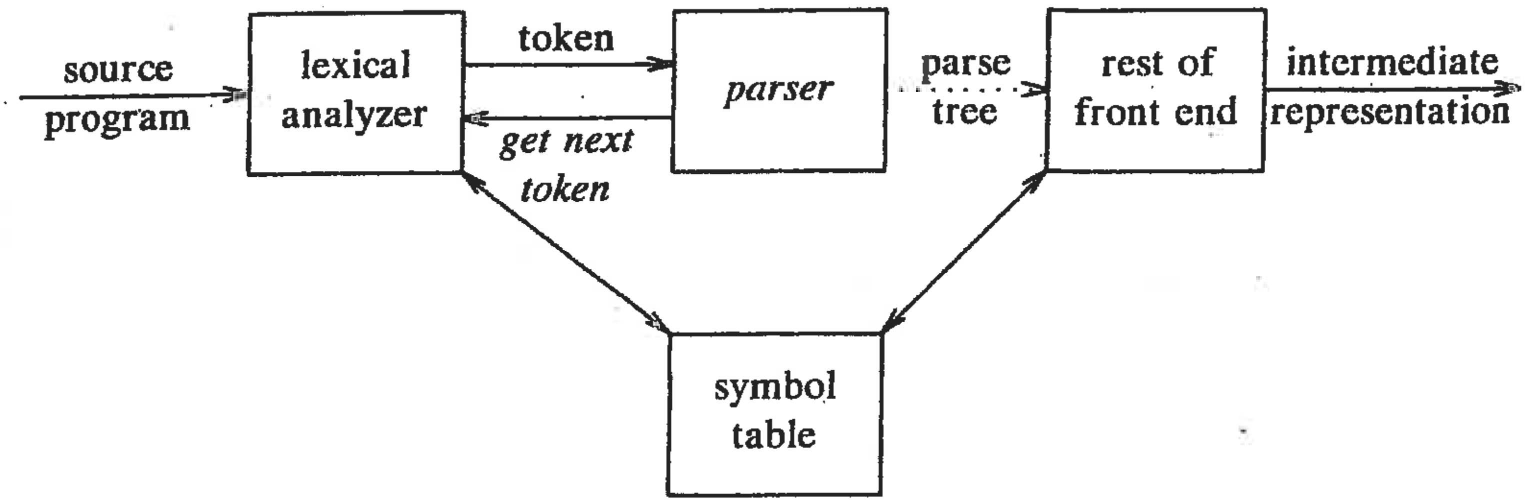


Fig. 4.1. Position of parser in compiler model.

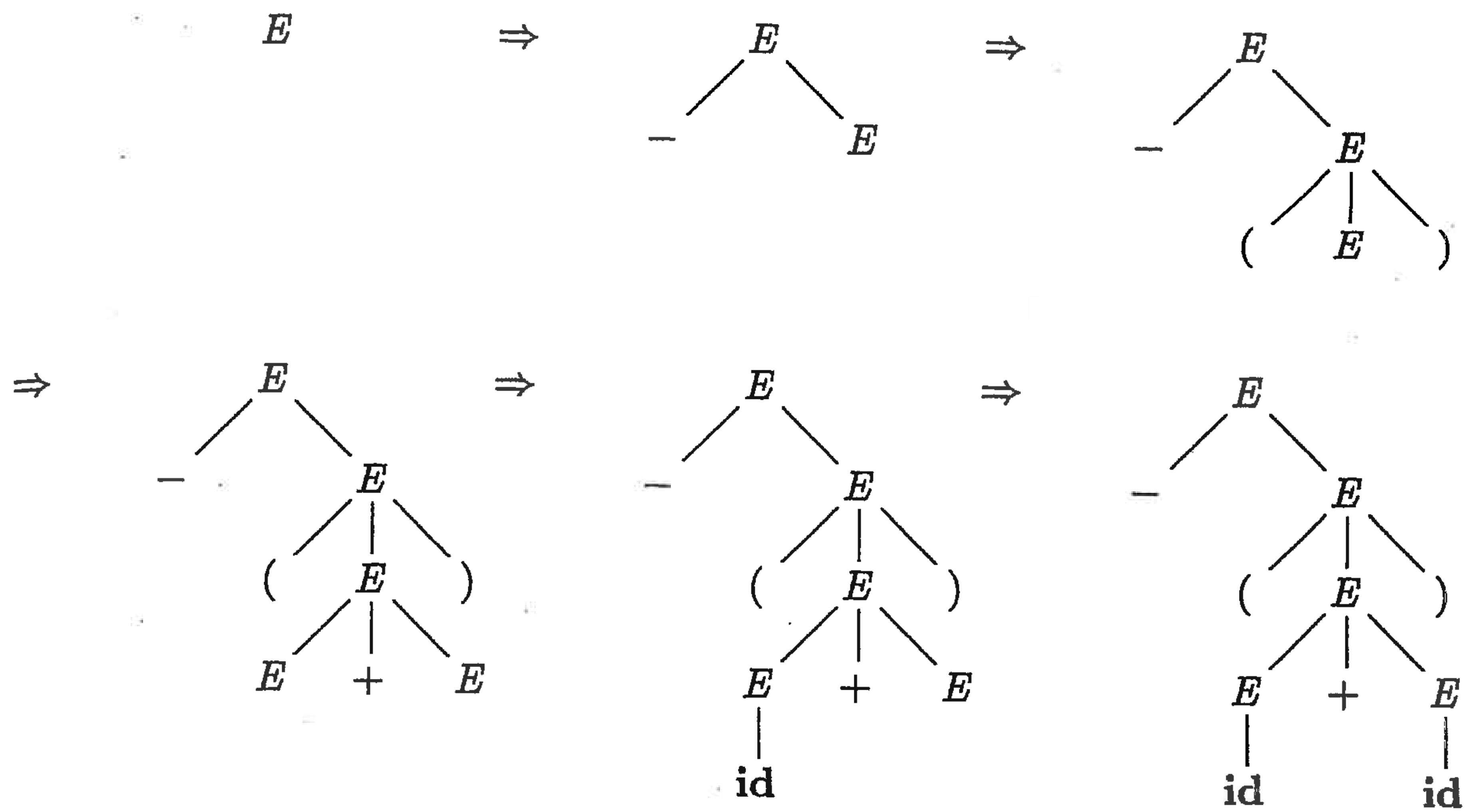


Figure 4.4: Sequence of parse trees for derivation (4.8)

