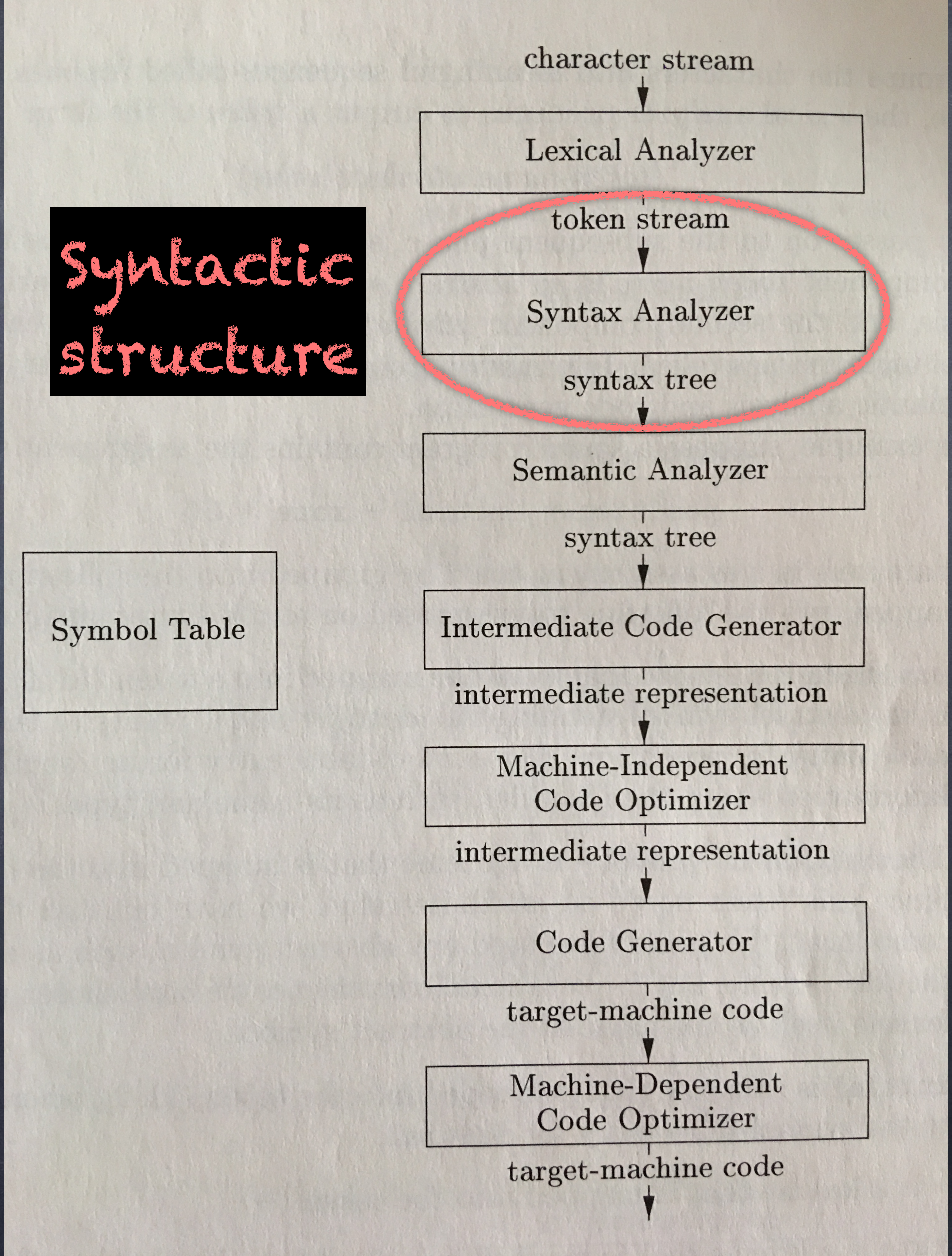
CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

Figure 1.7, page 5 of text



Syntactic structure

character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

target-machine code

Symbol Table

NOTE!

Textbook Typo:

- On page 254, line-4:
- "Fig. 4.31" should be "Fig. 4.37".

[pg. 242]

- "The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known"
- "[The LR-parsing method] can be implemented as efficiently as other [...] shift-reduce methods"
- "An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input."
- "The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods."

LR(k)

- LR(k) parser
  - L => left-to-right scanning of input
  - R => rightmost derivation in reverse
  - k => number of lookahead symbols
  - k is typically 0 or 1
  - LR => LR(1)

Lookahead here refers to how many input symbols can be consulted during parsing

LR(0) automaton and SLR

- SLR => Simple LR
- LR(0) automaton is constructed from G'
- "Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next input symbol a if state j has a transition on a. Otherwise, we choose to reduce; the items in state j will tell us which production to use." [p 247]

