LR Parsing Techniques

1

Outline

- Introduction
- Shift-Reduce Parsers
- LR Parsers

Introduction(1)

- Overview
  - **Top-down parsers**
    - starts constructing the parse tree at the top (root) of the tree and move down towards the leaves.
    - Easy to implement by hand, but work with restricted grammars.
    - example: predictive parsers
  - **Bottom-up parsers**
    - build the nodes on the bottom of the parse tree first.
    - Suitable for automatic parser generation, handle a larger class of grammars.
    - examples: shift-reduce parser (or LR (k) parsers)

Introduction(2)

- **Bottom-up parsers**
  - A bottom-up parser, or a shift-reduce parser, begins at the leaves and works up to the top of the tree.
  - The reduction steps trace a rightmost derivation on

More Example at Next Page to explain it.

Grammar

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

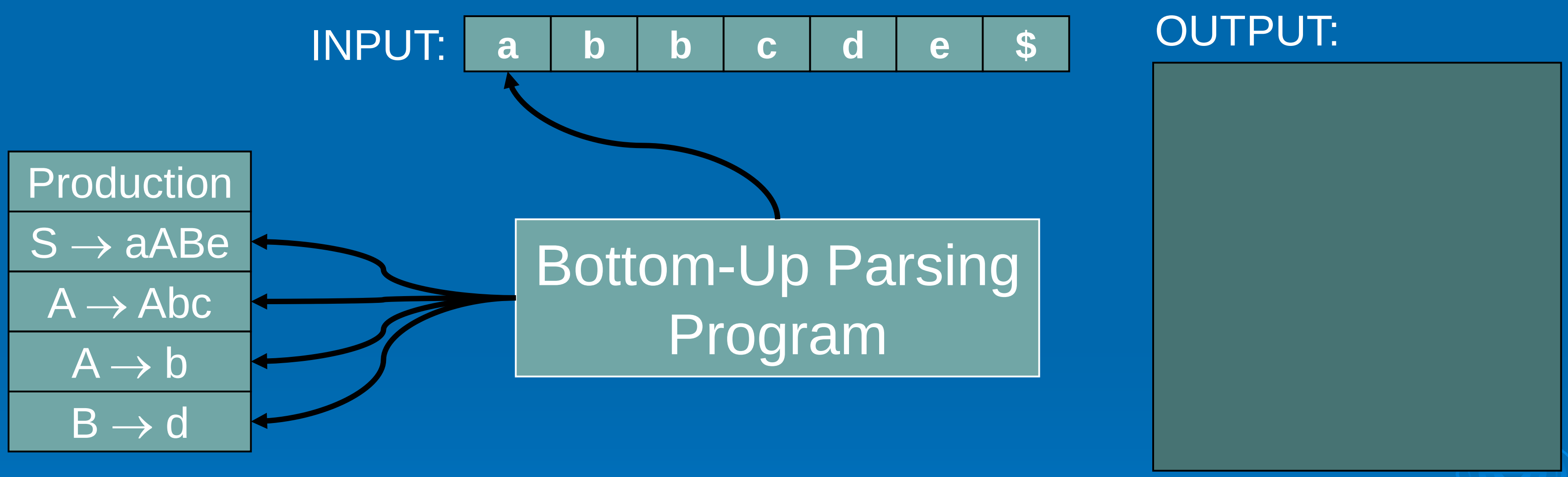
parse

The input string : abbcde.

Introduction(3)

Bottom-Up Parser Example

Shift a

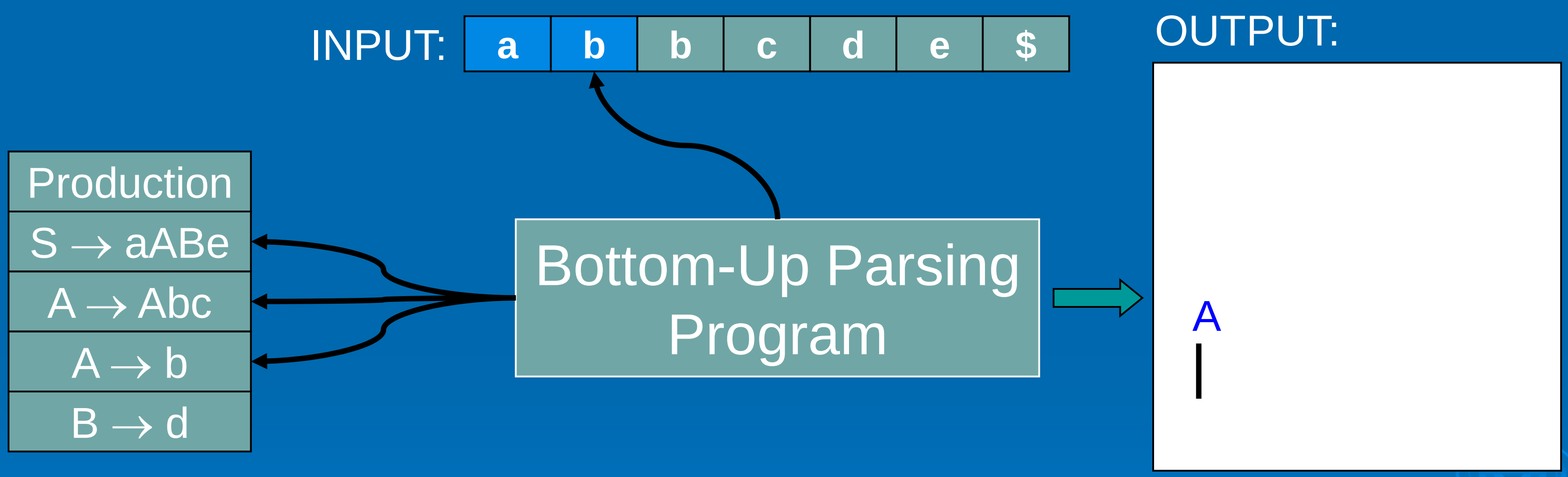


Introduction(4)

Bottom-Up Parser Example

Shift b

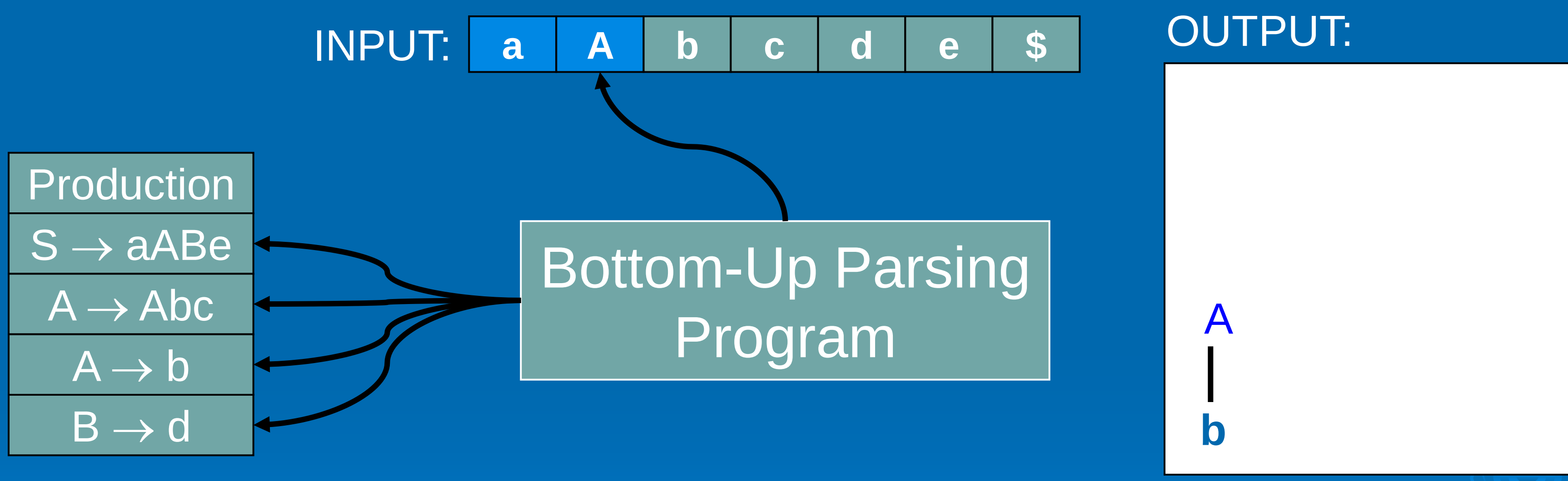
Reduce from b to A



Introduction(5)