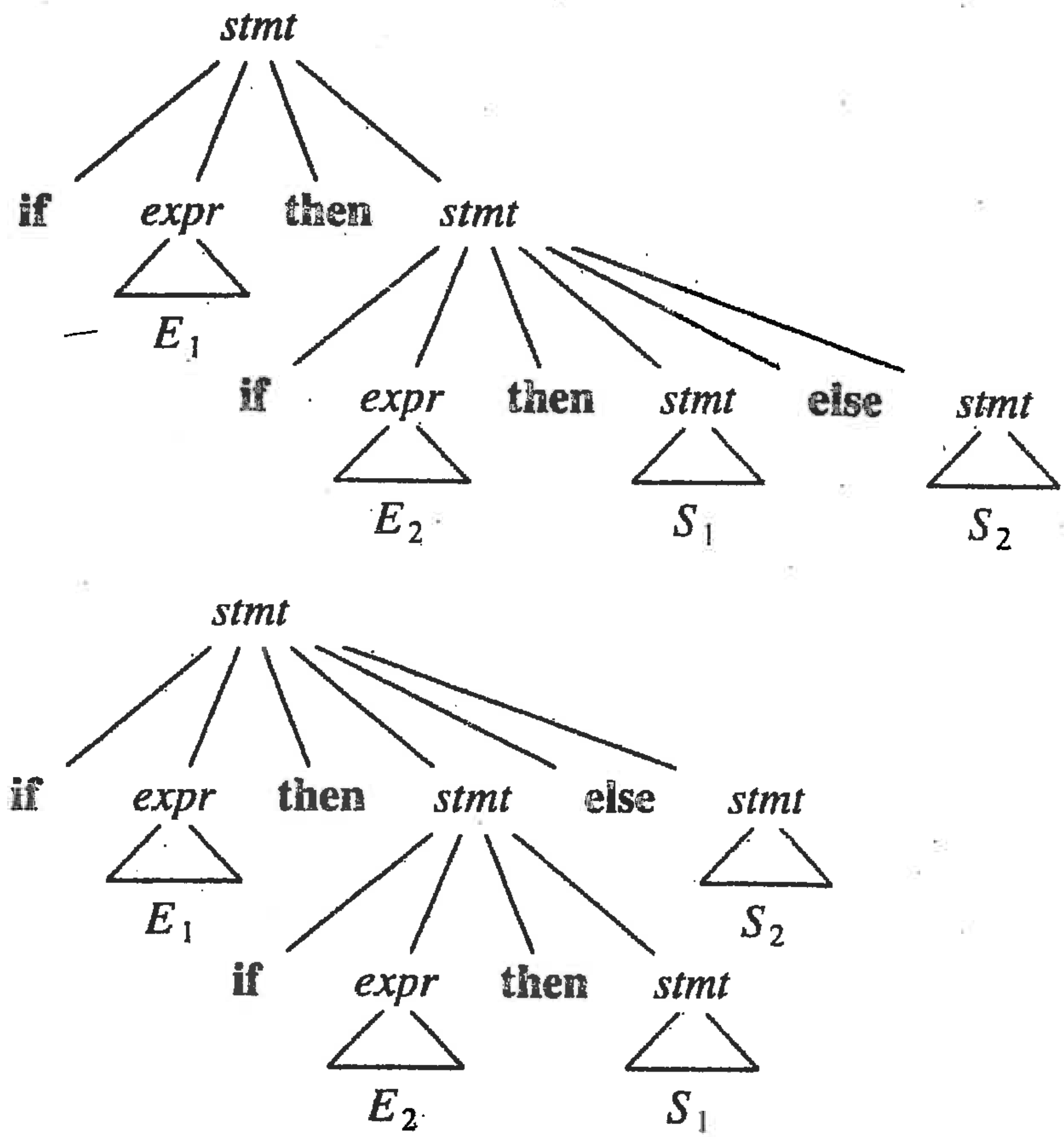


Fig. 4.8. Two parse trees for an ambiguous sentence.

