedence** reln b/w TOS term and current i/p

2. Handle found, but **no production** matching RHS of a grammar rule

Are the above two rules enough?

Handling Errors during Reductions

Error type 2 (no production) ((cf. Aho et al Sec 4.6)

- Closest production that “looks like”

  - error msg: `illegal <x> on line <yy>`

  - `missing <x> on line <yy>`

- When ops (+, -, *, /, ↑) are reduced, NT shd be on either sides

  - else error msg: `missing operand`

- Similarly for **id:** `missing operator`

- **( ):** `no expression between parentheses`

Handling Shift-Reduce errors

- Blank entry in the precedence matrix

  Soln.

- Either modify stack, or i/p or both
- Check there is no infinite loop
- For each blank entry, design error routine

Example

	id	(	)	$
id	e3	e3	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	e4
)	e3	e3	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	e2	e1

e1: /* called when whole expression is missing */
insert **id** onto the input
issue diagnostic: "`missing operand`"

e2: /* called when expression begins with a right parenthesis */
delete) from the input
issue diagnostic: "`unbalanced right parenthesis`"

e3: /* called when **id** or) is followed by **id** or (*/
insert + onto the input
issue diagnostic: "`missing operator`"

e4: /* called when expression ends with a left parenthesis */
pop (from the stack
issue diagnostic: "`missing right parenthesis`"

LR Parsers

- Most prevalent bottom-up parser *LR(k)*
- '*L*': Left-to-right scanning of i/p
- '*R*': rightmost derivation in reverse
- *k*: no. of i/p symbols used as lookahead
- *k*=0, 1 are of practical interest
- If *k* is omitted, k is assumed 1

Why LR Parsers?

- Table-driven like non-recursive LL parsers
- Can be constructed for any programming language constructs following a CFG
- Most general non-backtracking *Shift-Reduce* method
- Can detect syntactic errors early
- Class of grammars that can be handled in LR methods is superset of those in LL methods

  DRAWBACK: tedious! (automatic tool **Yacc**)

SLR (Simple LR): Some Definitions

- Problem: *when to shift? when to reduce?*

- *LR* parser decides based on *states* (used to remember where we are in a parse)
- States represent sets of “items”
- An *LR(0) item* of a grammar G is a production w/ a dot at some position

  For $A \rightarrow XYZ$, following are the items

  $A \rightarrow . XYZ$

  $A \rightarrow X . YZ$

  $A \rightarrow XY . Z$

  $A \rightarrow XYZ .$

  Production $A \rightarrow \varepsilon$ generates only one item $A \rightarrow .$

  (Intuitively, item indicates how much of a production we have seen at present)

Canonical LR(0)

- One collection of sets of LR(0) items, called *Canonical LR(0)* collection provides the basis of constructing DFA to help decide *shift* or *reduce*
- Automaton is called *LR(0) automaton*
- Each state of LR(0) automaton represents a set of items in canonical LR(0) collection
- To construct a canonical LR(0) collection we need an augmented grammar G' and

  two functions

  - CLOSURE

    (collection of equivalent items)

  - GOTO

    (transition)

LR(0) collection: Augmented Gr.

- For a grammar G with start S, the augmented grammar G' w/ start S' is

  S' → S

- Acceptance of a string when we can reduce

  S' → S.

CLOSURE of Item Sets

- If *I* is the set of items for *G*, then *CLOSURE(I)* is constructed from *I* using following rules

  1. add every item in *I* to *CLOSURE (I)*

  2. If **A → α . Bβ** is in *CLOSURE (I)* and **B → γ** is a production then add item **B → . γ** in *CLOSURE(I).*

Apply Rule 2 until there is no more item.

