LR Parsers

All grammars are not LL(1)

	a	c	d	$
X	X → a X → Y	X → Y	X → Y	
Y	Y → ε	Y → c Y → ε	Y → ε	
Z	Z→XYZ	Z→XYZ	Z → d Z→XYZ	Z→XYZ

Z → d
Z → XYZ
Y → ε
Y → c
X → Y
X → a

FIRST (X) = {a, c, ε}
FIRST (Y) = {c, ε}}
FIRST (Z) = {d, a, c}
FOLLOW(X) = {c, d, a}
FOLLOW(Y) = {a, c, d}
FOLLOW(Z) = {$}

All grammars are not LL(1)

A grammar is an LL(1) grammar if all productions conform to the following conditions:

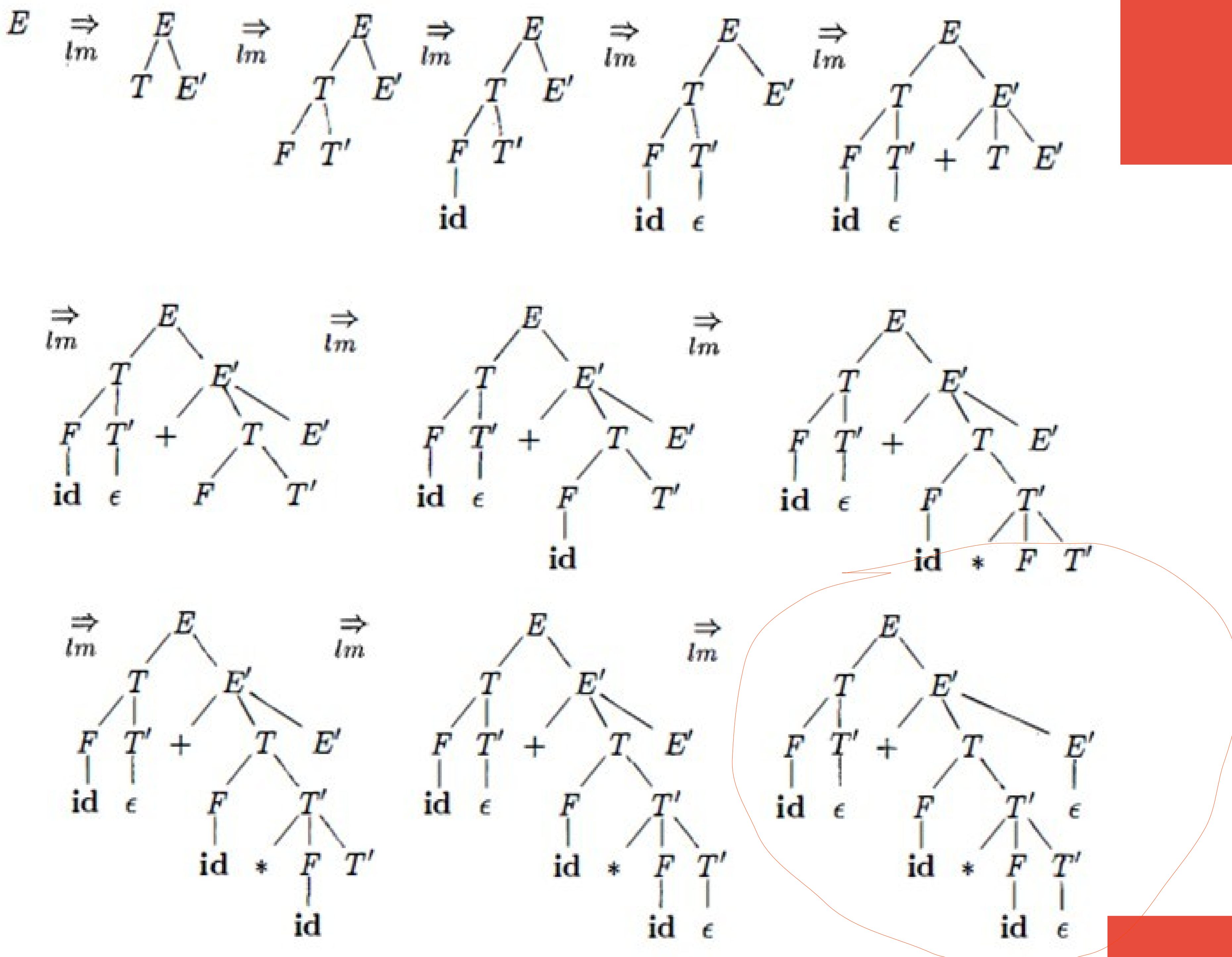
1. For each production $A \rightarrow \sigma 1 \mid \sigma 2 \mid \sigma 3 \ldots \mid \sigma n$,