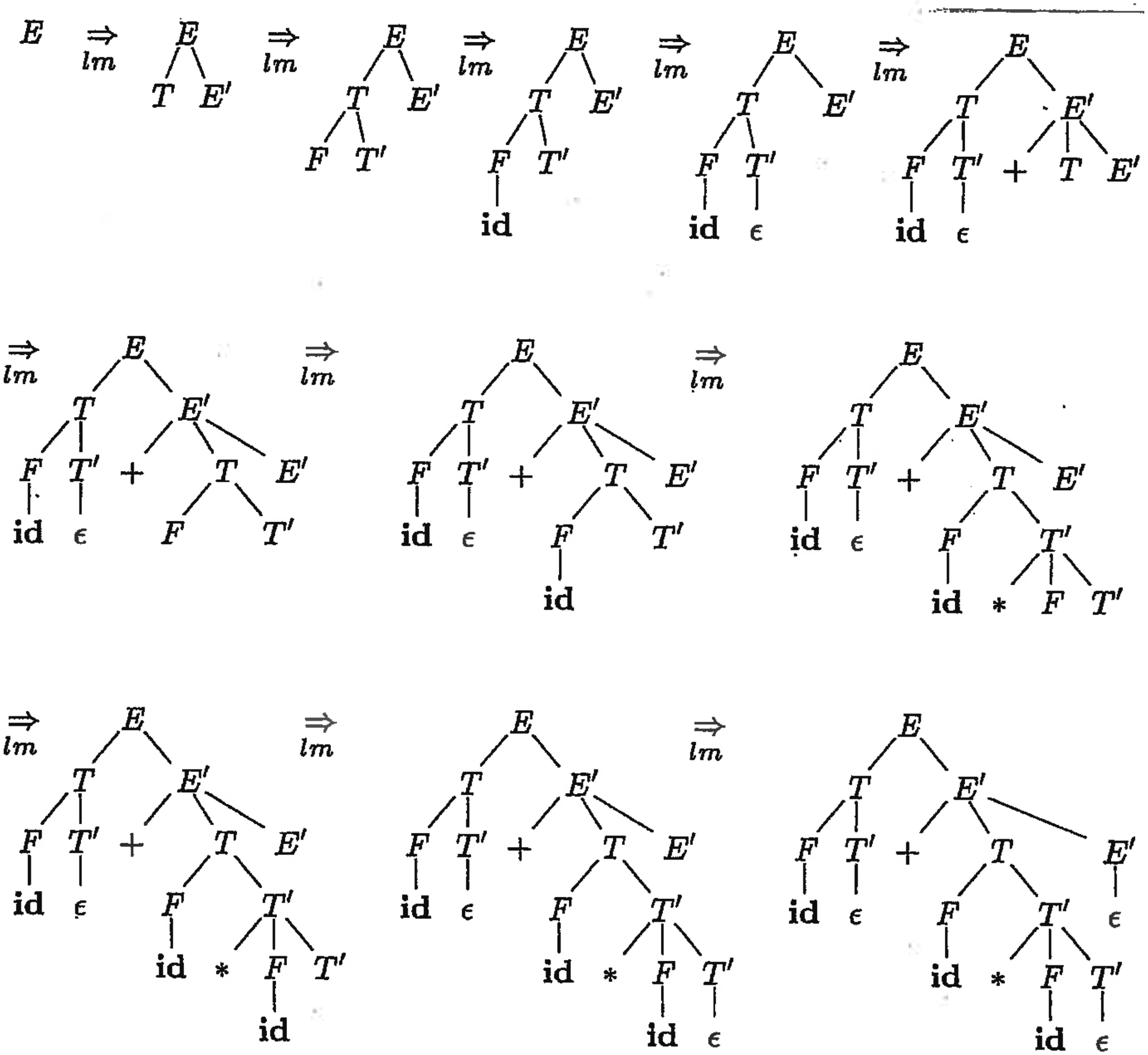


Figure 4.12: Top-down parse for **id** + **id** * **id**

NON - TERMINAL	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	$F \to \mathbf{id}$			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

NONTERMINAL	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		$E \rightarrow b$				

Fig. 4.18. Parsing table M for grammar (4.33).

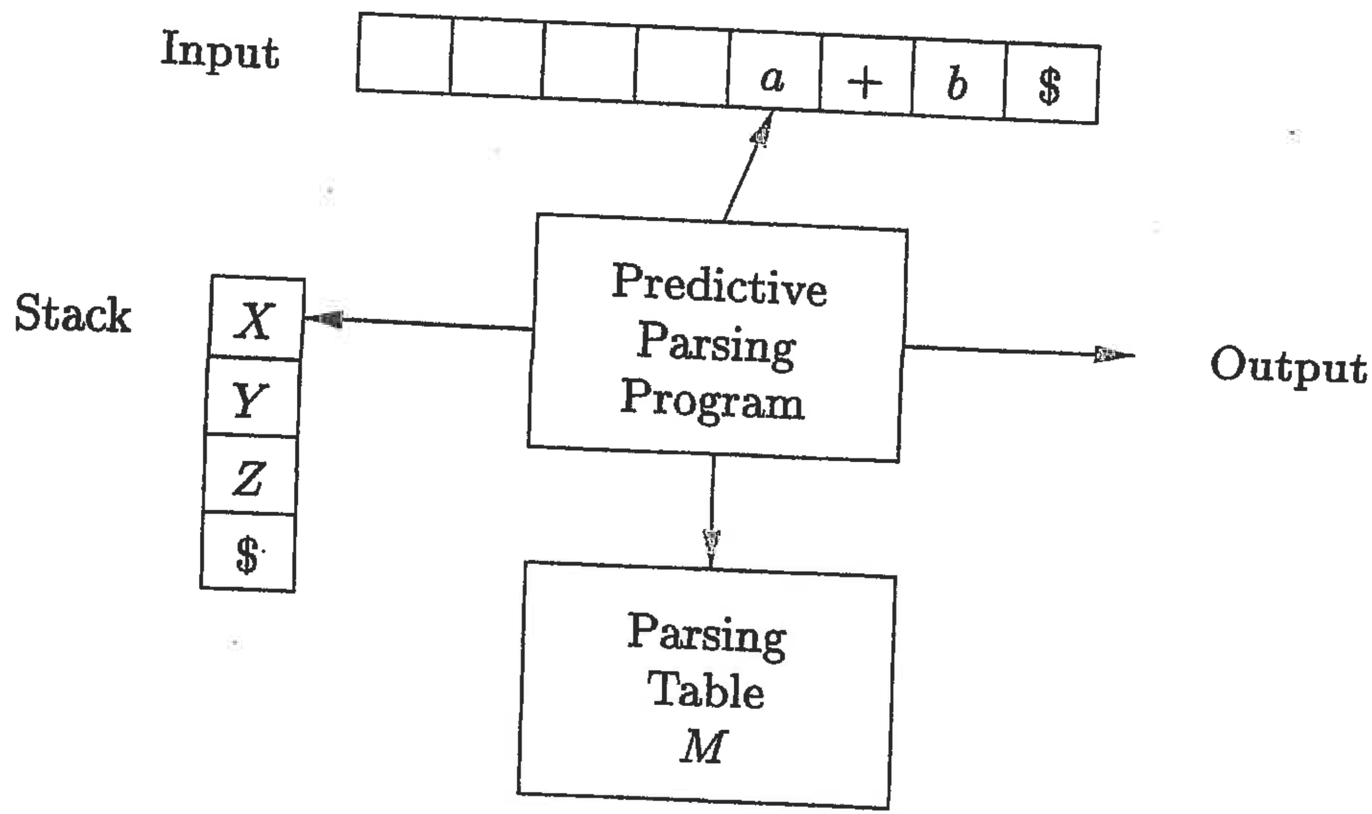


Figure 4.19: Model of a table-driven predictive parser

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

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

Fig. 4.22. Synchronizing tokens added to parsing table of Fig. 4.17.