Bottom-Up Parser Example

Shift A

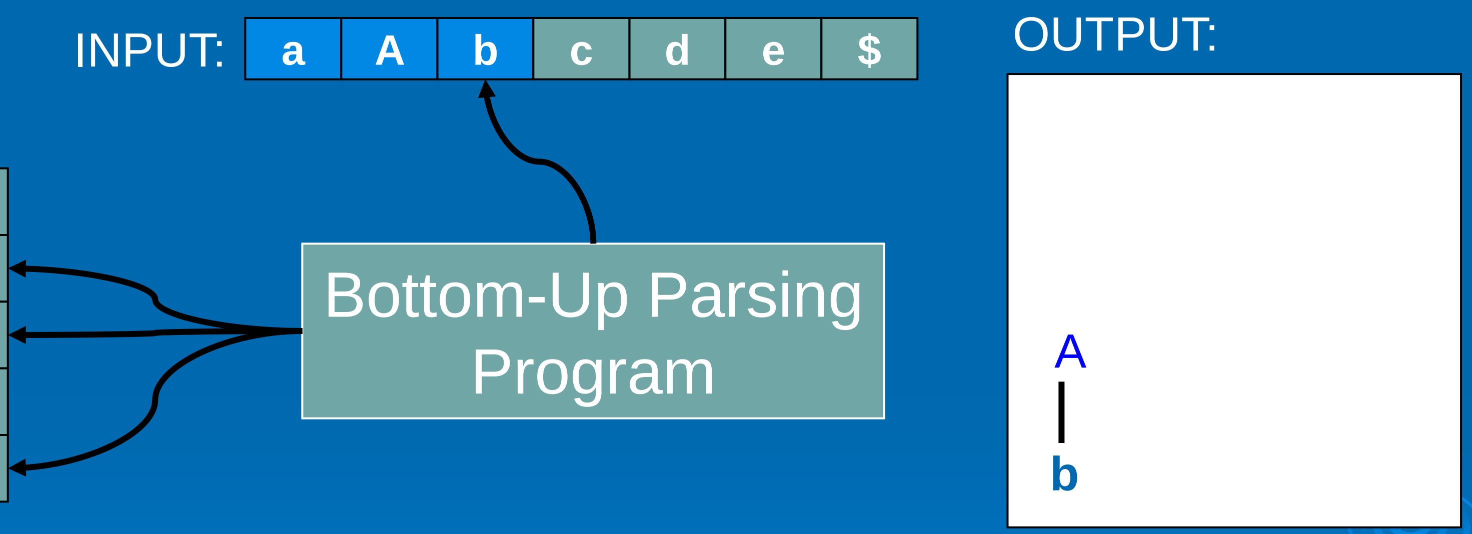


Introduction(6)

Bottom-Up Parser Example

Shift b

Production
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

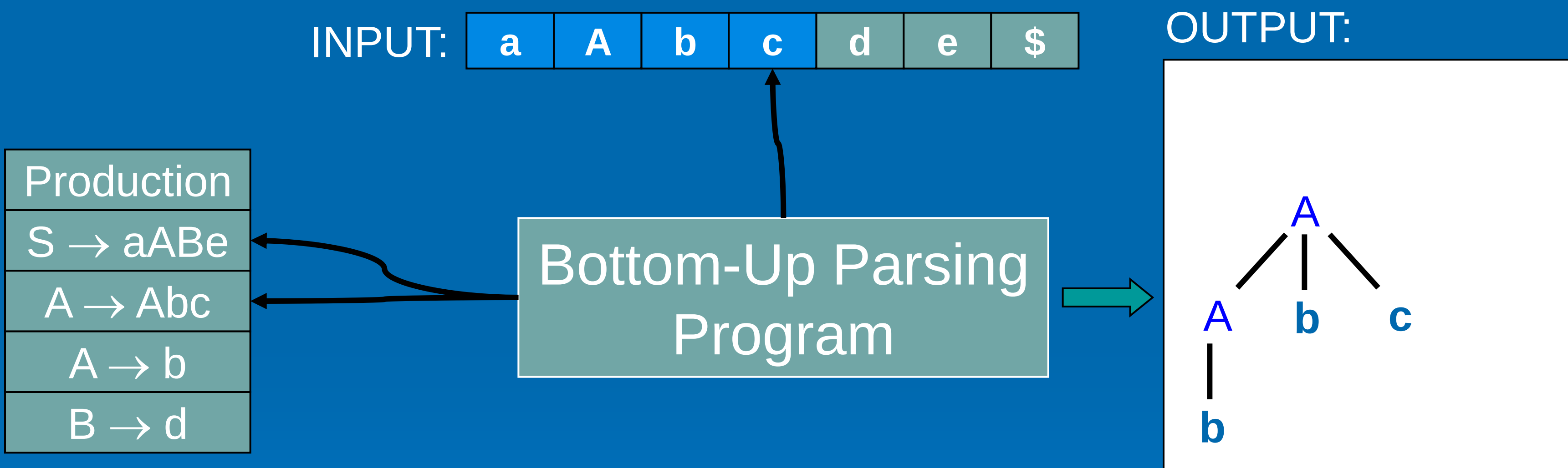


Introduction(7)