TERMINOLOGY

LR(0) automaton and SLR

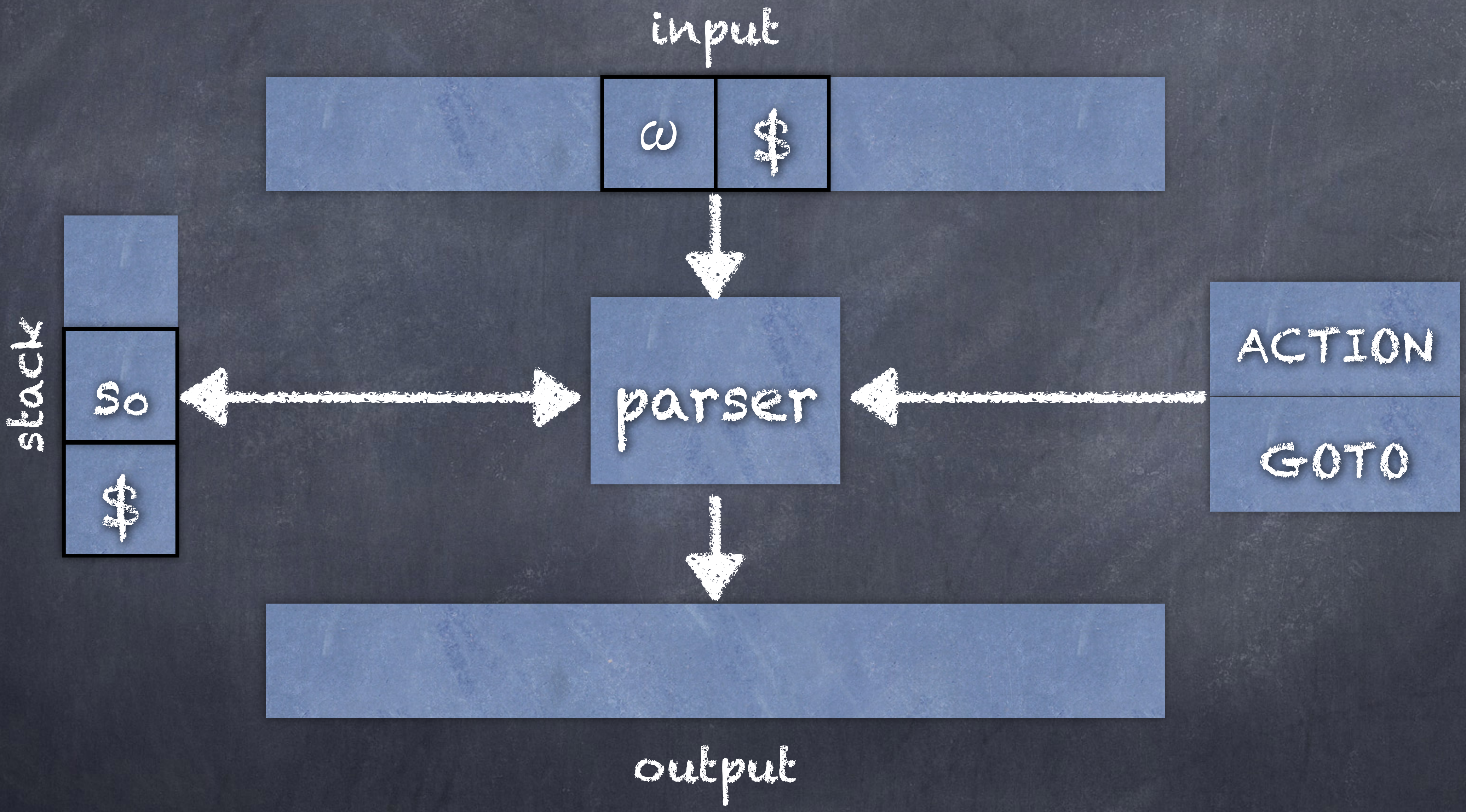
- SLR => Simple LR
- LR(0) automaton is constructed from G'
- "S r
  sy rom
  th n,
  sh j
  h
  ch j
  will tell us which production to use." [p 247]

Lookahead here refers to how many input symbols can be consulted during automaton construction (i.e. in the items)

We'll see later how to incorporate lookahead in the items.