$\text{FIRST}(\sigma_i) \cap \text{FIRST}(\sigma_j) = \emptyset$, for all i, j, i <> j

2. If nonterminal X derives an empty string, then

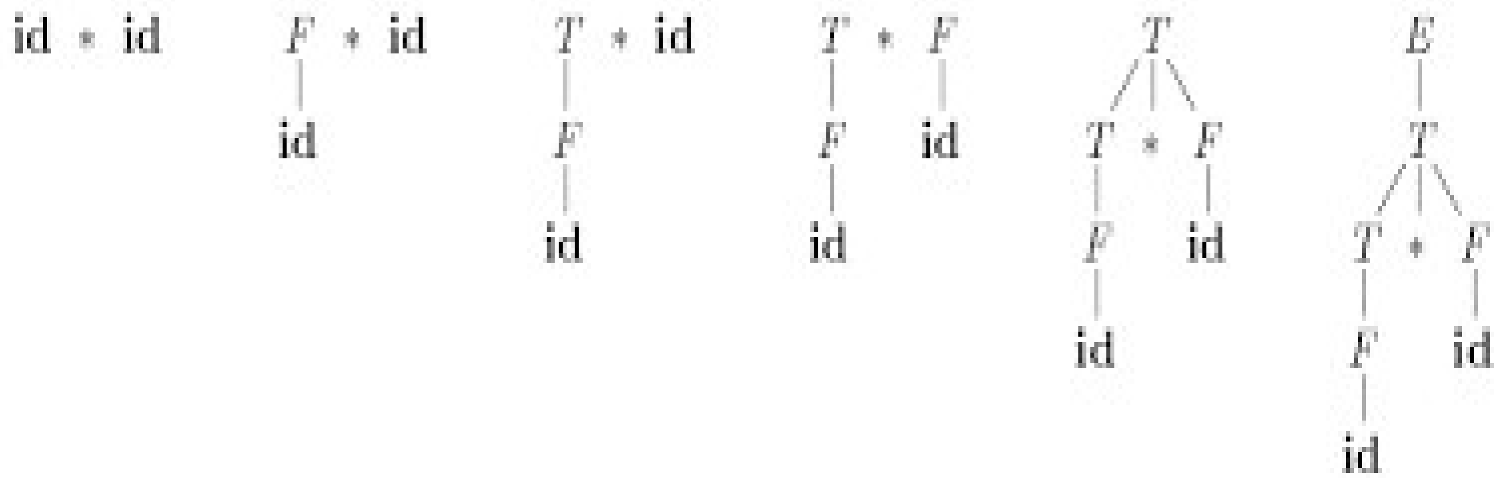
$\text{FIRST}(X) \cap \text{FOLLOW}(X) = \emptyset$