Bottom-Up Parser Example

Shift c

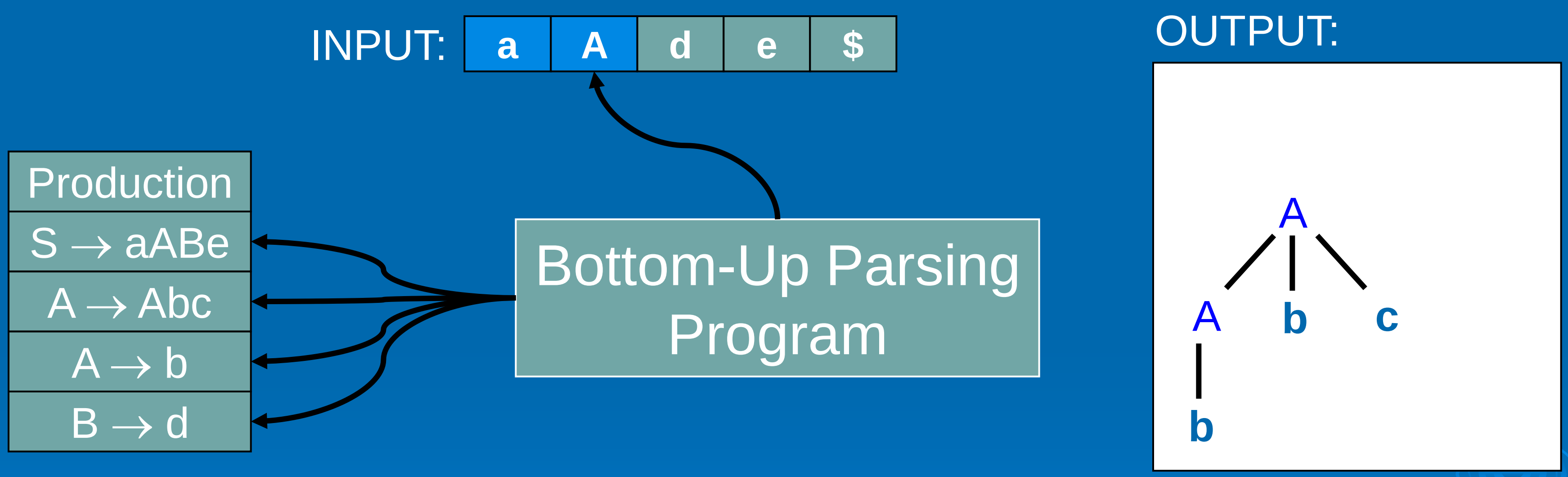
Reduce from Abc to A



Introduction(8)

Bottom-Up Parser Example

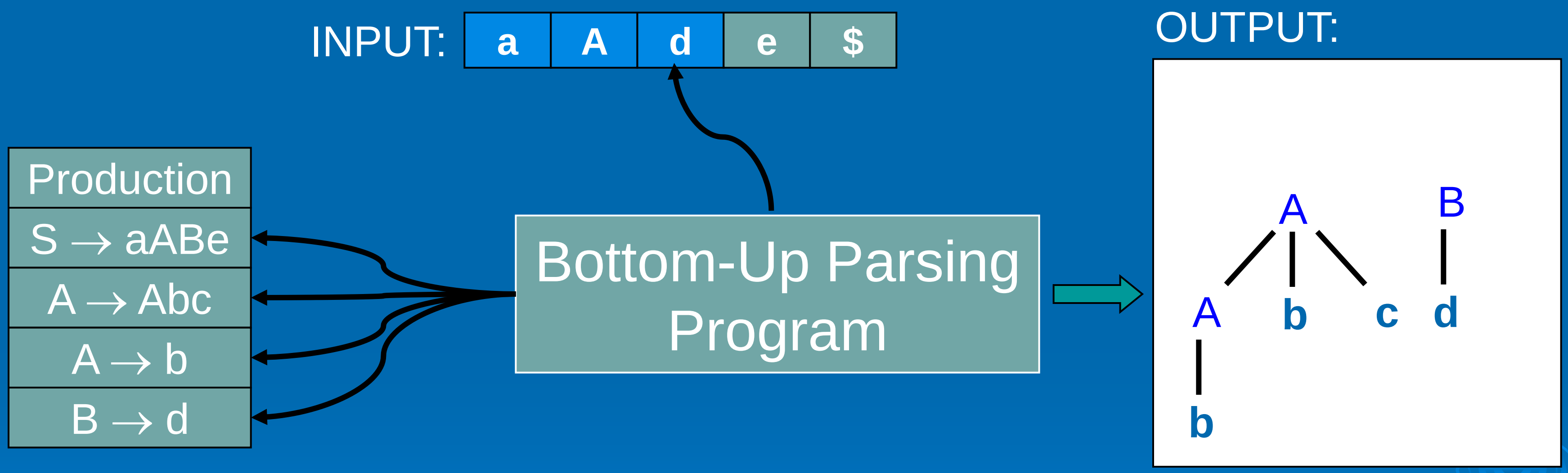
Shift A



Introduction(9)

Bottom-Up Parser Example

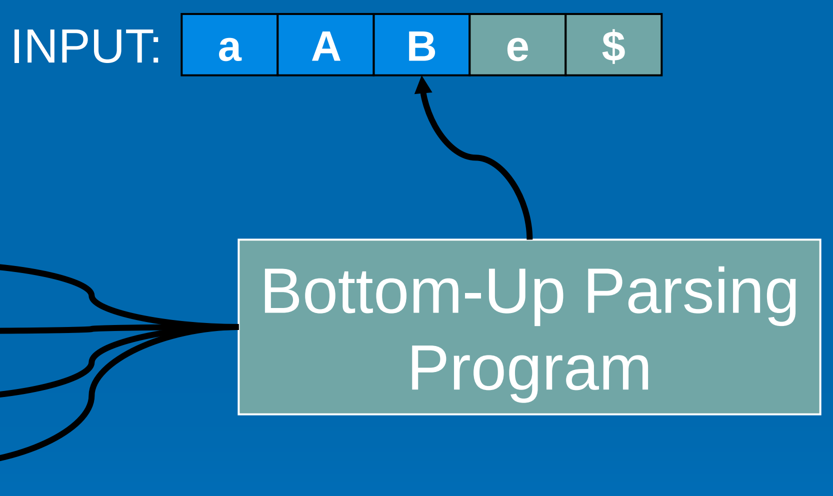
Shift d
Reduce from d to B



Introduction(10)

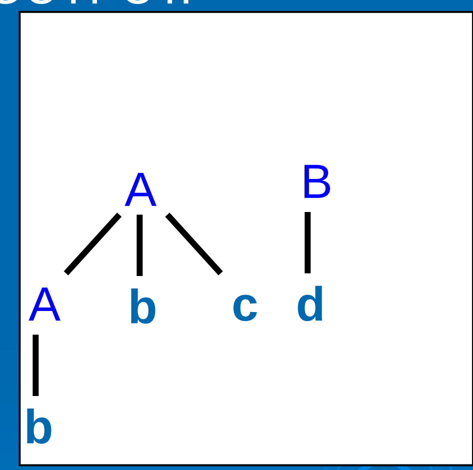
Bottom-Up Parser Example

Shift B



Production
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

OUTPUT:



Introduction(11)

Bottom-Up Parser Example

Shift e
Reduce from aABe to S

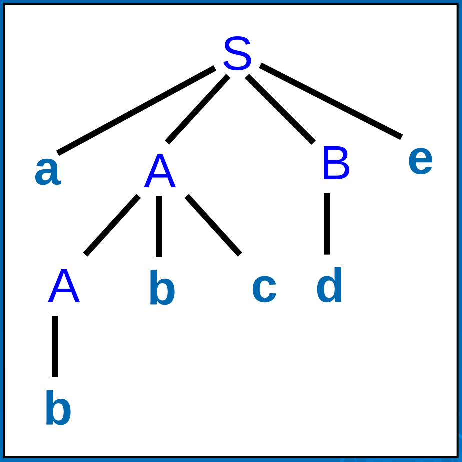
INPUT:

a	A	B	e	$

Production
S → aABe
A → Abc
A → b
B → d

Bottom-Up Parsing Program

OUTPUT:



Introduction(12)

Bottom-Up Parser Example

Shift S

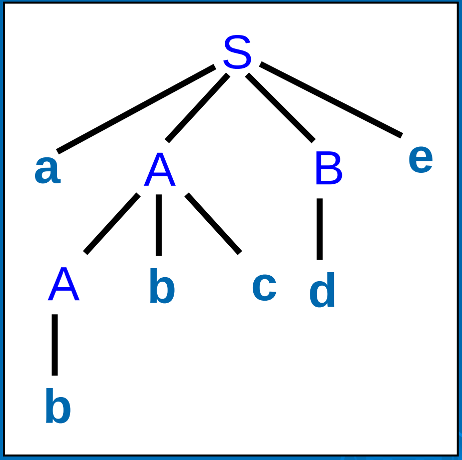
Hit the target $

INPUT: | S | $ |

Production
S → aABe
A → Abc
A → b
B → d

Bottom-Up Parsing Program

OUTPUT:



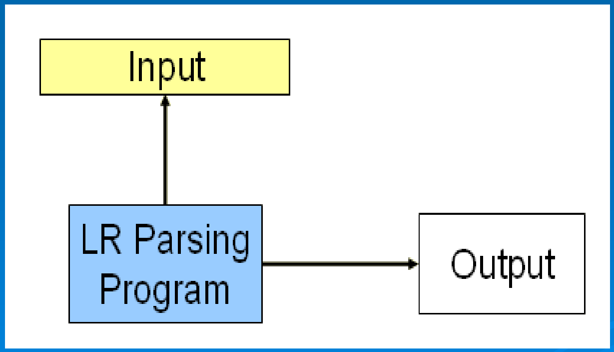
This parser is known as an LR Parser because it scans the input from Left to right, and it constructs a Rightmost derivation in reverse order.