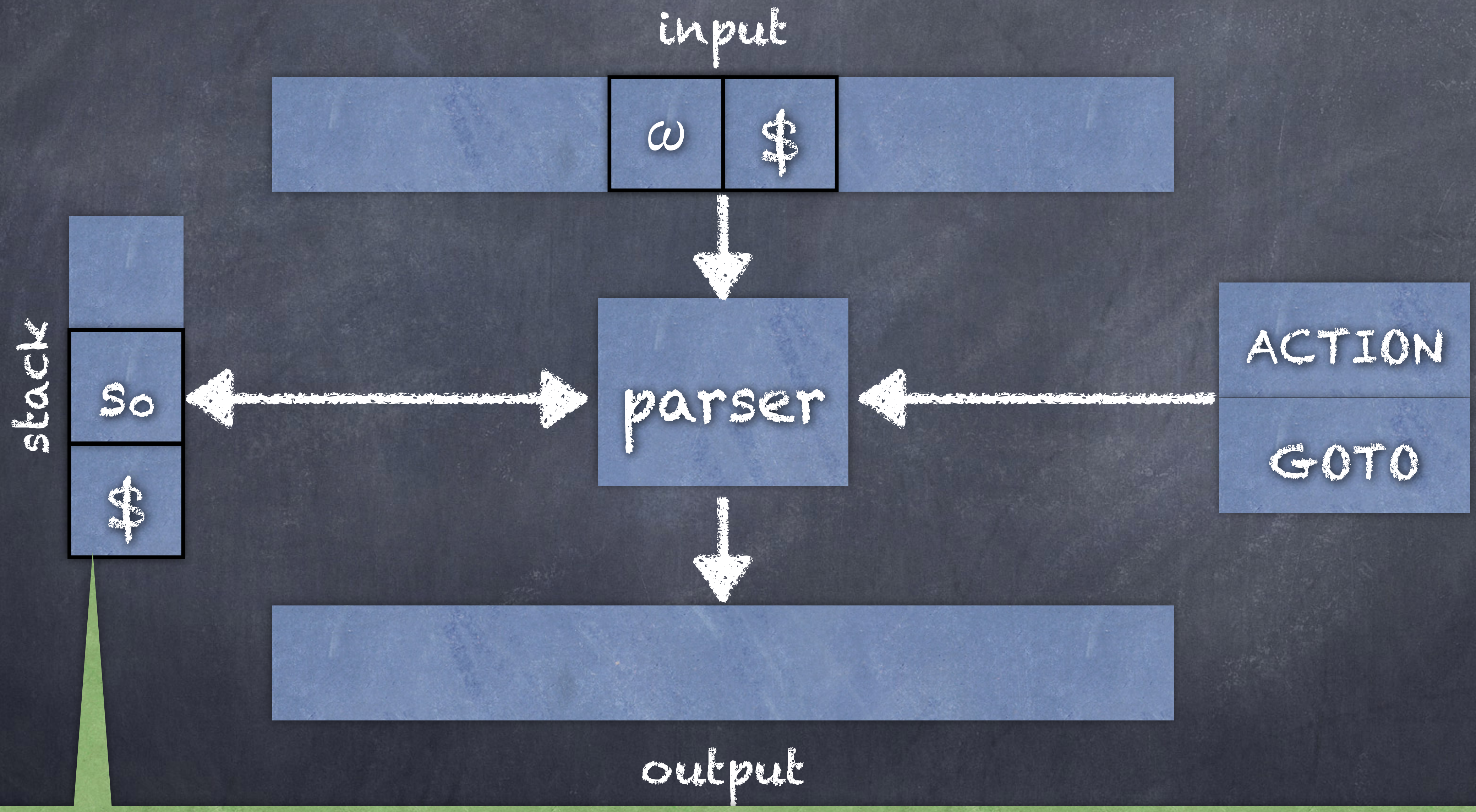
Initial state of the parser
(top of stack is current state in LR(0) automata)

modified from figure 4.35 [p. 248]



Initial state of the parser (top of stack is current state in LR(0) automata)

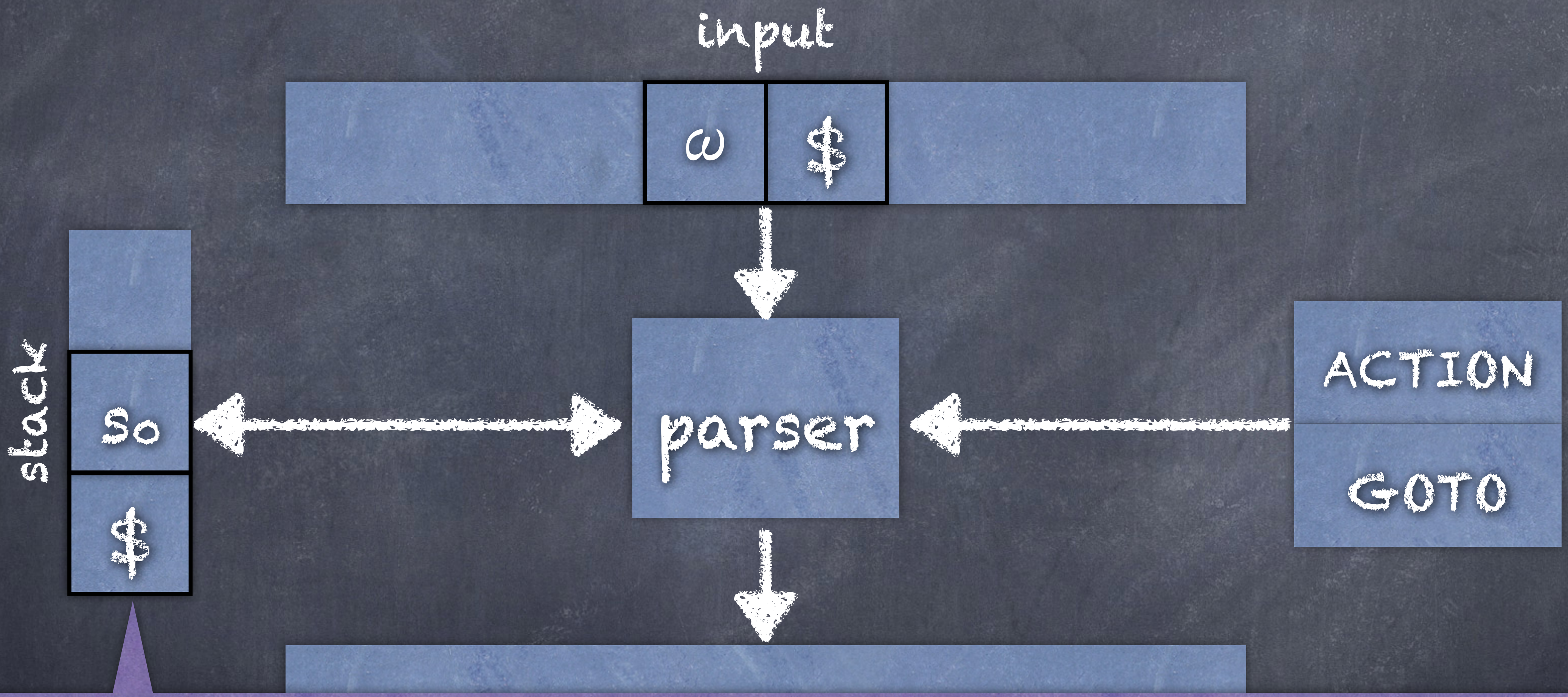
modified from figure 4.35 [p. 248]