TOP DOWN PARSING



Bottom up parsing

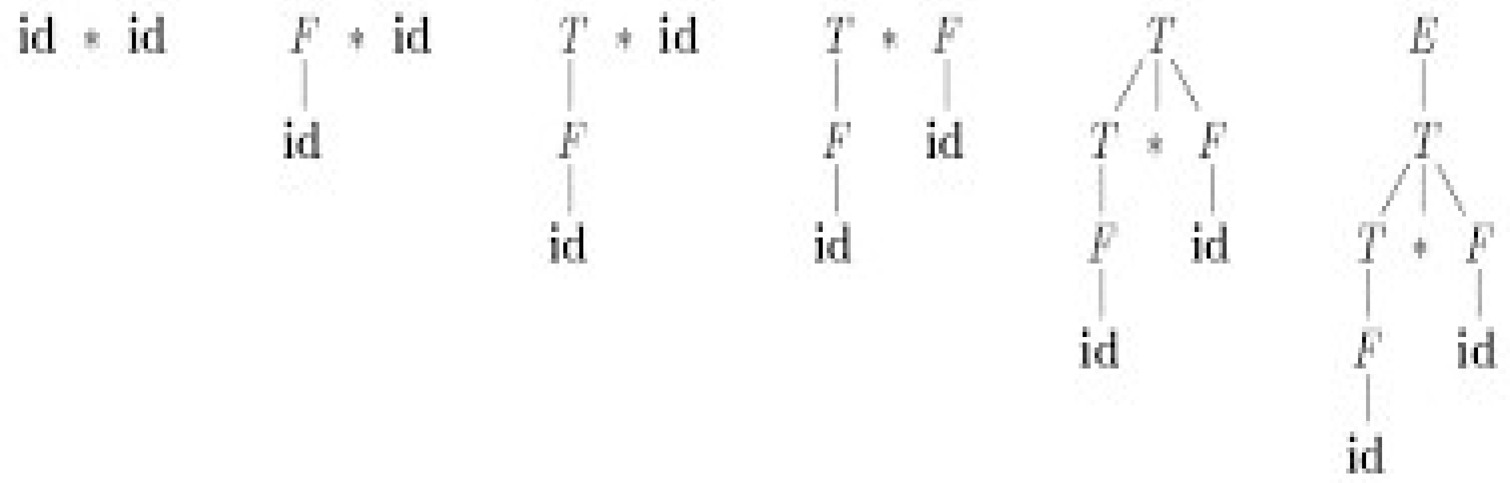
E → E + T | T
T → T * F | F
F → (E) | id



E => T => T * F => T * id => F * id => id * id

Bottom up parsing

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



E => T => T * F => T * id => F * id => id * id

Bottom up parsing

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

(The string, together with its position in the right sentential form where it occurs and the production used to reduce it).

Reducing β to A in $\alpha\beta w$ is termed as "pruning the handle"

Where w contains only terminal symbols.

Shift-reduce parsing

Step 1: Locating a substring

Step 2: Choosing a production

	Stack	Input
Initial condition	\$	w\$
Final condition	\$S	\$

Viable Prefix- The sequence of symbols on the parsing stack

Stack	Input	Action
\$	id * id \$	shift
\$ id	* id \$	reduce F → id
\$ F	* id \$	shift
\$ F	* id \$	reduce T → F
\$ T	* id \$	shift
\$ T *	id \$	shift
\$ T * id	\$	reduce F → id
\$ T * F	\$	reduce T→T* F
\$ T	\$	reduce E → T
\$ E	\$	Accept

LR Parsing

- **LR(k) parsing, introduce by D. Knuth in 1965**
  - L is for Left-to-right scanning of the input
  - R is for reverse Rightmost derivation
  - k is the number of lookahead tokens
- **Most general non-backtracking shift-reduce parsing method**
- **Can be constructed to recognise virtually all programming language constructs**
- **The class of grammar is a proper superset of the class of grammars that can be parsed with *predictive parsers***

LR Parsing

- Different types:
  - *Simple LR or SLR, the easiest method for constructing shift-reduce parsers*
  - *Canonical LR*
  - *LALR*
- LR-parsers represent the DFA as a 2D – table
- Rows correspond to DFA states
- Columns correspond to terminals (action table) and non-terminals (goto table)