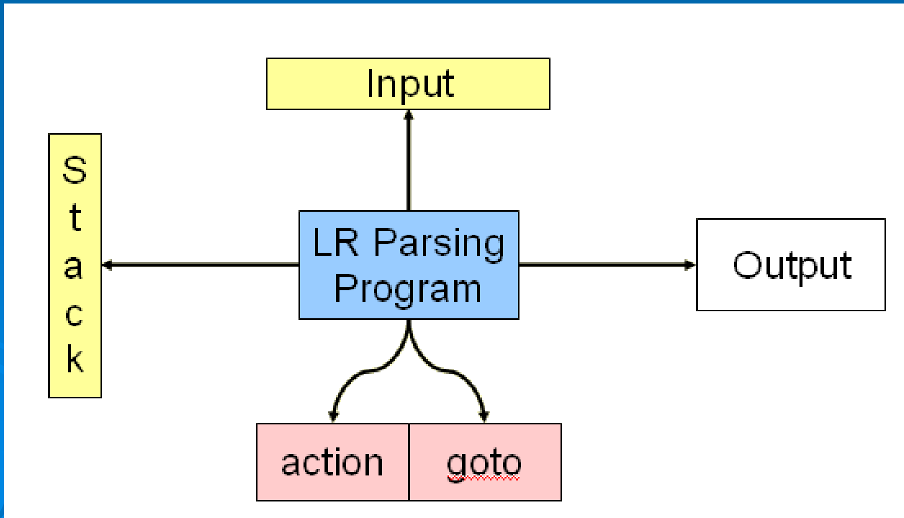
Introduction(13)

- Conclusion
  - The scanning of productions for matching with handles in the input string
  - Backtracking makes the method used in the previous example very inefficient.

Can we do better? Discuss in later!!!

Previous Architecture → ***Renew Architecture***





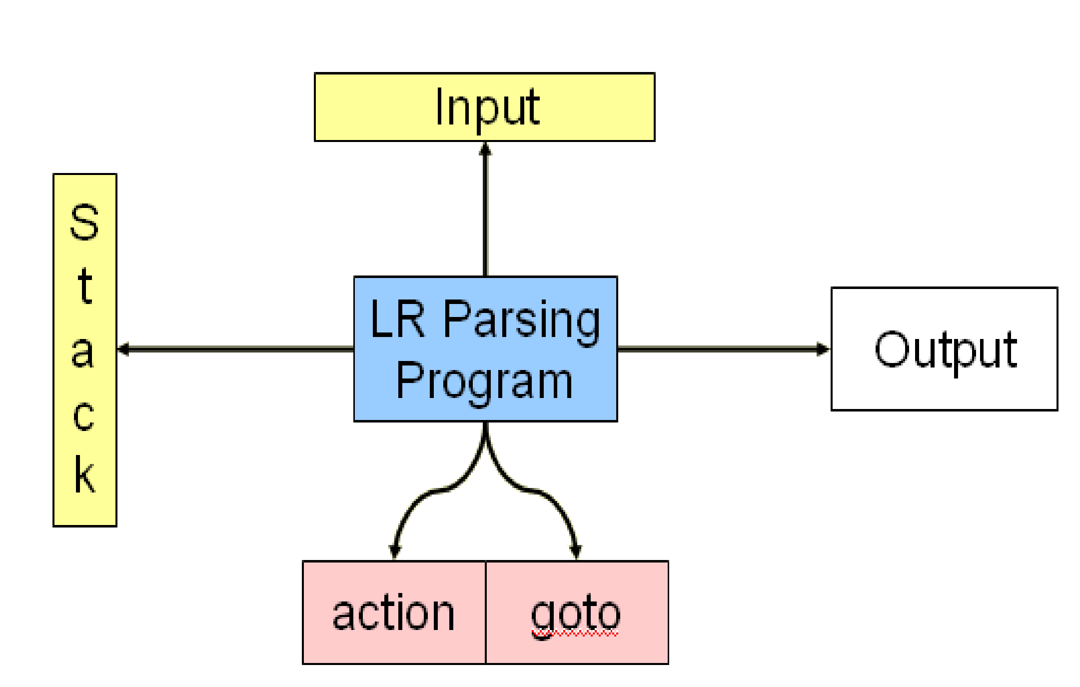
Outline

- Introduction
- Shift-Reduce Parsers
- LR Parsers

16

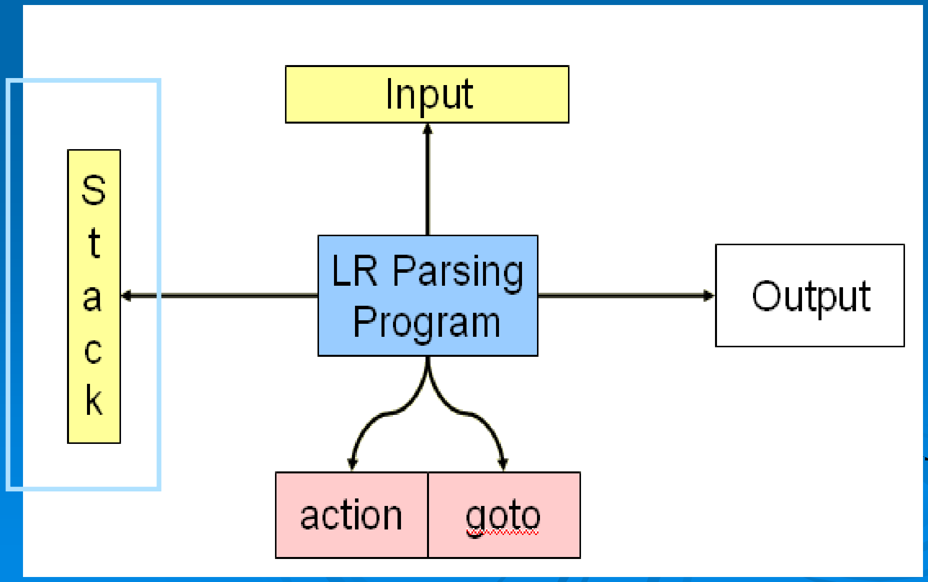
Shift-Reduce Parsers(1)

- Shift-Reduce (bottom-up) parser is known as an LR Parser
    - It scans the input from Left to right
    - Rightmost derivation in reverse order
- Kinds of LR
    - LR(k)
        - most powerful deterministic bottom-up parsing using k lookaheads
    - SLR(k)
    - LALR(k)
- mechanism to perform bottom-up parsing
    - finite state machine
    to manipulate “handle”
- Components
    - Parse stack
    - Shift-reduce driver
    - Action table
    - Goto table