"In the SLR method, the stack holds states from the LR(o) automaton; the canonical LR and LALR methods are similar." [p. 248]

Initial state of the parser (top of stack is current state in LR(0) automata)

modified from figure 4.35 [p. 248]



"By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state i to state j if GOTO(I_i,X) = I_j. All transitions to state j must be for the same grammar symbol X. Thus, each state, except the start state 0, has a unique grammar symbol associated with it." [p. 248]

written as s_0

LR parsing table

- ACTION function
  - Inputs: state i and an input symbol a (terminal or \$)
  - ACTION[i,a] is:
    - Shift j - shift a onto stack, using state j to represent a
    - Reduce A -> β
    - Accept
    - Error
- GOTO function - extend from sets of items to states.
  - GOTO[I_i,A] = I_j => GOTO[i,A] = j

Algorithm 4.46 [p. 253]
Constructing an SLR-parsing table

INPUT: An augmented grammar G'

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'
2. State i is constructed from I_i. The parsing items for state i are determined as follows:
    A. If $[A \to \alpha \bullet a \beta]$ is in I_i and $GOTO(I_i, a) = I_j$, then set ACTION[i,a] to "shift j". Here a must be a terminal.
    B. If $[A \to \alpha \bullet]$ is in I_i, then set ACTION[i,a] to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'.
    C. If $[S' \to S \bullet]$ is in I_i, then set ACTION[i,\$] to "accept."

If conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state I are constructed for all nonterminals A using the rule: If $GOTO(I_i, A) = I_j$, then GOTO[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error".
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \bullet S]$

FIRST(X)

REVIEW

- if $X \in T$ then $\text{FIRST}(X) = \{ X \}$
- if $X \in N$ and $X \to Y_1 \, Y_2 \ldots Y_k \in P$ for $k \geq 1$, then
    - add $a \in T$ to $\text{FIRST}(X)$ if $\exists i$ s.t. $a \in \text{FIRST}(Y_i)$ and $\varepsilon \in \text{FIRST}(Y_j) \; \forall \, j < i$ (i.e. $Y_1 \, Y_2 \ldots Y_k \Rightarrow^* \varepsilon$)
    - if $\varepsilon \in \text{FIRST}(Y_j) \; \forall \, j < k$ add ε to $\text{FIRST}(X)$

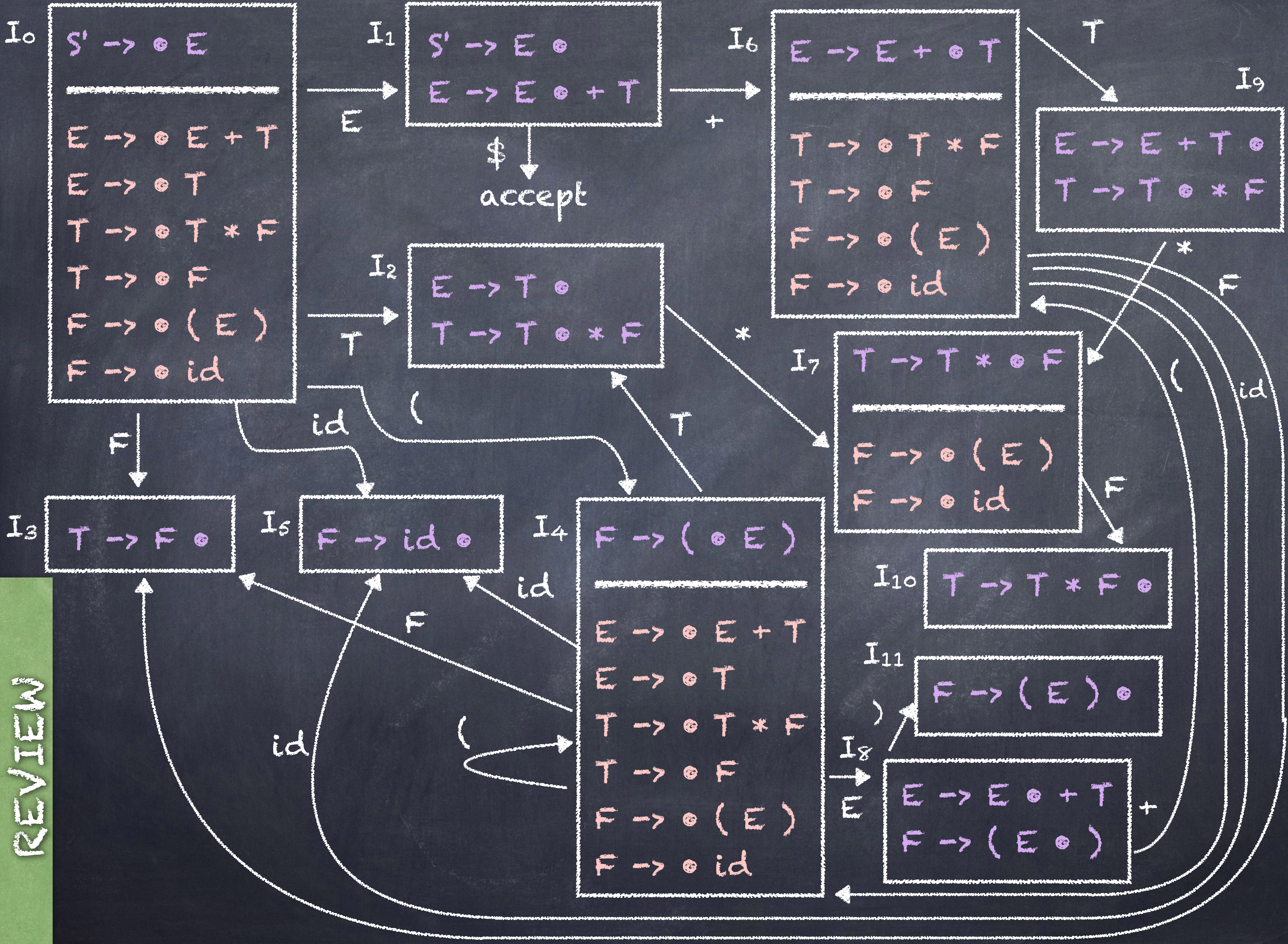
FOLLOW(X)