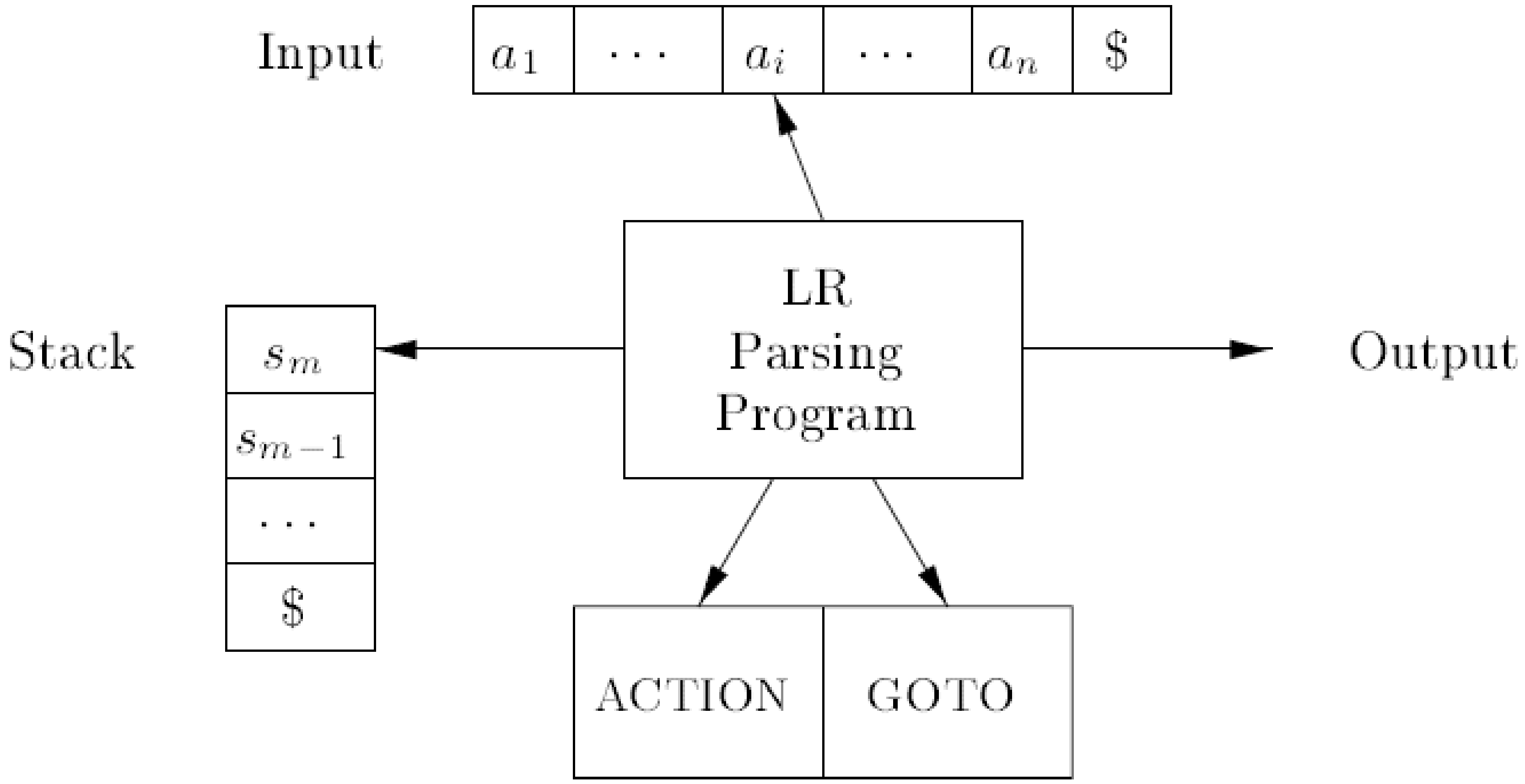
Transition between states

$$A \to a\,.\,X\,n \xrightarrow{X} A \to a\,X\,.\,n$$

Two possibilities

1. If X is a terminal, then push X onto the stack
2. If X is a non-terminal, X will be pushed onto the stack during a reduction by X → β.
3. Thus, we need to first recognize β
4. So for every production X → β, we need to indicate that X can be produced by reducing (recognising) any of the right hand sides of such productions.
5. This is basically the following transition.