CLOSURE: Example

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

If I is the set of one item $\{[E' \rightarrow . E]\}$,

CLOSURE(I) = I_0

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

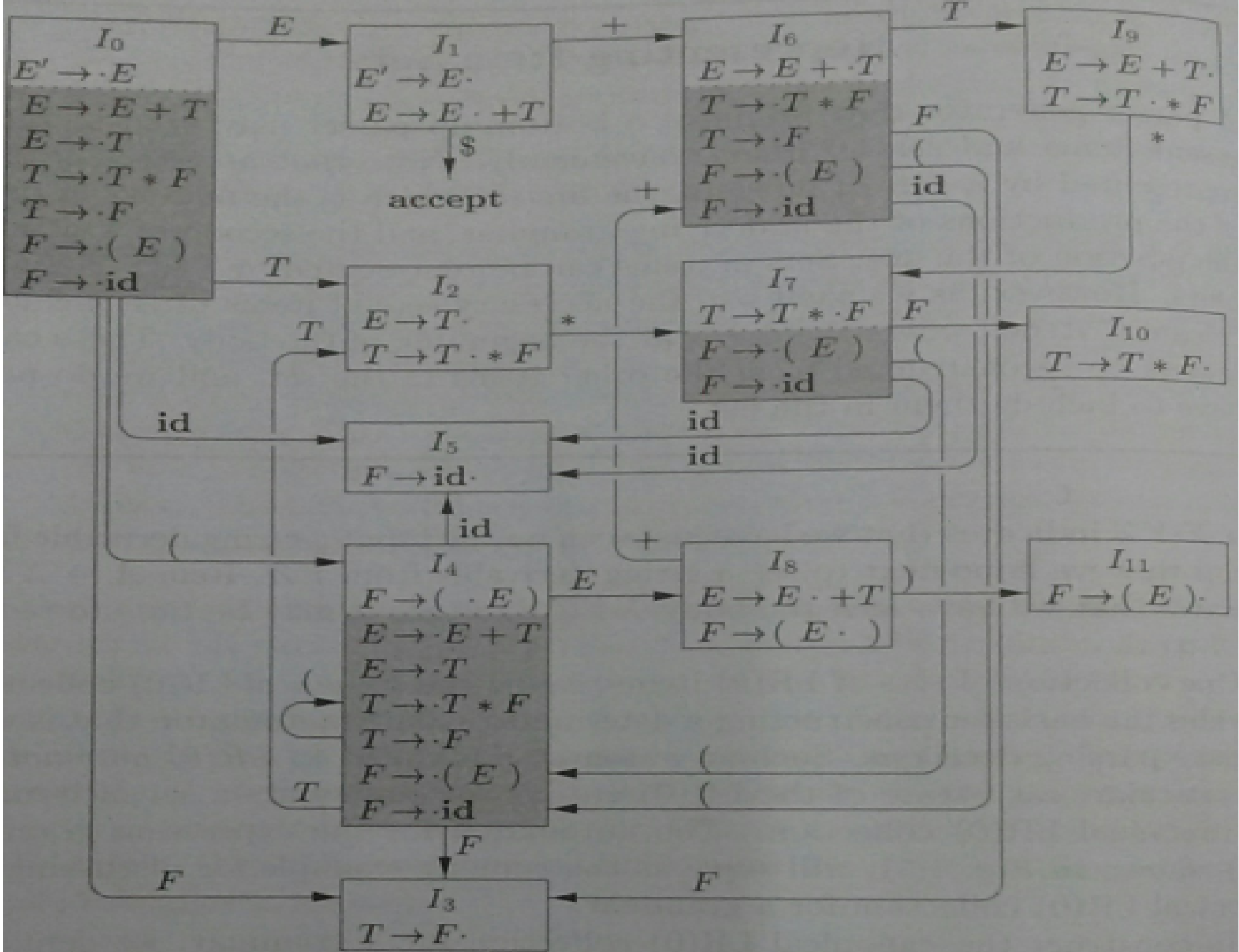
I_7: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