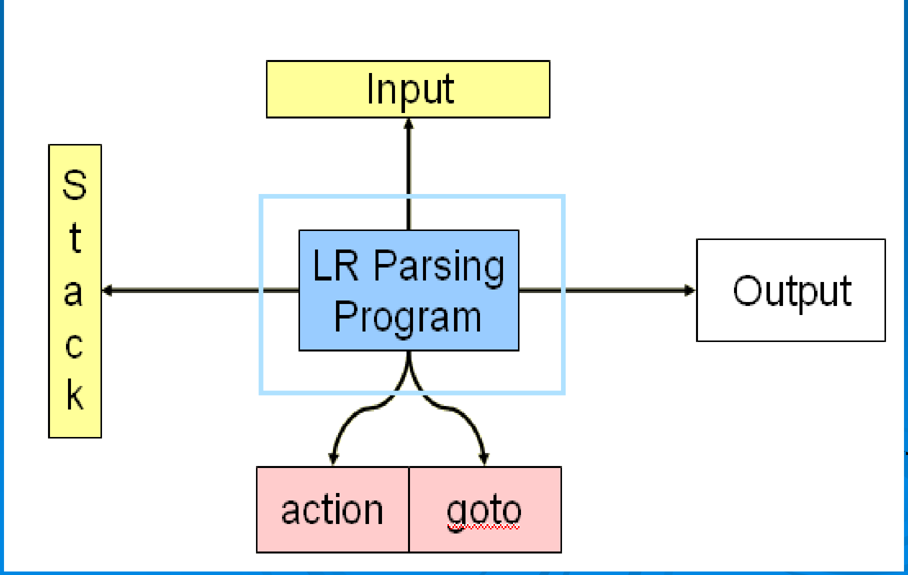
Shift-Reduce Parsers(2)

- Parse stack
  - Initially empty, contains symbols already parsed
    - Elements in the stack are terminal or non-terminal symbols
  - The parse stack catenated with the remaining input always represents a right sentential form



Shift-Reduce Parsers(3)

- Shift-Reduce driver
  - Shift -- when top of stack doesn't contain a handle of the sentential form
    - push input token (with contextual information) into stack
  - Reduce -- when top of stack contains a handle
    - pop the handle
    - push reduced non-terminal (with contextual information)

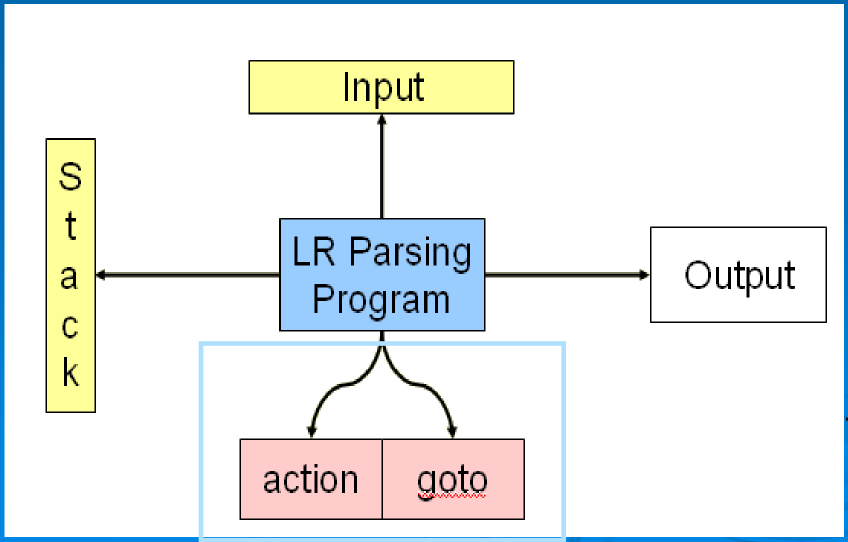


Shift-Reduce Parsers(4)

- Two questions
  - Have we reached the end of handles and how long is the handle?
  - Which non-terminal does the handle reduce to?
- We use tables to answer the questions
  - ACTION table
  - GOTO table

Shift-Reduce Parsers(5)

- LR parsers are driven by two tables:
  - Action table, which specifies that actions to take
    - **Shift, reduce, accept or error**
  - Goto table, which specifies state transition
    - To indicate transition of finite state machine
  - We push states, rather than symbols onto the stack
  - Each state represents the possible sub-trees of the parse tree



Shift-Reduce Parsers(6)

- grammar G_0

  1. <program> → begin <stmts> end $
  2. <stmts> → SimpleStmt ; <stmts>
  3. <stmts> → begin <stmts> end ; <stmts>
  4. <stmts> → λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								
<program>												
<stmts>		S			S		S			S		

Figure 6.2 A Shift-Reduce **action** Table for G_0

Action Table

blank -- ERROR

Goto Table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce **go_to** Table for G_0

Shift-Reduce Parsers(7)

```
void shift_reduce_driver(void)
{
    /* Push the Start State, S0,
     * onto an empty parse stack. */
    push(S0);
    while (TRUE) {              /* forever */
    /* Let S be the top parse stack state;
     * let T be the current input token.*/
        switch (action[S][T]) {
        case ERROR:
            announce_syntax_error();
            break;
        case ACCEPT:
        /* The input has been correctly
         * parsed. */
            clean_up_and_finish();
            return;
        case SHIFT:
            push(go_to[S][T]);
            scanner(&T);
            /* Get next token. */
            break;
        case REDUCEi:
    /* Assume i-th production is
     * X → Y1 · · · Ym.
     * Remove states corresponding to
     * the RHS of the production. */
            pop(m);
            /* S' is the new stack top. */
            push(go_to[S'][X]);
            break;
        }
    }
}
```

Shift-Reduce Parsers(8)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stm
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(1)	0	begin SimpleStmt ; SimpleStmt ; end $	Shift 1

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(9)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program> → begin<stmts>end$
  2. <stmts> → SimpleStmt;<stmts>
  3. <stmts> → begin<stmts>end;<stmts>
  4. <stmts> → λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(2)	0,1	SimpleStmt ; SimpleStmt ; end $	Shift 5

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(10)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>	.	2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(3)	0,1,5	; SimpleStmt ; end $	Shift 6

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(11)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts>
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(4)	0,1,5,6	SimpleStmt ; end $	Shift 5

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(12)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce **go_to** Table for G_0

- grammar G_0
  1. <program> → begin<stmts>end$
  2. <stmts> → SimpleStmt;<stmts>
  3. <stmts> → begin<stmts>end;<stmts>
  4. <stmts> → λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(5)	0,1,5,6,5	; end $	Shift 6

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(13)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce go_to Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts>
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(6)	0,1,5,6,5,6,λ	end $ /* goto(6,<stmts>) = 10 */	Reduce 4

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

<program>												
<stmts>		2			7		10			11		

Shift-Reduce Parsers(14)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program> → begin<stmts>end$
  2. <stmts> → SimpleStmt;<stmts>
  3. <stmts> → begin<stmts>end;<stmts>
  4. <stmts> → λ
- tracing steps

Step	Parse Stack	Remaining Input		Action
(7)	0,1,5,6,*5,6,10*	end $	/* goto(6,<stmts>) = 10 */	Reduce 2

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

<program>												
<stmts>		2			7		10			11		

Shift-Reduce Parsers(15)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts>
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(8)	0,1,*5,6,10*	end $ /* goto(1,<stmts>) = 2 */	Reduce 2