- Place \$ in FOLLOW(S), where S is the start symbol (\$ is an end marker)
- if A -> $\alpha B \beta \in P$, then FIRST(β) - $\{\varepsilon\}$ is in FOLLOW(B)
- if A -> $\alpha B \in P$ or A -> $\alpha B \beta \in P$ where $\varepsilon \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B)

FIRST(X) and FOLLOW(X)

X	FIRST(X)	FOLLOW(X)
S'	id, (	$
E	id, (	+,), $
T	id, (	*, +,), $
F	id, (	*, +,), $
id	id	*, +,), $
(	(	id, (
)	)	*, +,), $
+	+	id, (
*	*	id, (

REVIEW



I_0

S' -> • E

E -> • E + T

E -> • T

T -> • T * F

T -> • F

F -> • (E)

F -> • id

I_1

S' -> E •

E -> E • + T

I_2

E -> T •

T -> T • * F

I_3

T -> F •

I_4

F -> (• E)

E -> • E + T

E -> • T

T -> • T * F

T -> • F

F -> • (E)

F -> • id

I_5

F -> id •

I_6

E -> E + • T

T -> • T * F

T -> • F

F -> • (E)

F -> • id

I_7

T -> T * • F

F -> • (E)

F -> • id

I_8

E -> E • + T

F -> (E •)

I_9

E -> E + T •

T -> T • * F

I_{10}

T -> T * F •

I_{11}

F -> (E) •

2A. If $[A \to \alpha \bullet a\beta]$ is in I_i and $GOTO(I_i,a)= I_j$, then set ACTION[i,a] to "shift j". Here a must be a terminal. This will be written as 'sj' where j is a state number.

2B. If $[A \to \alpha \bullet]$ is in I_i, then set ACTION[i,a] to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'. This will be written as 'rp' where p is a production number (see below).

2C. If $[S' \to S \bullet]$ is in I_i, then set ACTION[i,\$] to "accept."

3. The goto transitions for state I are constructed for all nonterminals A using the rule: If $GOTO(I_i,A)=I_j$, then GOTO[i,A] = j. This will be written as 'j' where j is a state number.

Production numbers:

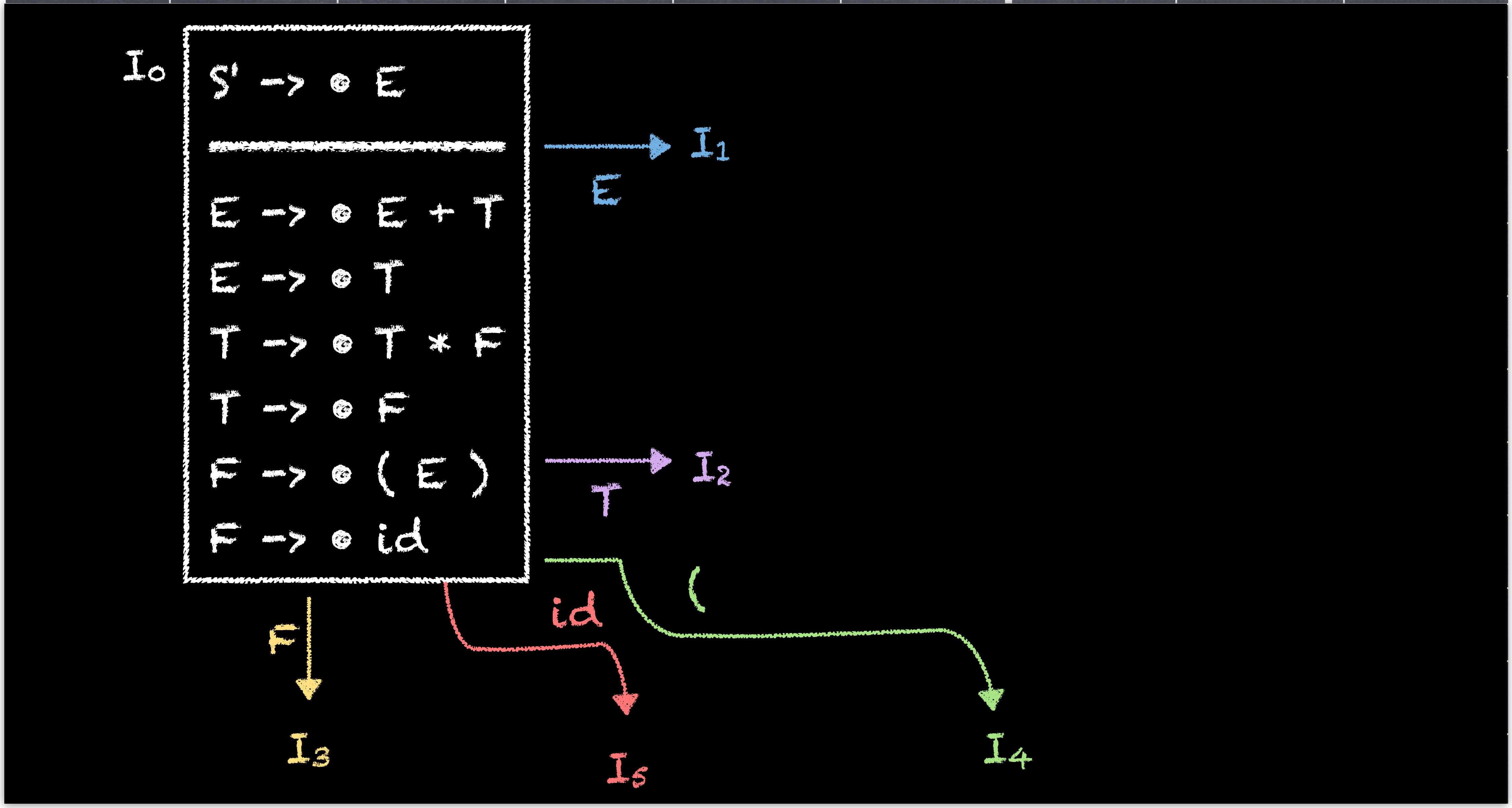
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:

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

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									



I_0

S' -> • E

E -> • E + T
E -> • T
T -> • T * F
T -> • F
F -> • (E)
F -> • id

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5	error	error	s4	error	error	1	2	3
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Implicitly all the empty ACTION cells in a row have 'error' entries.

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

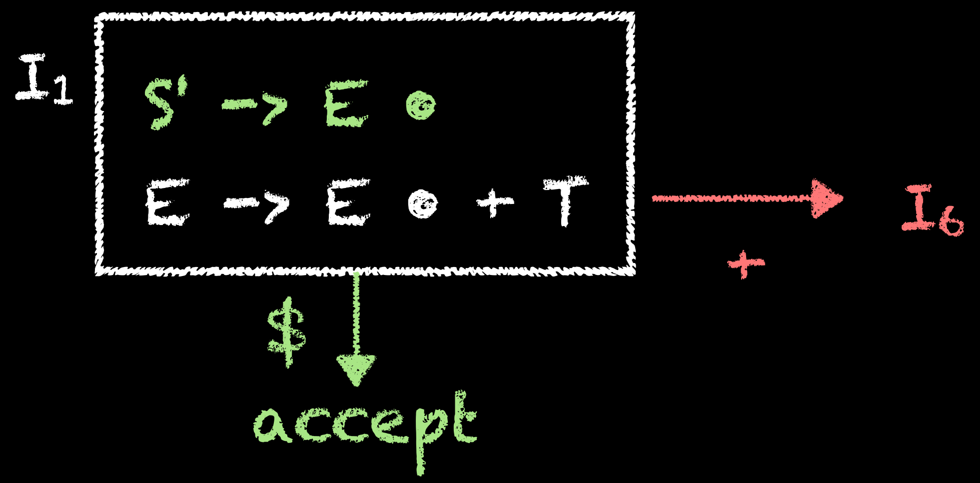


Figure 4.37 [p. 252]

Parsing table for expression grammar

Production numbers:

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

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3									
4									
5									
6									
7									
8									
9									
10									
11									

X	FIRST(X)	FOLLOW(X)
S'	id, (	$
E	id, (	+,), $
T	id, (	*, +,), $
F	id, (	*, +,), $
id	id	*, +,), $
(	(	id, (
)	)	*, +,), $
+	+	id, (
*	*	id, (

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:

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

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4									
5									
6									
7									
8									
9									
10									
11									

I_3 T -> F •

X	FIRST(X)	FOLLOW(X)
S'	id, (	$
E	id, (	+,), $
T	id, (	*, +,), $
F	id, (	*, +,), $
id	id	*, +,), $
(	(	id, (
)	)	*, +,), $
+	+	id, (
*	*	id, (

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4									
5									
6									
7									
8									
9									
10									
11									