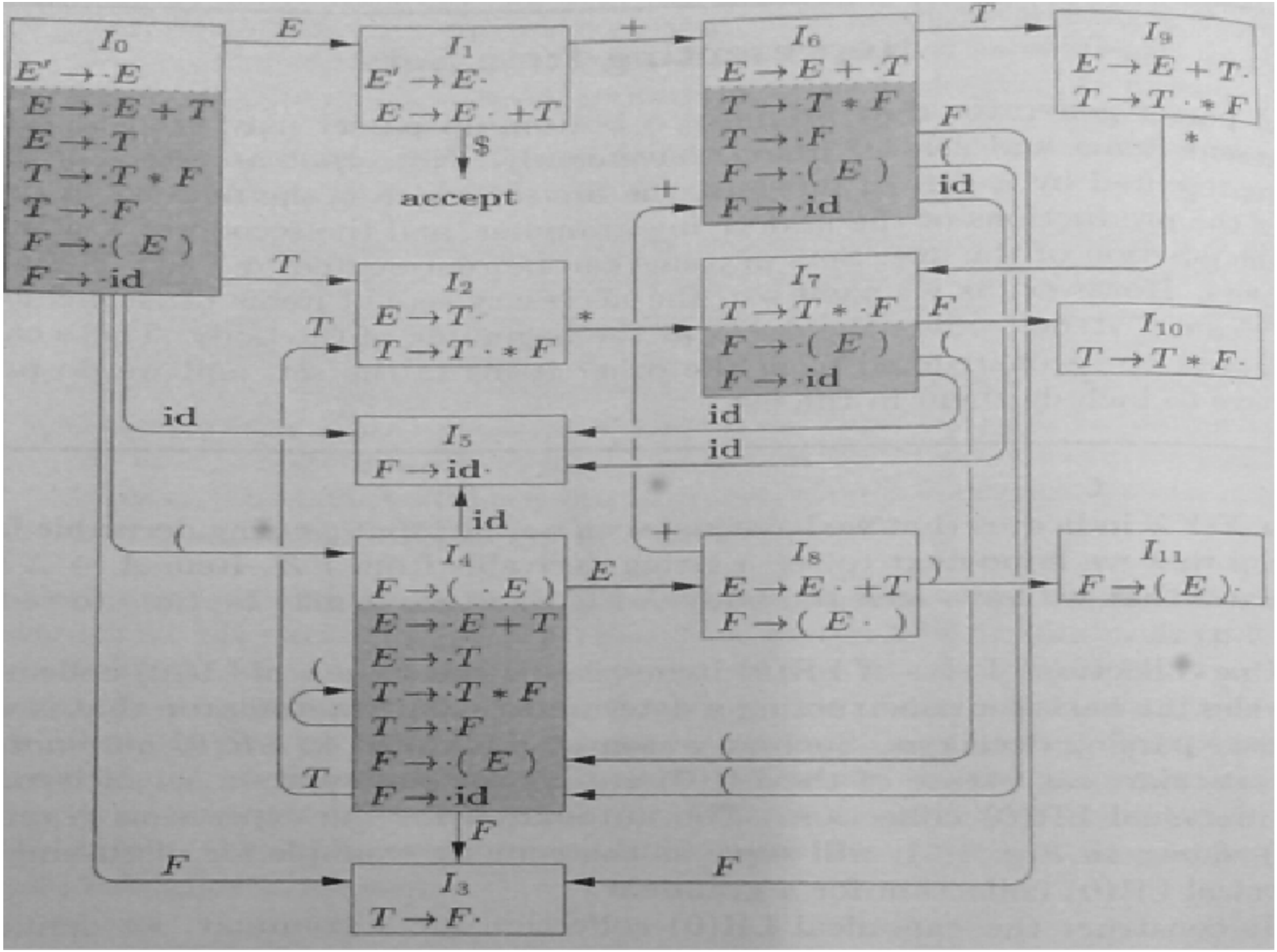
I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
accept
E
T
F
$+$
$*$
$($
$)$
$\mathbf{id}$
$\$$


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
accept
E
T
F
$+$
$*$
$($
$)$
$\mathbf{id}$
$\$$

CLOSURE (I): Algo

SetOfItems CLOSURE(I) {

J = I;

repeat

for (each item $A \rightarrow \alpha . B \beta$ in J)

for (each production $B \rightarrow \gamma$ of G)

if ($B \rightarrow . \gamma$ is not in J)

add $B \rightarrow . \gamma$ to J ;

until no more items are added to J on one round;

return J;

}

Kernel and Non-kernel items

- *Kernel items:* the initial item, S' -> **.** S, and all items whose dots are not at the left end.