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

Symbol	0	1	2	3	4	5	6	7	8	9	10	11
<program>												
<stmts>		2			7		10			11		

Shift-Reduce Parsers(16)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce go_to Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts>
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(9)	0,1,2	end $	Shift 3

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(17)

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>	.	2			7		10			11		

Figure 6.3 A Shift-Reduce **go_to** Table for G_0

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<stmts>
  4. <stmts> →λ
- tracing steps

Step	Parse Stack	Remaining Input	Action
(10)	0,1,2,3	$	Accept

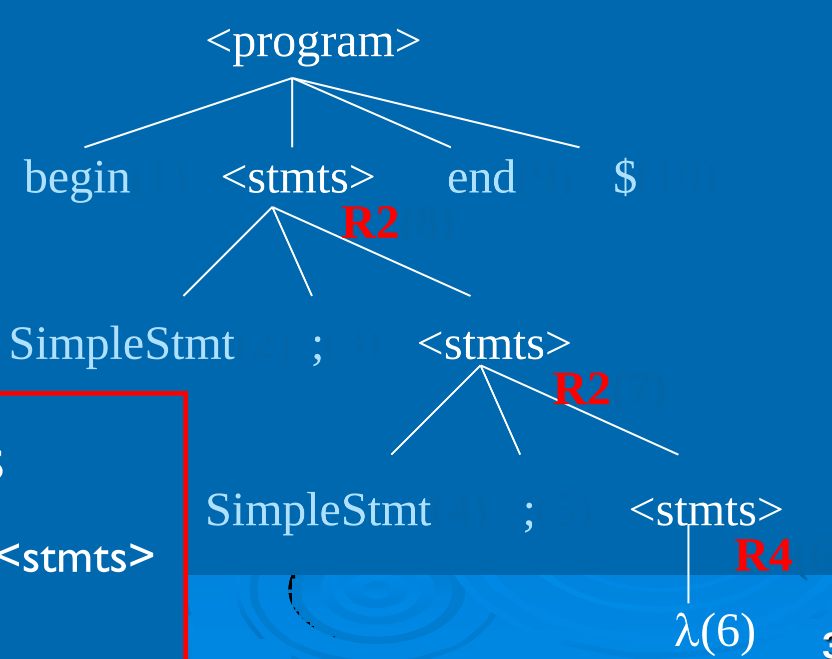
action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Shift-Reduce Parsers(18)

- **tracing steps**

Step	Parse Stack	Remaining Input	Action
(1)	0	begin SimpleStmt ; SimpleStmt ; end $	Shift 1
(2)	0,1	SimpleStmt ; SimpleStmt ; end $	Shift 5
(3)	0,1,5	; SimpleStmt ; end $	Shift 6
(4)	0,1,5,6	SimpleStmt ; end $	Shift 5
(5)	0,1,5,6,5	; end $	Shift 6
(6)	0,1,5,6,5,6	end $ /* goto(6,<stmts>) = 10 */	Reduce 4
(7)	0,1,5,6,*5,6,10*	end $ /* goto(6,<stmts>) = 10 */	Reduce 2
(8)	0,1,*5,6,10*	end $ /* goto(1,<stmts>) = 2 */	Reduce 2
(9)	0,1,2	end $	Shift 3
(10)	0,1,2,3	$	Accept



- grammar G_0
  1. <program> → begin <stmts> end $
  2. <stmts> → SimpleStmt ; <stmts>
  3. <stmts> → begin <stmts> end ; <stmts>
  4. <stmts> → λ

Outline

- Introduction
- Shift-Reduce Parsers
- LR Parsers

35

LR Parsers

- LR(n)   n=0~k
  - Read from Left, Right-most derivation, n look-ahead
- LR parsers are deterministic
  - No backup or retry parsing actions
- LR(0):
  - Without prediction read from Left, Right-most derivation, 0 look-ahead
- LR(1):
  - 1-token look-ahead
  - General
- LR(k) parsers
  - Decide the next action by examining the tokens already shifted and at most k look-ahead tokens
  - The most powerful of deterministic
  - Difficult to implement

LR(0) Table Construction(1)

- A production has the form
  - $A \rightarrow X_1 X_2 \ldots X_j$
- By adding a dot, we get a configuration (or an item)
  - $A \rightarrow \bullet X_1 X_2 \ldots X_j$
  - $A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j$
  - $A \rightarrow X_1 X_2 \ldots X_j \bullet$
- The • indicates how much of a RHS has been shifted into the stack.
- An item with the • at the end of the RHS
  - $A \rightarrow X_1 X_2 \ldots X_j \bullet$
  - indicates (or recognized) that RHS should be reduced to LHS
- An item with the • at the beginning of RHS
  - $A \rightarrow \bullet X_1 X_2 \ldots X_j$
  - predicts that RHS will be shifted into the stack

LR(0) Table Construction(2)

- An LR(0) state is a set of configurations
  - This means that the actual state of LR(0) parsers is denoted by one of the items.
- The closure0 operation:
  - if there is an configuration **B→δ • A** ρ in the set then add all configurations of the form
- The initial configuration
  - **s0 = closure0({S→ • α $}**

EX: for grammar G_1 :

1. S'→S$
2. S→ID|λ

closure0({ S → • S $ }) =

{ S' → • S$,

S → • ID,

S → λ • }

special case: λ

```
Configuration_set closure (configuration_set s)
{
  configuration_set s' = s ;
  do
  {
     if( B →δ • A ρ ∈ s' for A ∈ Vn )
     {
       /*
        * Predict productions with A
        * as the left-hand side
        */
         Add all configurations of the form
         A → • γ to s'
      }
  } while (more new configurations can be added) ;
  return 0;
}
```

LR(0) Table Construction(3)

- Q1: Why the grammar use $S' \rightarrow S\$$?
  - Ans: Easy to check the ending of parser!

EX: If S' is not exist~

$S \rightarrow ID\$$

$S \rightarrow \lambda\$$

When we button up to reduce the original symbol S, there are two paths to achieve it.

Multipath is a problem that if we have complex grammars like C. A lot of paths we need to check the ending symbol $.

EX: for grammar G_1 :

1. $S' \rightarrow S\$$
2. $S \rightarrow ID | \lambda$

closure0({ $S \rightarrow \bullet S \$$ }) =

{ $S' \rightarrow \bullet S\$$,

$S \rightarrow \bullet ID$,

$S \rightarrow \lambda \bullet$ }

LR(0) Table Construction(4)

- Given a *configuration set* **s**, we can compute its successor, **s'** , under a symbol **X**
  - Denoted go_to0(s,X)=s'

```
Configuration_set goto (configuration_set s , symbol x)
{
  Sb = Ø ;
  for (each configuration c ∈ S)
    if( each configuration c ∈ S)
      Add A → βx • γ to sb ;
  /*
   * That is, we advance the • past the symbol X,
   * if possible. Configurations not having a
   * dot preceding an X are not included in sb .
   */

   /* Add new predictions to  sb via closure0. */
  return closure0(sb) ;
}
```