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

Fig. 4.23. Parsing and error recovery moves made by predictive parser.

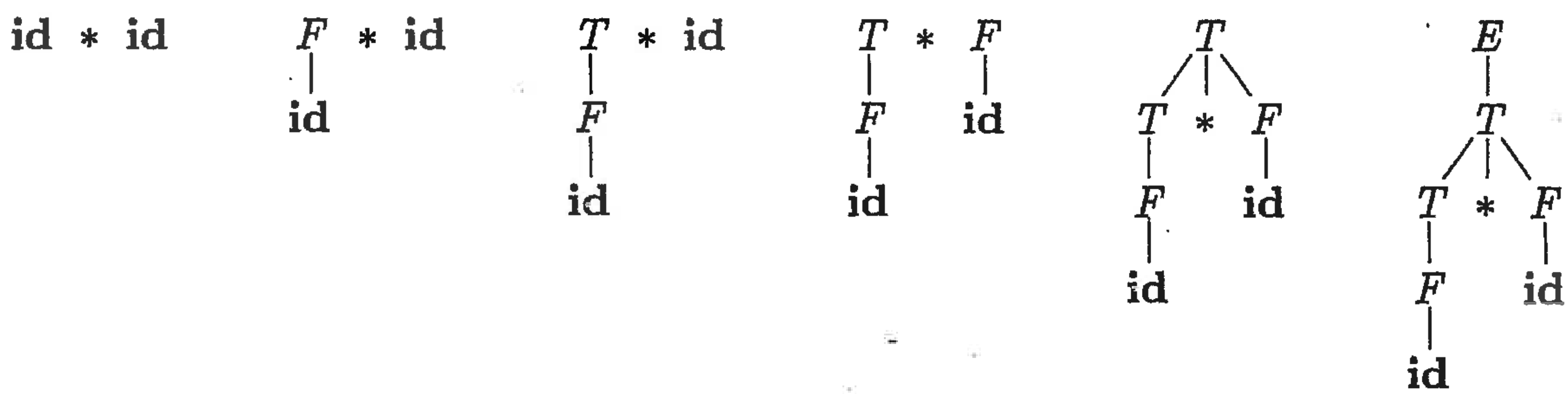


Figure 4.25: A bottom-up parse for **id** * **id**

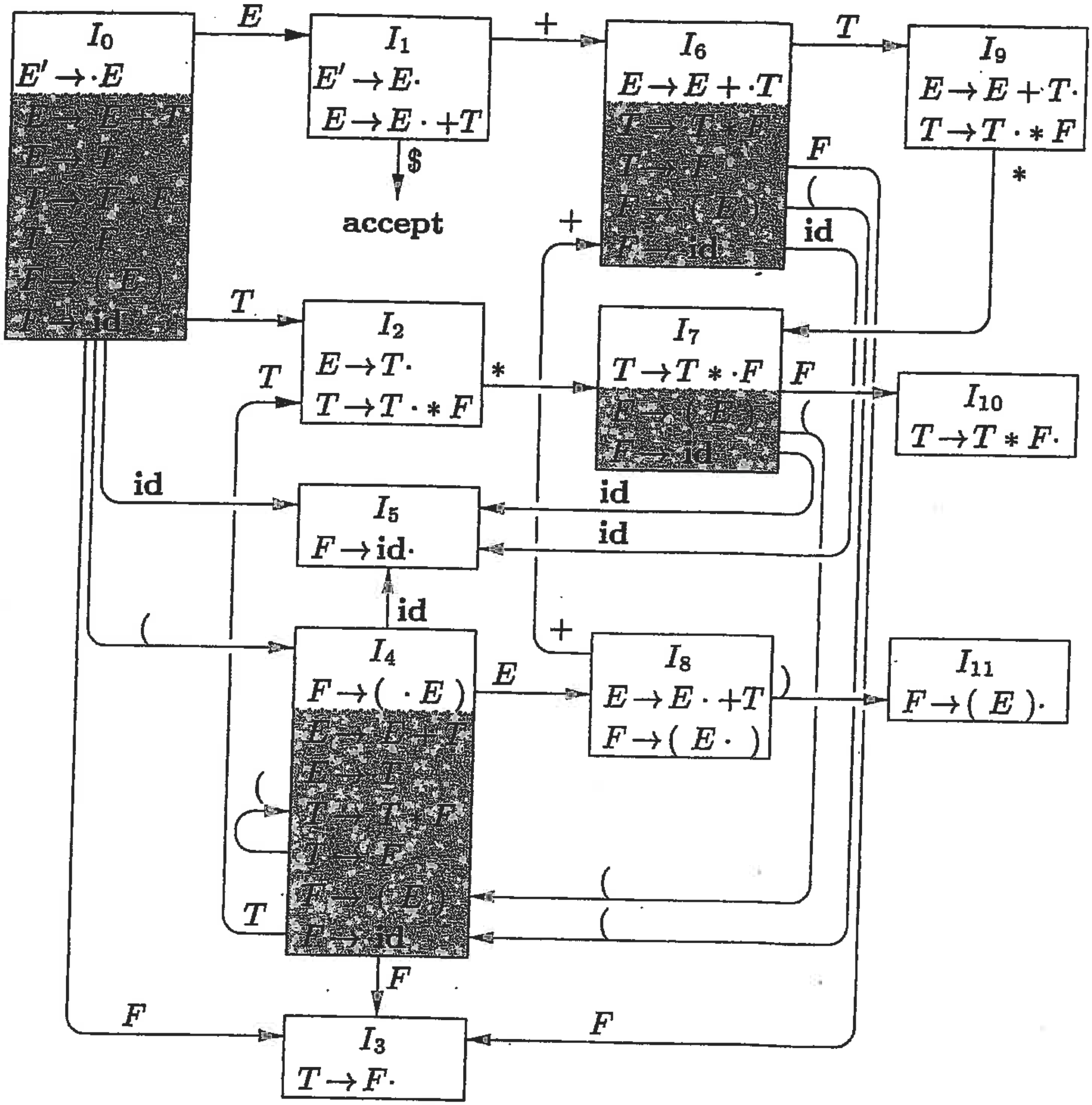


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production 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;
}
```

Figure 4.32: Computation of CLOSURE