- *Non-kernel items*: all items with their dots at the left end, except S' → **.** S

- Pruning: for minimum storage, throw away non-kernel items! (they can be generated by CLOSURE)

GOTO

GOTO (I, X): transitions in LR(0) automaton

I: set of items, *X*: grammar symbol

- *GOTO (I, X) = CLOSURE* of all items [$A \rightarrow \alpha X \, . \, B$]
  s. t. [$A \rightarrow \alpha . \, XB$] is in *I.*

states: sets of items (*I*);
GOTO(I, X): transitions from the state for *I* on seeing *X.*

Example

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

- If $I = \{[E' \rightarrow E \,.\,], [E' \rightarrow E \,.\, + T]\}$, $GOTO(I, +) = ?$

Soln.

$E \rightarrow E + \mathbf{.}\ T$

$T \rightarrow .\ T * F$

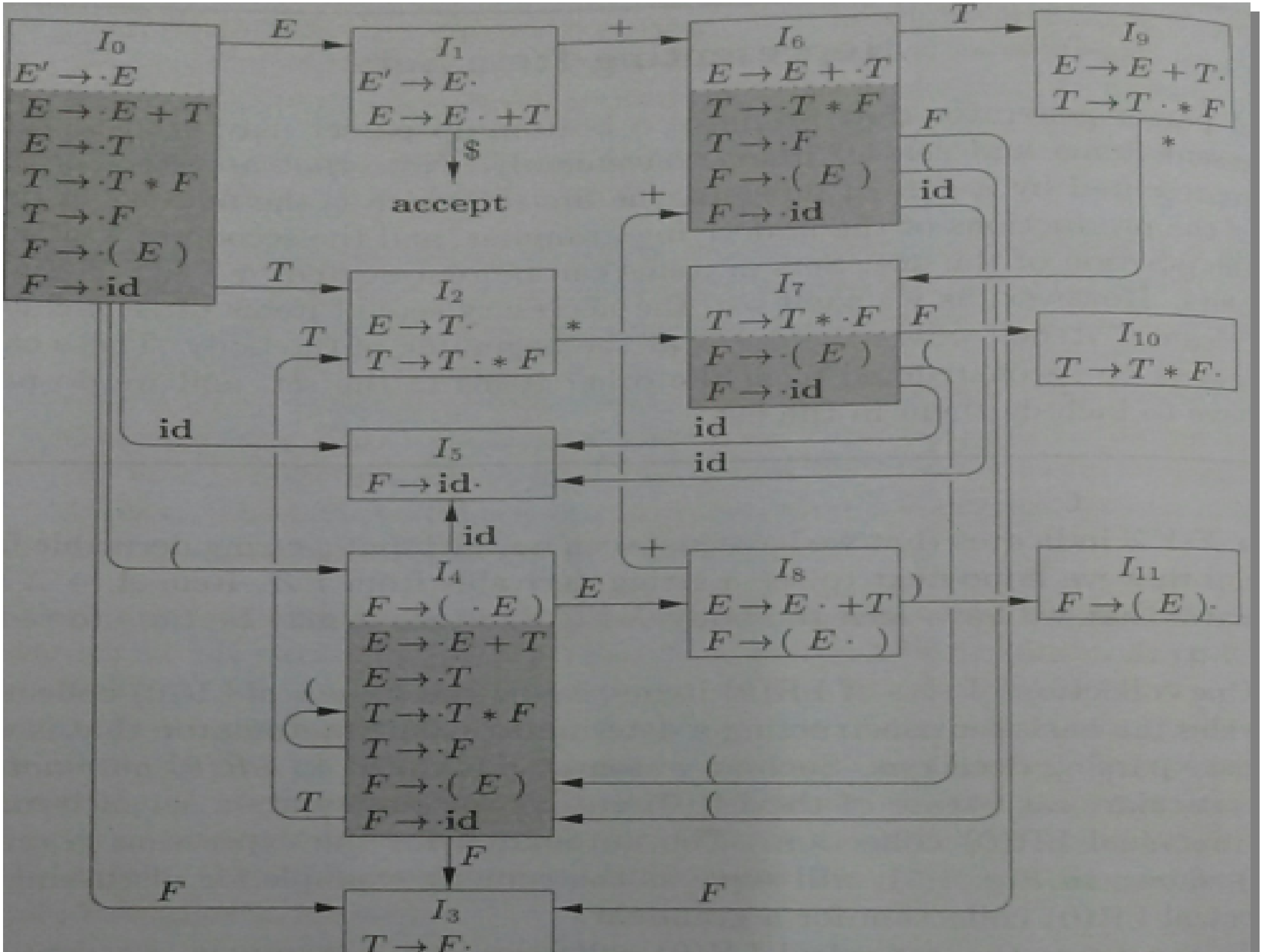
$T \rightarrow \mathbf{.}\ F$

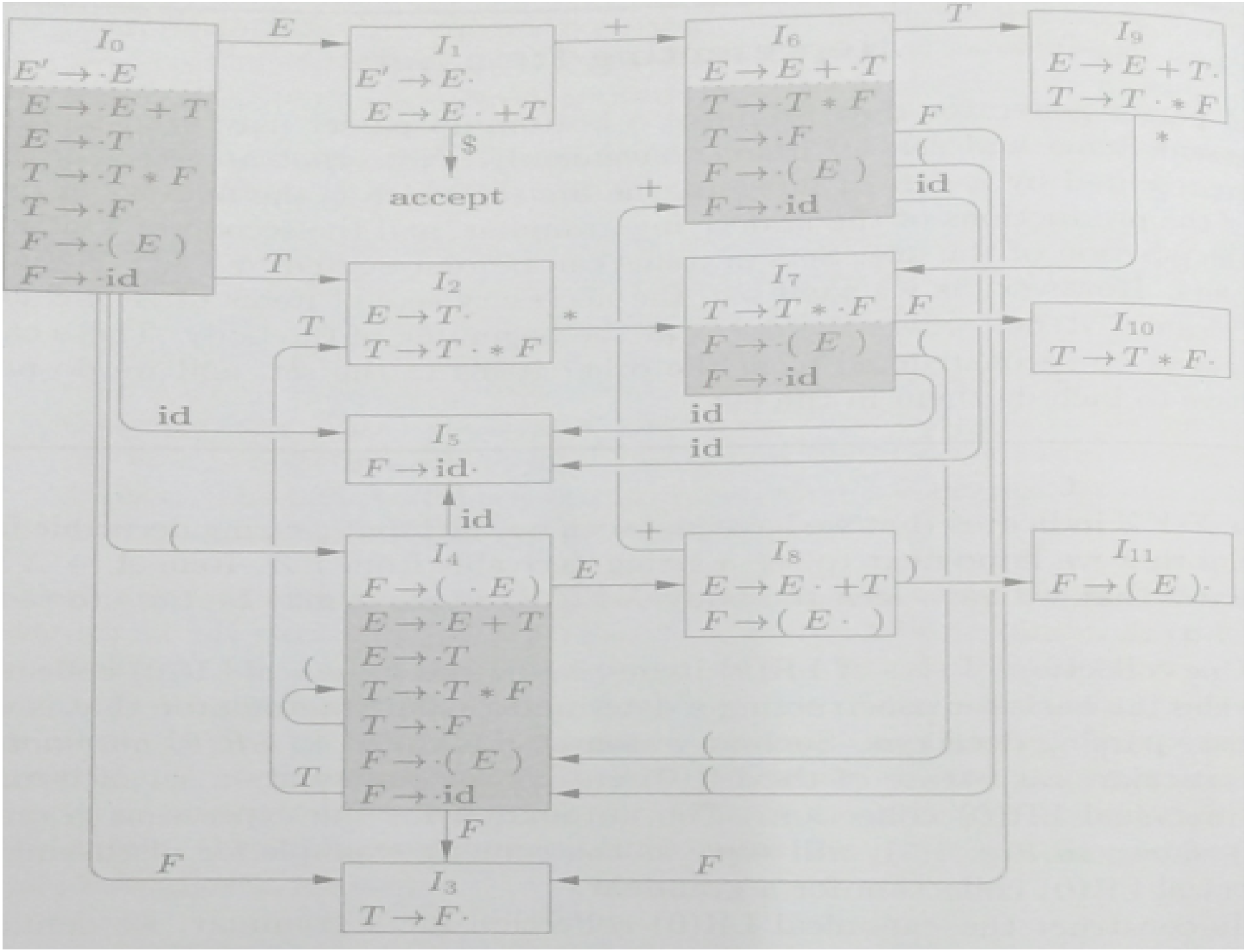
$F \rightarrow \mathbf{.}\ (E)$

$F \rightarrow \mathbf{.}\ \mathbf{id}$

Canonical collection of 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 sets of items are added to C on a
    round;
}
```

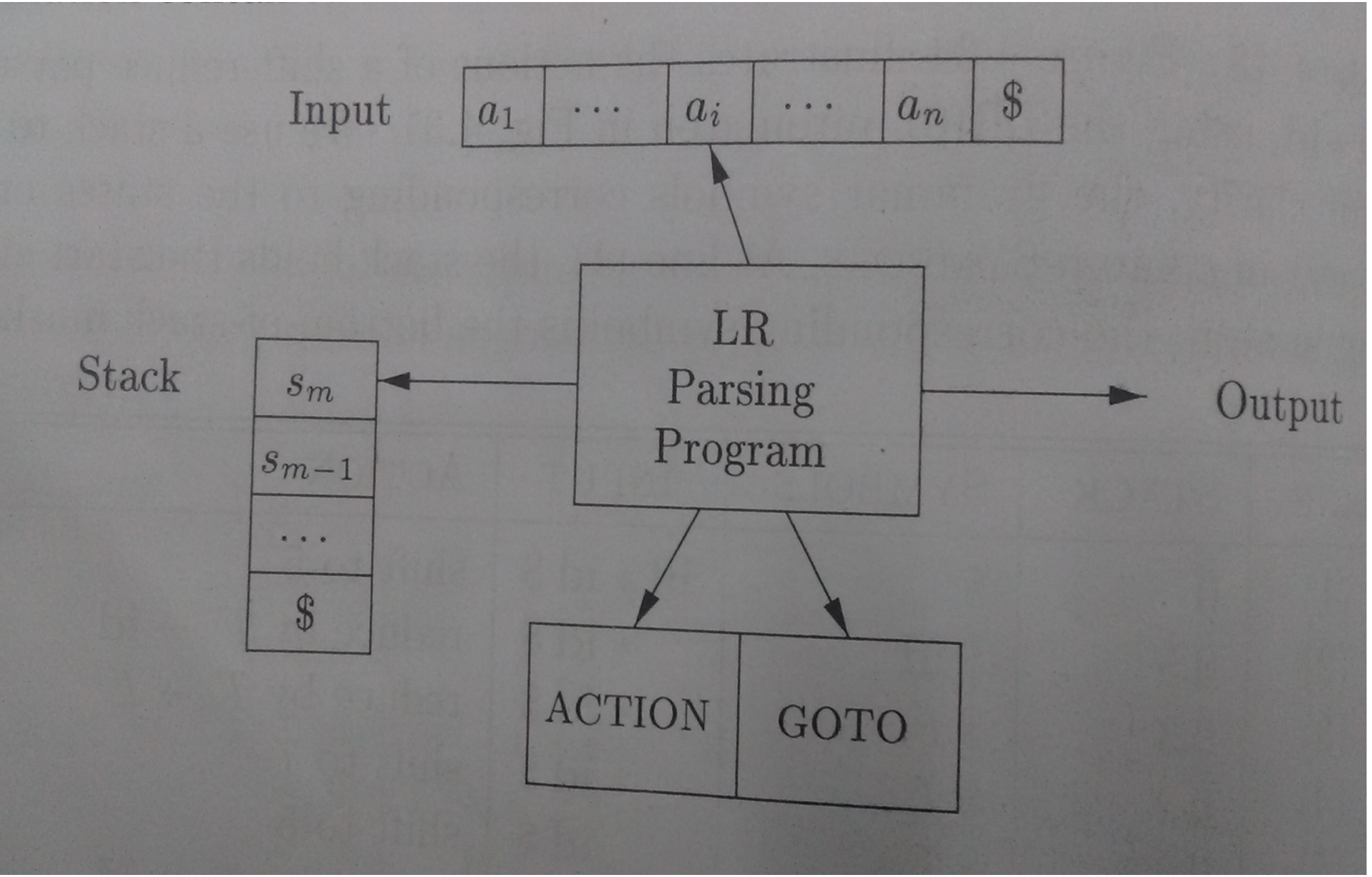

I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
accept
E
T
F
$+$
$*$
$($
$)$
$\mathbf{id}$
$\$$


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
accept
E
T
F
$+$
$*$
$($