$$A \to a\,.\,X\,n \xrightarrow{\varepsilon} X \to \beta$$



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

SLR Parsing

LR(0) - zero tokens of look-ahead

SLR - Simple LR: like LR(0), but uses FOLLOW sets to build more "precise" parsing tables

First we augment the grammar G with a new start symbol S' and a production

S' → S

Closure operation -

For a set of items I, we construct closure(I) as follows:

Every item in I is added to closure (I)

If $A \rightarrow \alpha \,.\, B \beta$ is in closure (I) and $B \rightarrow \gamma$ is a production, then $B \rightarrow \,.\, \gamma$ is added to closure (I)

Goto operation -

Goto(I, X) is the closure of the set of all items [$A \rightarrow \alpha X \,.\, \beta$], such that [$A \rightarrow \alpha \,.\, X \beta$] is in I.

Example

Augmented Grammar:

E' → E

E → E + T | T

T → T* F | F

F → (E) | id

If I = { E' → • E },

then CLOSURE(I) is

E' → • E

E → • E + T

E → • T

T → • T * F

T → • F

F → • (E)

F → • id

Example

Augmented Grammar:

E' → E

E → E + T | T

T → T* F | F

F → (E) | id

I =

E' → E •
E → E • + T

GOTO(I, +)

E → *E* + . *T*
T → . *T* * *F*
T → . *F*
F → . (*E*)
F → . **id**

The Canonical LR(0) Collection -- Example

I_0: $E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_1: $E' \rightarrow E.$
$E \rightarrow E.+T$

I_2: $E \rightarrow T.$
$T \rightarrow T.*F$

I_3: $T \rightarrow F.$

I_4: $F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5: $F \rightarrow id.$