```
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;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

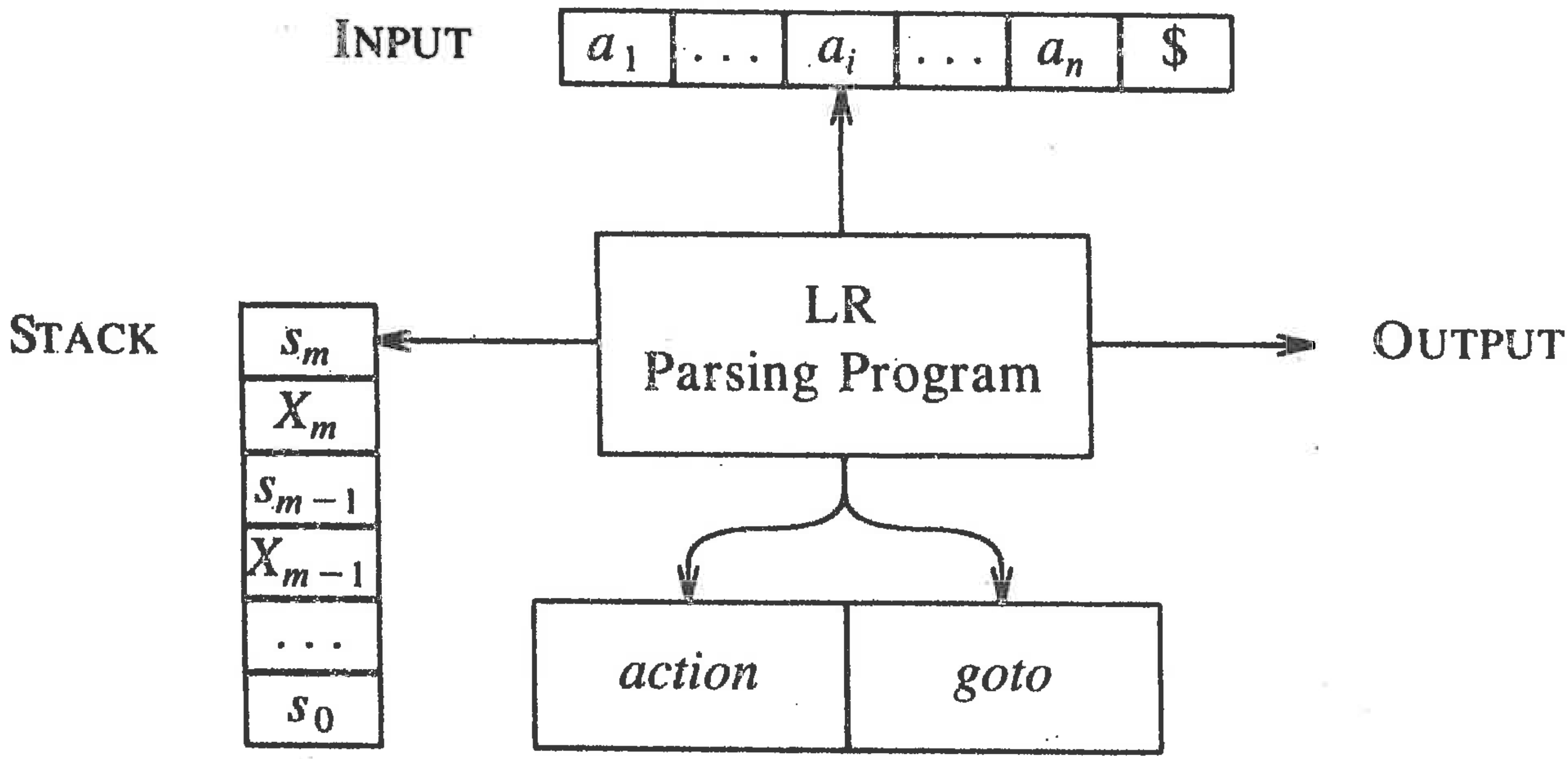


Fig. 4.35. Model of an LR parser.

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			

Fig. 4.37. Parsing table for expression grammar.

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

Fig. 4.38. Moves of LR parser on **id** $*$ **id** + **id**.

$$
\begin{aligned}
I_0:\quad & S' \rightarrow \cdot S \\
& S \rightarrow \cdot L = R \\
& S \rightarrow \cdot R \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id} \\
& R \rightarrow \cdot L
\end{aligned}
$$

$$
I_1:\quad S' \rightarrow S \cdot
$$

$$
\begin{aligned}
I_2:\quad & S \rightarrow L \cdot = R \\
& R \rightarrow L \cdot
\end{aligned}
$$

$$
I_3:\quad S \rightarrow R \cdot
$$

$$
\begin{aligned}
I_4:\quad & L \rightarrow * \cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id}
\end{aligned}
$$

$$
I_5:\quad L \rightarrow \mathbf{id} \cdot
$$

$$
\begin{aligned}
I_6:\quad & S \rightarrow L = \cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id}
\end{aligned}
$$

$$
I_7:\quad L \rightarrow * R \cdot
$$

$$
I_8:\quad R \rightarrow L \cdot
$$

$$
I_9:\quad S \rightarrow L = R \cdot
$$

Fig. 4.39. Canonical LR(0) collection for grammar (4.49).