… and so on …

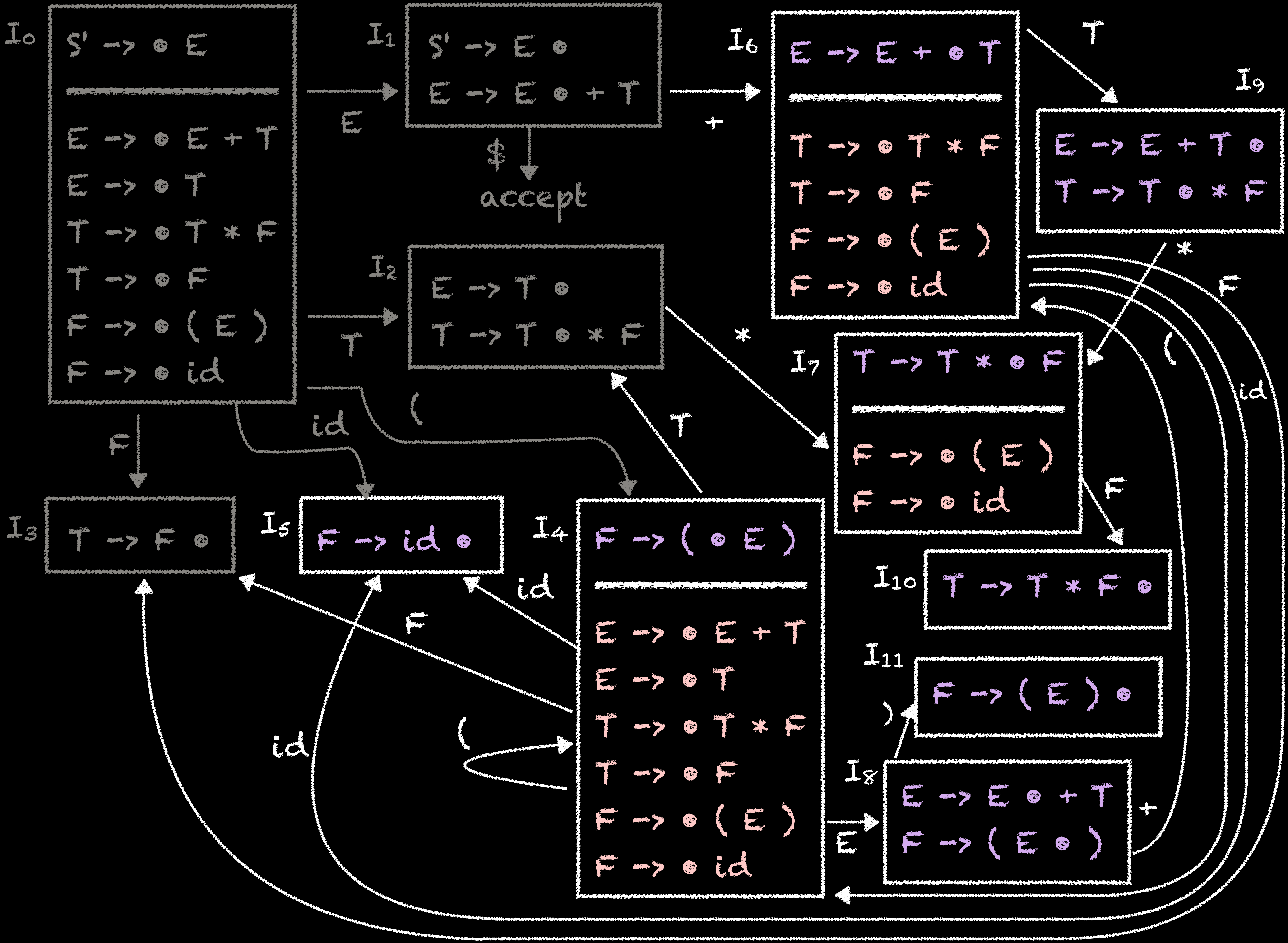

I0
S' -> • E
E -> • E + T
E -> • T
T -> • T * F
T -> • F
F -> • (E)
F -> • id
E
I1
S' -> E •
E -> E • + T
$
accept
+
I6
E -> E + • T
T -> • T * F
T -> • F
F -> • (E)
F -> • id
T
I9
E -> E + T •
T -> T • * F
*
F
(
id
I2
E -> T •
T -> T • * F
T
*
I7
T -> T * • F
F -> • (E)
F -> • id
F
I10
T -> T * F •
F
I3
T -> F •
id
I5
F -> id •
(
I4
F -> (• E)
E -> • E + T
E -> • T
T -> • T * F
T -> • F
F -> • (E)
F -> • id
T
id
F
id
(
E
I8
E -> E • + T
F -> (E •)
)
I11
F -> (E) •
+

Figure 4.37 [p. 252]
Parsing table for expression grammar

Production numbers:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Algorithm 4.44 [p. 250-251]
The LR-parsing algorithm

- INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G
- OUTPUT: If w is in L(G), the reduction steps of a bottom-up parse for w; otherwise, an error indication
- METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state. The parser then executes the program in Fig. 4.36.