I_6: $E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

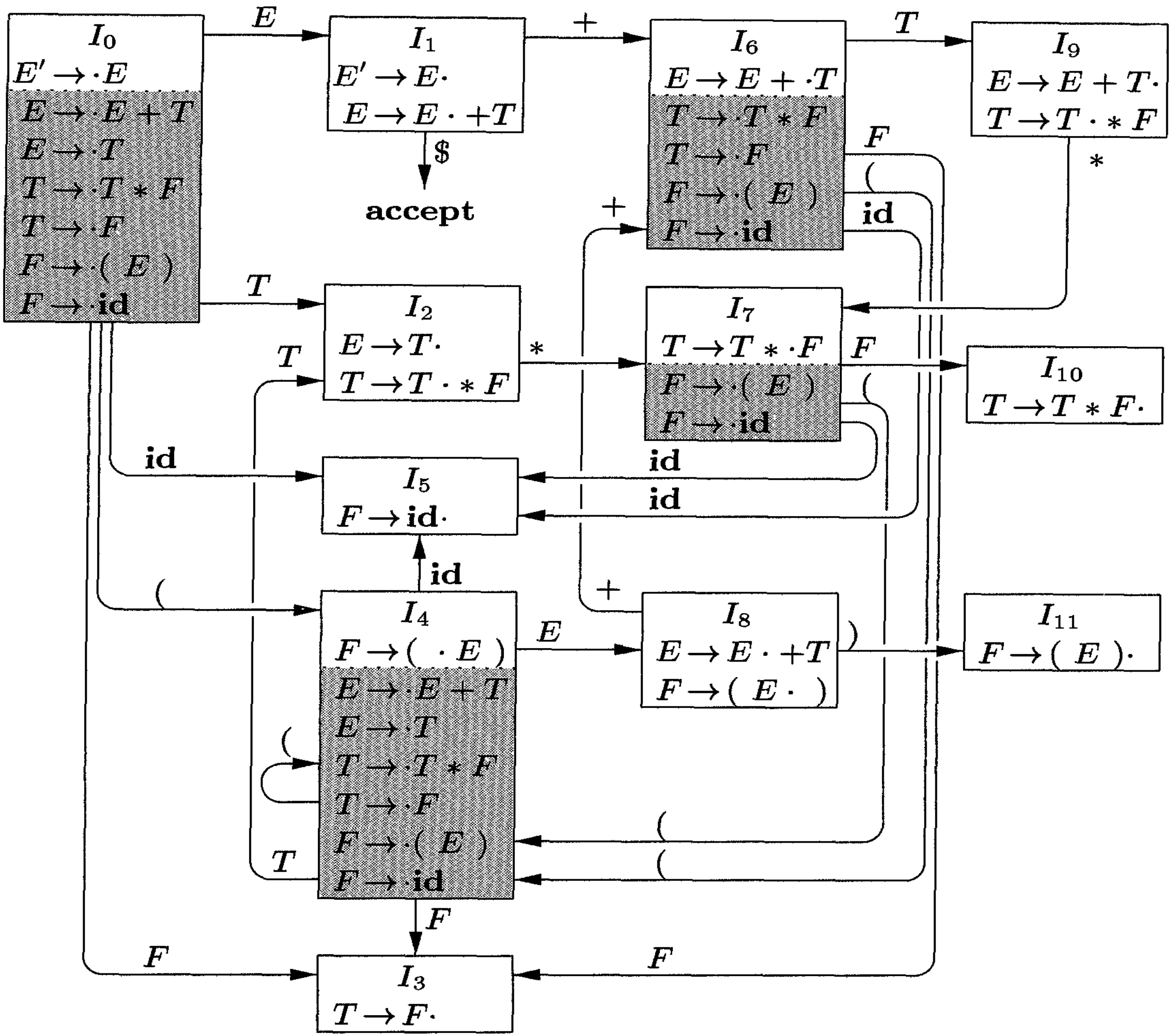
I_7: $T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_8: $F \rightarrow (E.)$
$E \rightarrow E.+T$

I_9: $E \rightarrow E+T.$
$T \rightarrow T.*F$

I_{10}: $T \rightarrow T*F.$

I_{11}: $F \rightarrow (E).$



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

Constructing SLR Parsing Table

Consider the grammar G'

1)Construct the canonical collection of sets of LR(0) items for G',

2)C ← {I_0, I_1 . . . I_n}

3)Create the parsing <u>action</u> table as follows:

1)If a is a terminal, A → α . a β in I_i, and goto(I_i,a) = I_j , then action[i,a] = shift j

2)If A → α . is in I_i , then action[i,a] is reduce A → α for all a in FOLLOW(A), where A <> S'

3)If S' → S is in I_I , then action [I, $] is accept

4)Create the parsing <u>goto</u> table

1)For all non-terminals A, if goto(I_i,A) = I_j , then goto[i,A] = j

5) **All other entries will be marked as error**

6)Initial state of the parser contains S' → S

Parsing Tables of Expression Grammar

E’ → E
1. E → E + T
2. E → T
3. T → T* F
4. T → F
5. F → (E)
6. F → id

	Action Table							Goto Table		
state	**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				

Parsing

Stack	Input	Parser Action
S0	id+id $	Shift and move to state 5
S0 id S5	+id $	Reduce F → id
S0 F S3	+id $	Reduce T→ id
S0 T S2	+id $	Reduce E → T
S0 E S1	+id $	Shift and move to state 6
S0 E S1 + S6	id $	Shift and move to state 5
S0 E S1 + S6 id S5	$	Reduce F → id
S0 E S1 + S6 F S3	$	Reduce T → F
S0 E S1 + S6 T S9	$	Reduce E → E+T
S0 E S1	$	Accept