$)$
$\mathbf{id}$
$\$$

Use of LR(0) Automaton

- Suppose string of grammar symbols take LR(0) automaton from state 0 to some state *j*
- **Shift** on next i/p symbol *a* if state *j* has a transition on *a*
- **Reduce** based on production at state *j*, otherwise

LR Parsing


Input
a_1
$\cdots$
a_i
$\cdots$
a_n
$
Stack
s_m
s_{m-1}
$\cdots$
$
LR
Parsing
Program
Output
ACTION
GOTO

LR Parsing

- I/P: A string *w* followed by $
- A stack of *states* (unlike symbols in Shift-Reduce)
- A driver program
- Parsing Table

  - Action

  - GOTO

  (states: set of items)

Structure of LR Parsing Table

- ACTION (state *i*, input symbol *a)*
  - **shift** *j (*to stack, also *a*)
  - **reduce** A → β (pop β from stack and push A)
  - **accept** (entire w is accepted & parsing ends)
  - **error** (parser encounters some error)
- GOTO extended definition *wrt* states and symbol A
  - GOTO (I_i, *A)* = I_j

LR Parser Configurations

- Complete state of the parser

  $(s_0 s_1 \ldots S_m, a_i a_{i+1} \ldots . a_n \$)$

  (s_i: states, a_j: remaining i/p)

- Similar to $(X_1 X_2 \ldots . X_m \; a_i a_{i+1} \ldots . a_n)$ obtained in SR parser (X_i: Grammar symbols)

Behaviour of LR Parser

- ACTION[s_m, a_i] = shift *s,*

  ($s_0 s_1 \ldots s_m s$, $a_{i+1} \ldots . a_n \$$)