Figure 4.36 [p. 251]

```
let a be the first symbol of w$
while (true) {
   let s be the state on top of the stack
   if (ACTION[s,a] = shift t) {
      push t onto the stack
      let a be the next input symbol
   } else if (ACTION[s,a] = reduce A -> β) {
      pop |β| symbols 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
   else call error-recovery routine
}
```

REVIEW

Figure 4.37 [p. 252]
Parsing table for expression grammar

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

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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 parser configuration

- An LR parser configuration is a pair:

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

  - $s_0 s_1 \ldots s_m$ is the stack (bottom to top)
  - $a_i a_{i+1} \ldots a_n \$$ is the (remaining) input

- Represents the right-sentential form

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

Parsing: id + id * id

Stack	Input	Action	Output
$ 0	id + id * id $		

Parsing: id + id * id

Stack	Input	Action	Output
$ 0	id + id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Parsing: id + id * id

Stack	Input	Action	Output
$ 0	id + id * id $	s5	

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Parsing: id + id * id

Stack	Input	Action	Output
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Parsing: id + id * id

Stack	Input	Action	Output
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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				

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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				

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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				

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $		

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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				

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $	r4	T -> F

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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				

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $	r4	T -> F
$ 0 2	+ id * id $	r2	E -> T

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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

Parsing: id + id * id

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

Stack	Input	Action	
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $	r4	T -> F
$ 0 2	+ id * id $	r2	E -> T

Try to complete the rest on your own!

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Stack	Input	Action	Output
$ 0	id + id * id $	s5	
$ 0 5	+ id * id $	r6	F -> id
$ 0 3	+ id * id $	r4	T -> F
$ 0 2	+ id * id $	r2	E -> T
$ 0 1	+ id * id $	s6	
$ 0 1 6	id * id $	s5	
$ 0 1 6 5	* id $	r6	F -> id
$ 0 1 6 3	* id $	r4	T -> F
$ 0 1 6 9	* id $	s7	
$ 0 1 6 9 7	id $	s5	
$ 0 1 6 9 7 5	$	r6	F -> id
$ 0 1 6 9 7 10	$	r3	T -> T * F
$ 0 1 6 9	$	r1	E -> E + T
$ 0 1	$	accept	

This is the output

F -> id

T -> F

E -> T

F -> id

T -> F

F -> id

T -> T * F

E -> E + T

It is a rightmost derivation in reverse

E -> E + T

T -> T * F

F -> id

T -> F

F -> id

E -> T

T -> F

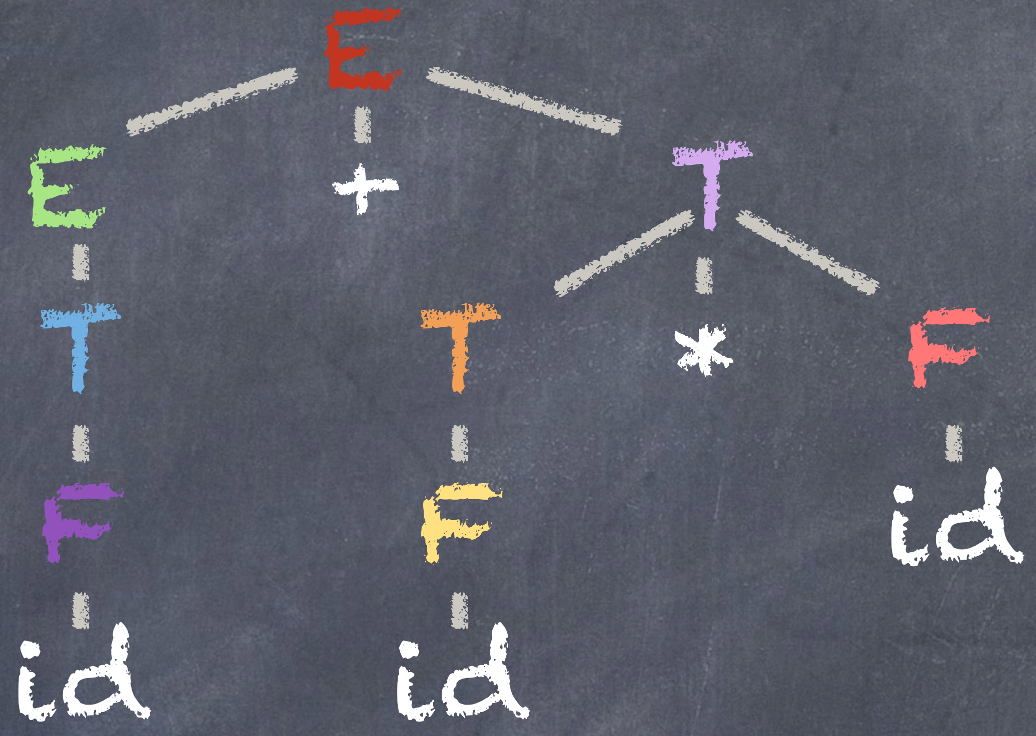
F -> id

Here's the derivation:

E -> E + T	E -> E + T
T -> T * F	-> E + T * F
F -> id	-> E + T * id
T -> F	-> E + F * id
F -> id	-> E + id * id
E -> T	-> T + id * id
T -> F	-> F + id * id
F -> id	-> id + id * id

and the corresponding parse tree:

E -> E + T
-> E + T * F
-> E + T * id
-> E + F * id
-> E + id * id
-> T + id * id
-> F + id * id
-> id + id * id



For Wednesday

- Class will focus on Sprint 2:
  - structure of Bison's .y file
  - yylex and yyparse
  - the union
  - symbol tables (read esp. section 2.7.1)
  - general advice