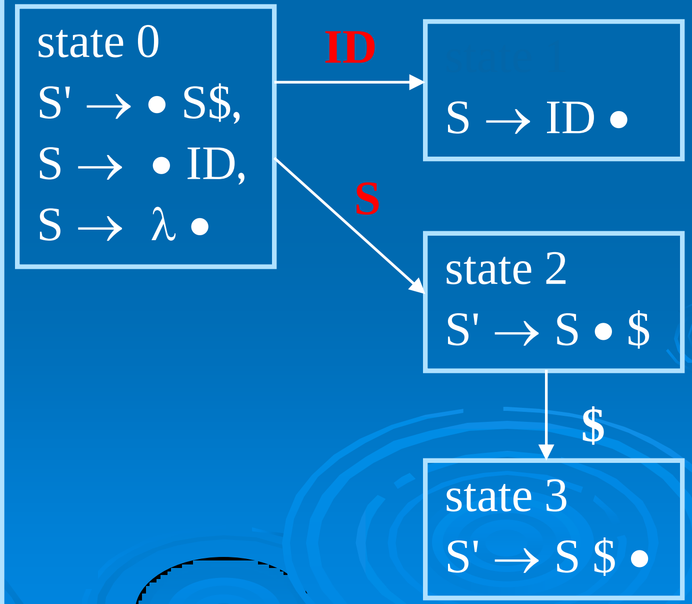
LR(0) Table Construction(5)

- Characteristic finite state machine (**CFSM**)
  - It is a finite automaton
  - Identifying configuration sets and successor operation with CFSM states and transitions

```
void_build_CFSM(void)
{
  S = SET_OF(S0);
  while (S is nonempty) {
    Remove a configuration set s from S;
  /* Consider both terminals and non-terminals */
    for ( X in Symbols) {
      if(go_to0(s,X) does not label a CFSM state) {
        Create a new CFSM state and label it
          with go_to0(s , X) into S;
      }
      Create a transition under X from the state s
          labels to the state go_to0(s , X)
    }
  }
}
```

EX: for grammar G_1 :

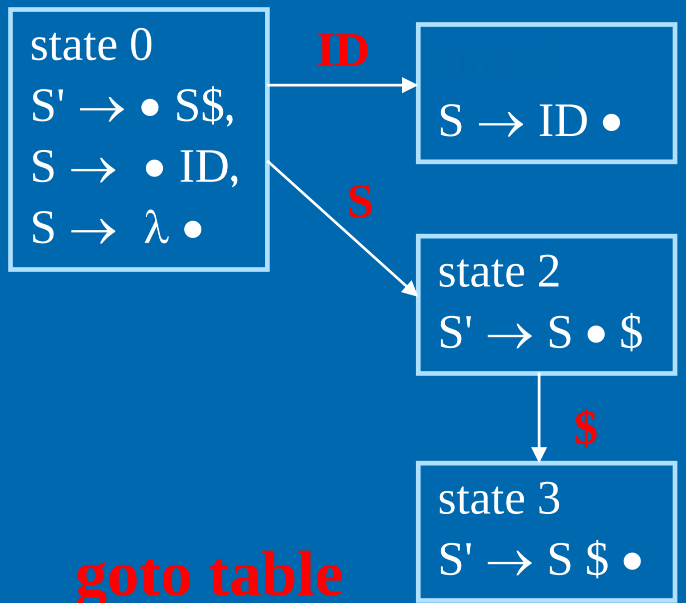
1. $S' \rightarrow S\$$
2. $S \rightarrow ID | \lambda$



LR(0) Table Construction(6)

- CFSM is the goto table of LR(0) parsers.

```
Int ** build_go_to_table(finite_automation CFSM)
{
  const int N = num_states (CFSM);
  int **tab;
  Dynamically allocate a table of dimension
    N × num_symbols (CFSM) to represent
    the go_to table and assign it to tab;
  Number the states of CFSM from 0 to N-1,
    with the Start State labeled 0;
  for( S = 0 ; S<=N-1 ; S++) {
  /* Consider both terminals and non-terminals. */
    for ( X in Symbols) {
      if ( State S has a transition under X to some state T)
          tab [S][X] = T ;
        else
          tab [S][X] = EMPTY;
    }
   }
  return tab;
}
```



goto table

State	Symbol		
	ID	$	S
0	1	4	2
1	4	4	4
2	4	3	4
3	4	4	4
4			

LR(0) Table Construction(7)

- Because LR(0) uses no look-ahead, we must extract the action function directly from the configuration sets of CFSM
- Let Q={Shift, Reduce_1, Reduce_2 , …, Reduce_n}
  - There are n productions in the CFG
- S_0 be the set of CFSM states
  - $P:S_0 \rightarrow 2^Q$
- $P(s)=\{\text{Reduce}_i \mid B\rightarrow\rho \bullet \in s$ and production i is $B\rightarrow\rho \} \cup$ (if $A\rightarrow \alpha\bullet a\beta \in s$ for $a \in V_t$ Then {Shift} Else $\emptyset$)

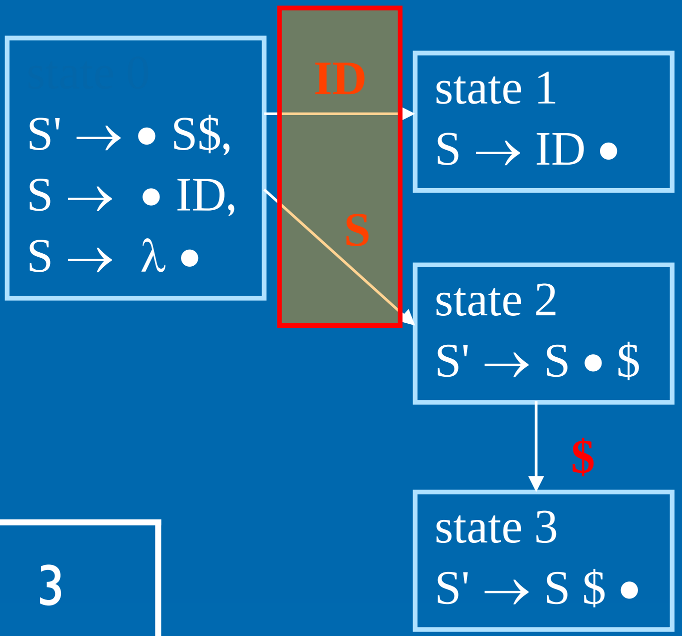
LR(0) Table Construction(8)

- G is LR(0) if and only if $\forall\ s \in S_0\ |P(s)|=1$
- If G is LR(0), the action table is trivially extracted from P
  - $P(s)=\{Shift\} \Rightarrow action[s]=Shift$
  - $P(s)=\{Reduce_j\}$, where production j is the augmenting production, $\Rightarrow action[s]=Accept$
  - $P(s)=\{Reduce_i\}$, $i \neq j$, $action[s]=Reduce_i$
  - $P(s)=\varnothing \Rightarrow action[s]=Error$

LR(0) Table Construction(9)

EX: for grammar G_1 :

1. $S' \rightarrow S\$$
2. $S \rightarrow ID | \lambda$

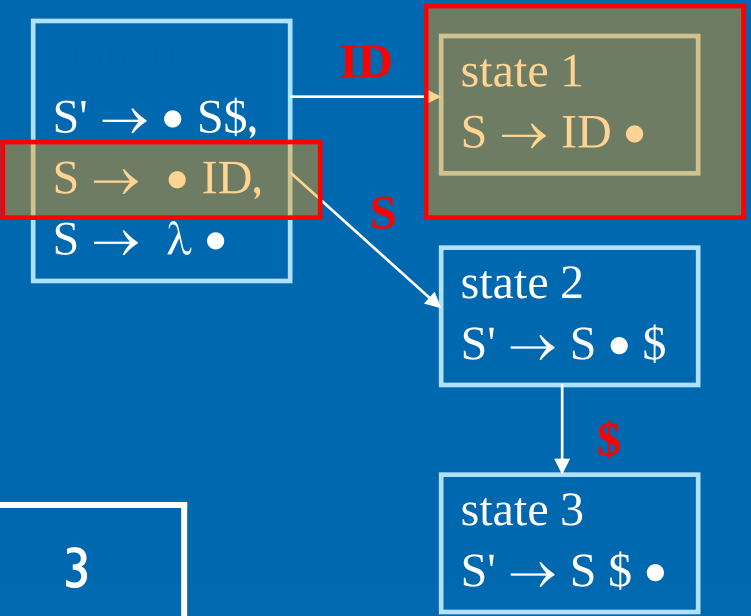


state	0	1	2	3
action	S	R2	S	Accept

LR(0) Table Construction(10)