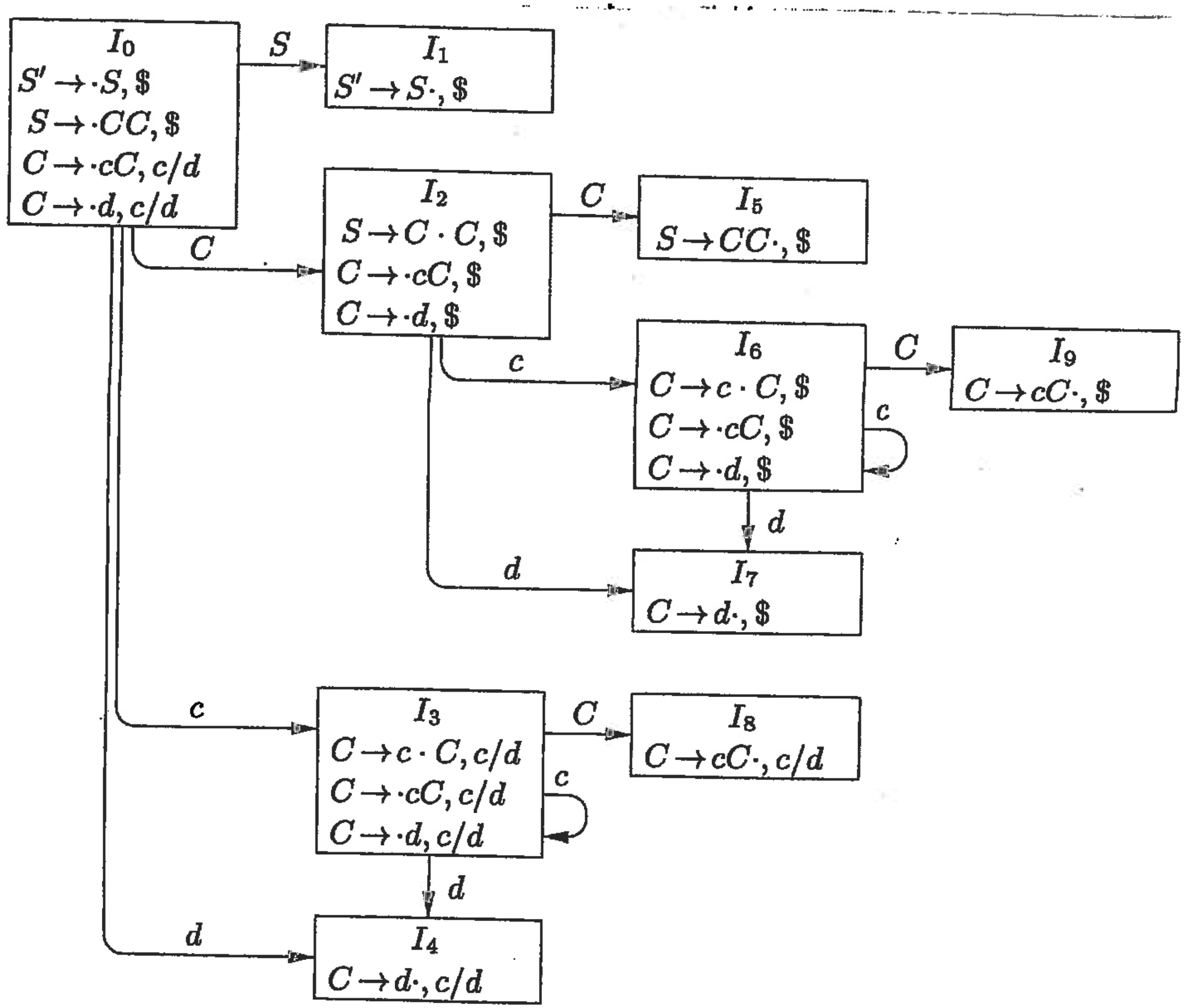


Figure 4.41: The GOTO graph for grammar (4.55)

STATE	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.42. Canonical parsing table for grammar (4.55).

STATE	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.43. LALR parsing table for grammar (4.54).

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_3: $E \to \mathbf{id}\cdot$

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

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_6: $E \to (E\cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E)\cdot$

Figure 4.48: Sets of LR(0) items for an augmented expression grammar

STATE	ACTION						GOTO
	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		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Figure 4.49: Parsing table for grammar (4.3)

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S \cdot$

I_2: $S \to i \cdot SeS$
$S \to i \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a \cdot$

I_4: $S \to iS \cdot eS$
$S \to iS \cdot$

I_5: $S \to iSe \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS \cdot$

Fig. 4.50. LR(0) states for augmented grammar (4.67).

STATE	*action*				*goto*
	i	*e*	*a*	\$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Fig. 4.51. LR parsing table for abstract "dangling-else" grammar