- ACTION[s_m, a_i] = reduce A → β,

  ($s_0 s_1 \ldots s_{m-r} s$, $a_i a_{i+1} \ldots . a_n \$$) where

  r = length of β, s = GOTO[s_{m-r}, A]

- ACTION[s_m, a_i] = accepted, parsing complete
- ACTION[s_m, a_i] = error, calls error routine

LR Parsing Algo

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;

        move to the next input symbol;

    } **else if** (ACTION [*s, a*] = **reduce** A → β) {

        pop |β| symbols off 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: **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)	02 710	$T*F	**$**	reduce by T —> T * *F*
(8)	02	$T	**$**	reduce by *E* —> *T*
(9)	01	*$E*	**$**	accept

Consider this …

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

Parsing Table

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			

Example: id*id + id

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id * id + id $**	shift
(2)	0 5	**id**	*** id + id $**	reduce by $F \rightarrow$ **id**
(3)	0 3	F	*** id + id $**	reduce by $T \rightarrow F$
(4)	0 2	T	*** id + id $**	shift
(5)	0 2 7	T *	**id + id$**	shift
(6)	0 2 7 5	T * **id**	**+ id$**	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	T * F	**+ id$**	reduce by $T \rightarrow T * F$
(8)	0 2	T	**+ id$**	reduce by $E \rightarrow T$
(9)	0 1	E	**+ id$**	shift
(10)	0 1 6	E +	**id$**	shift
(11)	0 1 6 5	E + **id**	**$**	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E + F$	**$**	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	**$**	reduce by $E \rightarrow E + T$
(14)	0 1	E	**$**	accept

Constructing SLR-Parsing Tables

1. Build $C = \{I_0, I_1, \ldots, I_n\}$, collection of LR(0) itemsets for G'

2. State i is constructed from I_i. Determine parsing actions for i:

(a) If [A → α. a β] is in I_i and GOTO (I_i , a) = I_j, then set ACTlON[*i, a*] to "**shift j**" Here ***a*** must be a terminal

(b) If [A → α.] is in I_i, then set ACTION[i,a] to

"**reduce A → α**" for all ***a*** in FOLLOW(A); here A may not be S'

(c) If [S' → S.] is in I_i then set ACTION[i,$] to "**accept**"

3. GOTO for state *i* constructed for all nonterminals A using

if GOTO(I_i,A) = Ij, then GOTO[i, A] = j.

4. All entries not defined by rules (2) and (3) are "**error**"

5. initial state constructed from set of items containing [S' → • S]

Example: Build a SLR parsing Table

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_7: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$

Parsing Table

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			