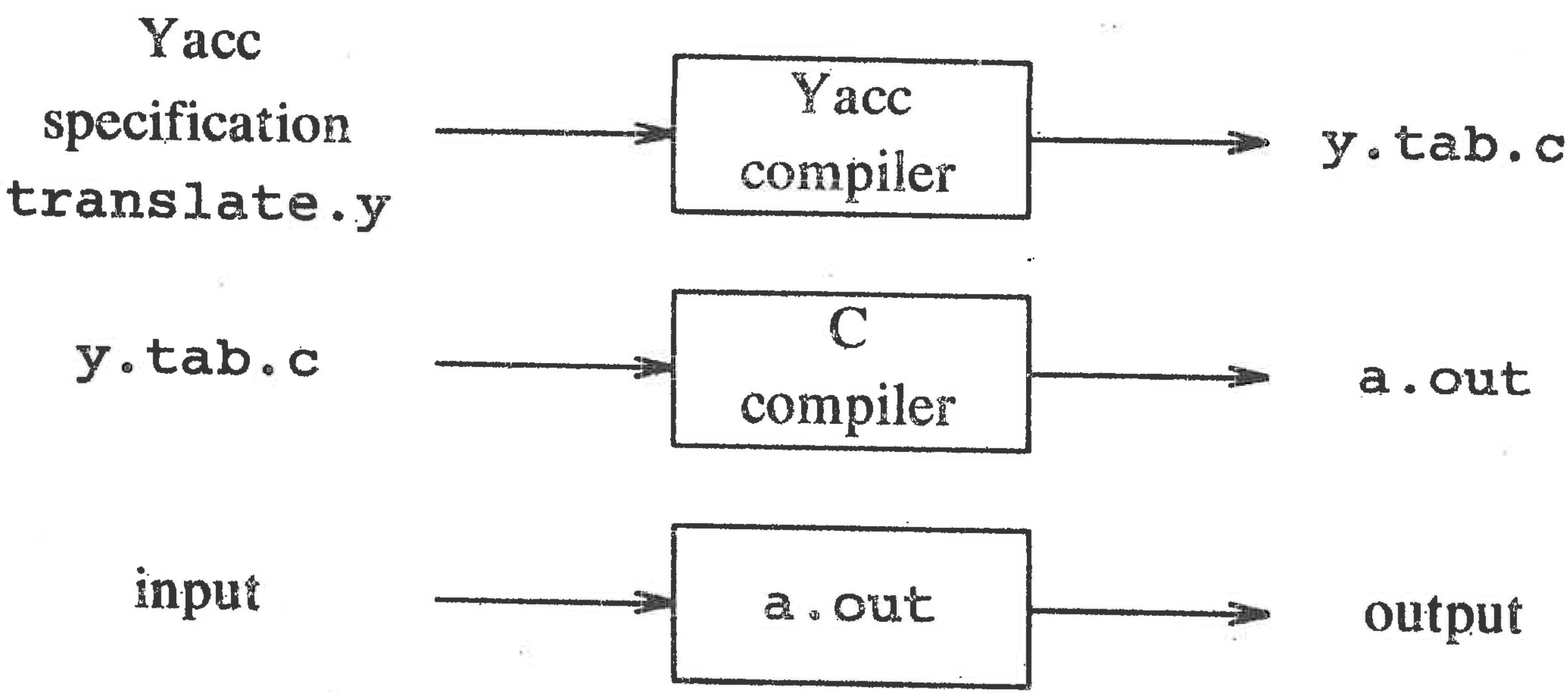


Fig. 4.57. Creating an input/output translator with Yacc.

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :   expr '\n'          { printf("%d\n", $1);
        ;
expr    :   expr '+' term      { $$ = $1 + $3; }
        |   term
        ;
term    :   term '*' factor    { $$ = $1 * $3; }
        |   factor
        ;
factor  :   '(' expr ')'       { $$ = $2; }
        |   DIGIT
        ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Fig. 4.58. Yacc specification of a simple desk calculator.

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines     :  lines expr '\n'   { printf("%g\n", $2); }
          |  lines '\n'
          |  /* ε */
          ;
expr      :  expr '+' expr     { $$ = $1 + $3; }
          |  expr '-' expr     { $$ = $1 - $3; }
          |  expr '*' expr     { $$ = $1 * $3; }
          |  expr '/' expr     { $$ = $1 / $3; }
          |  '(' expr ')'      { $$ = $2; }
          |  '-' expr  %prec UMINUS  { $$ = - $2; }
          |  NUMBER
          ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Fig. 4.59. Yacc specification for a more advanced desk calculator.

```
number    [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
[ ]       { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
                return NUMBER; }
\n|.      { return yytext[0]; }
```

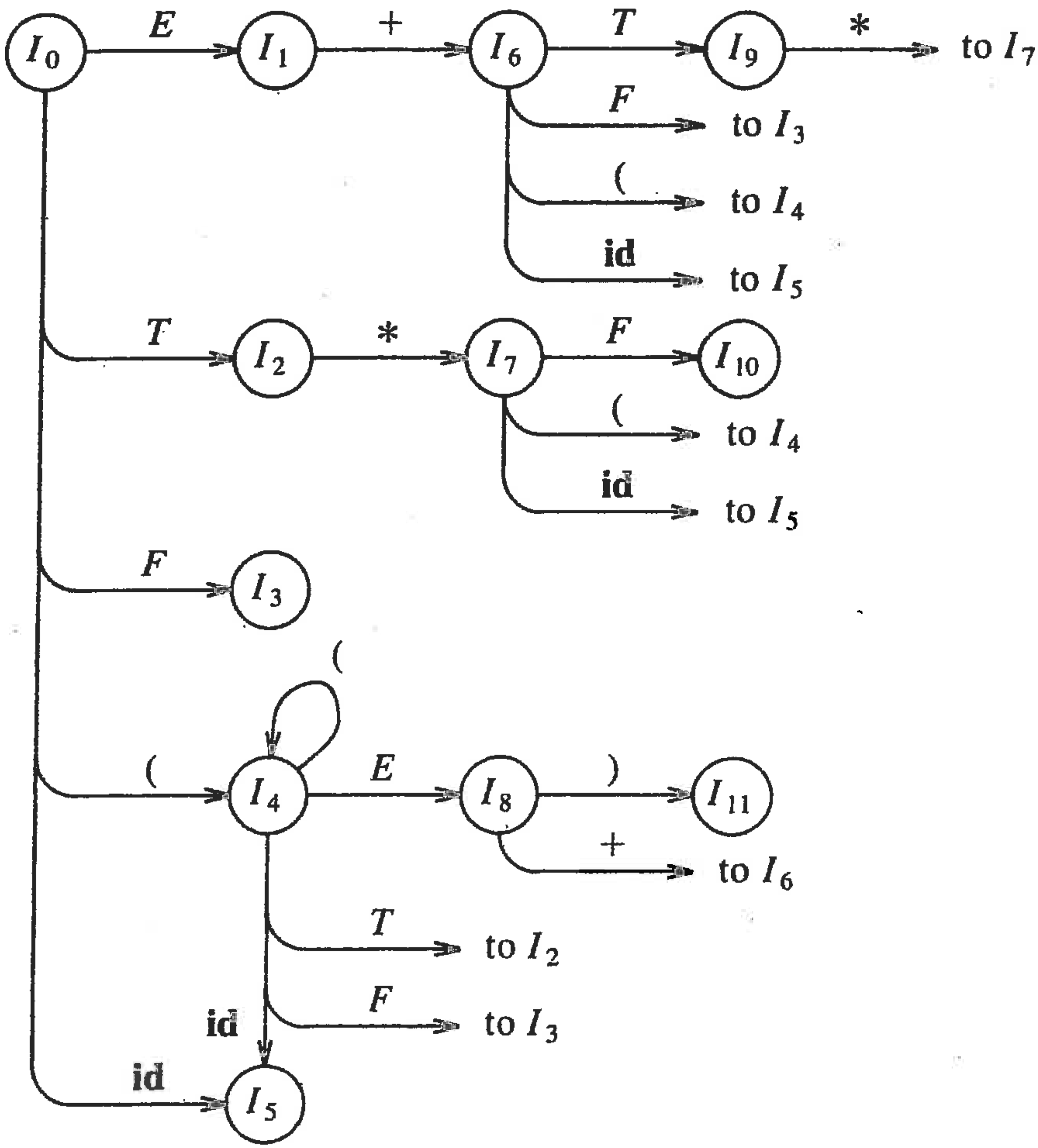
Figure 4.60: Lex specification for yylex() in Fig. 4.59

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   :  lines expr '\n'   { printf("%g\n", $2); }
        |  lines '\n'
        |  /* empty */
        |  error '\n'{ yyerror("reenter last line:");
                       yyerrok; }
        ;
expr    :  expr '+' expr    { $$ = $1 + $3; }
        |  expr '-' expr    { $$ = $1 - $3; }
        |  expr '*' expr    { $$ = $1 * $3; }
        |  expr '/' expr    { $$ = $1 / $3; }
        |  '(' expr ')'     { $$ = $2; }
        |  '-' expr  %prec UMINUS  { $$ = - $2; }
        |  NUMBER
        ;
%%
#include "lex.yy.c"
```

Fig. 4.61. Desk calculator with error recovery.



A Transition diagram of DFA D for viable prefixes.

I_0: $E' \to \cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot \{ E \}$
$E \to \cdot c$

I_1: $E' \to E \cdot$
$E \to E \cdot$**sub** E **sup** E
$E \to E \cdot$**sub** E
$E \to E \cdot$**sup** E

I_2: $E \to \{ \cdot E \}$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot \{ E \}$
$E \to \cdot c$

I_3: $E \to c \cdot$

I_4: $E \to E$ **sub**$\cdot E$ **sup** E
$E \to E$ **sub**$\cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot \{ E \}$
$E \to \cdot c$

I_5: $E \to E$ **sup**$\cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot \{ E \}$
$E \to \cdot c$

I_6: $E \to E \cdot$**sub** E **sup** E
$E \to E \cdot$**sub** E
$E \to E \cdot$**sup** E
$E \to \{ E \cdot \}$

I_7: $E \to E \cdot$**sub** E **sup** E
$E \to E$ **sub** $E \cdot$**sup** E
$E \to E \cdot$**sub** E
$E \to E$ **sub** $E \cdot$
$E \to E \cdot$**sup** E

I_8: $E \to E \cdot$**sub** E **sup** E
$E \to E \cdot$**sub** E
$E \to E \cdot$**sup** E
$E \to E$ **sup** $E \cdot$

I_9: $E \to \{ E \} \cdot$

I_{10}: $E \to E$ **sub** E **sup**$\cdot E$
$E \to E$ **sup**$\cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot \{ E \}$
$E \to \cdot c$

I_{11}: $E \to E \cdot$**sub** E **sup** E
$E \to E$ **sub** E **sup** $E \cdot$
$E \to E \cdot$**sub** E
$E \to E \cdot$**sup** E
$E \to E$ **sup** $E \cdot$

Fig. B. LR(0) sets of items for grammar (4.25).

STATE	*action*						*goto*
	sub	sup	{	}	*c*	$	*E*
0			s2		s3		1
1	s4	s5				acc	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

Fig. C. Parsing table for grammar (4.25).

Below is an example input file.

```
E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
```

Below is the corresponding example output file. The sets of items should actually be printed in a single column, but was not on this page to save paper.

```
Augmented Grammar
-----------------
'->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i

Sets of LR(0) Items
-------------------
I0:
   '->@E                    goto(E)=I1
   E->@E+T
   E->@T                    goto(T)=I2
   T->@T*F
   T->@F                    goto(F)=I3
   F->@(E)                  goto(()=I4
   F->@i                    goto(i)=I5

I1:
   '->E@
   E->E@+T                  goto(+)=I6

I2:
   E->T@
   T->T@*F                  goto(*)=I7

I3:
   T->F@

I4:
   F->(@E)                  goto(E)=I8
   E->@E+T
   E->@T                    goto(T)=I2
   T->@T*F
   T->@F                    goto(F)=I3
   F->@(E)                  goto(()=I4
   F->@i                    goto(i)=I5

I5:
   F->i@

I6:
   E->E+@T                  goto(T)=I9
   T->@T*F
   T->@F                    goto(F)=I3
   F->@(E)                  goto(()=I4
   F->@i                    goto(i)=I5

I7:
   T->T*@F                  goto(F)=I10
   F->@(E)                  goto(()=I4
   F->@i                    goto(i)=I5

I8:
   F->(E@)                  goto())=I11
   E->E@+T                  goto(+)=I6

I9:
   E->E+T@
   T->T@*F                  goto(*)=I7

I10:
   T->T*F@

I11:
   F->(E)@
```

STATE	action						goto
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Parsing table for grammar (4.22).