EX: for grammar G_1 :

1. $S' \to S\$$
2. $S \to ID | \lambda$

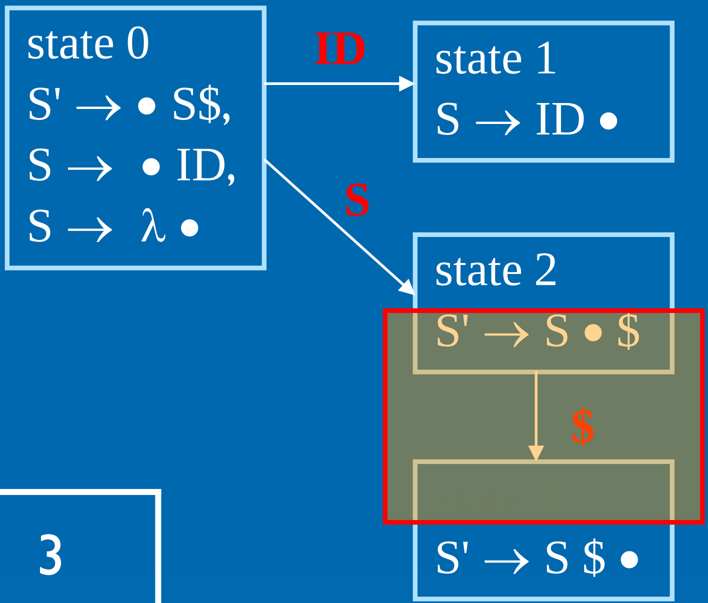


state	0	1	2	3
action	S	R2	S	Accept

LR(0) Table Construction(11)

EX: for grammar G_1 :

1. $S' \rightarrow S\$$
2. $S \rightarrow ID | \lambda$

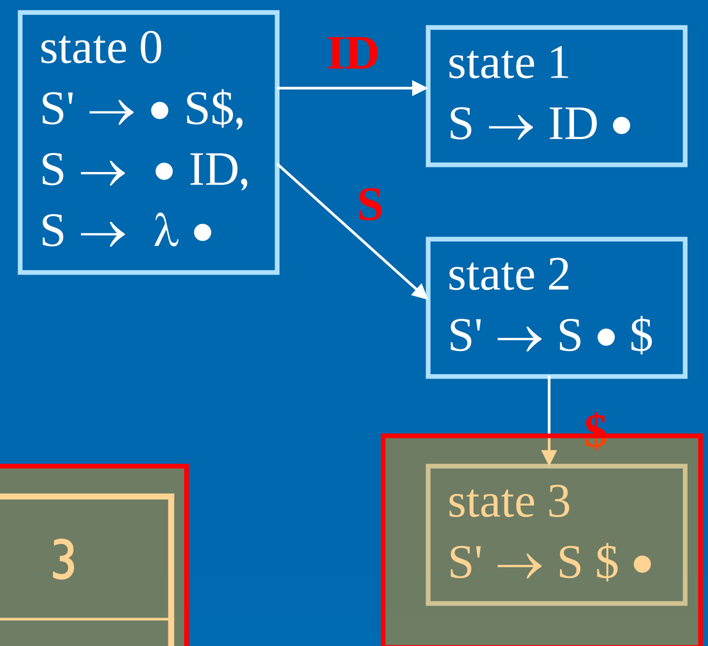


state	0	1	2	3
action	S	R2	S	Accept

LR(0) Table Construction(12)

EX: for grammar G_1 :

1. $S' \rightarrow S\$$
2. $S \rightarrow ID | \lambda$



state	0	1	2	3
action	S	R2	S	Accept

LR(0) Table Construction(13)

- Any state $s \in S_0$ for which $|P(s)|>1$ is said to be *inadequate*
- Two kinds of parser conflicts create inadequacies in configuration sets
  - Shift-reduce conflicts
  - Reduce-reduce conflicts
- If is easy to introduce inadequacies in CFSM states
  - Hence, few real grammars are LR(0). For example,
    - Consider λ-productions
      - The only possible configuration involving a λ-production is of the form $A \rightarrow \lambda \bullet$
      - However, if A can generate any terminal string other than λ, then a shift action must also be possible (First(A))
    - LR(0) parser will have problems in handling operator precedence properly

LR(0) Tracing Example(0)

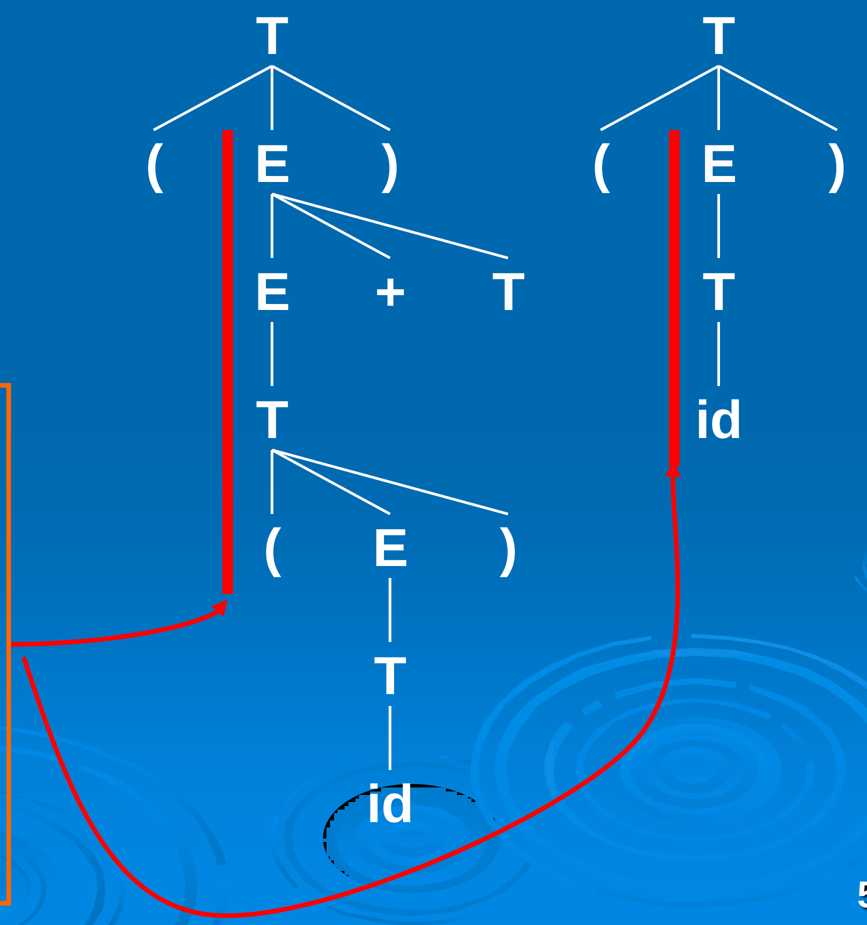
- Before tracing , we will need to know the mind of CFSM

for grammar G_2 :

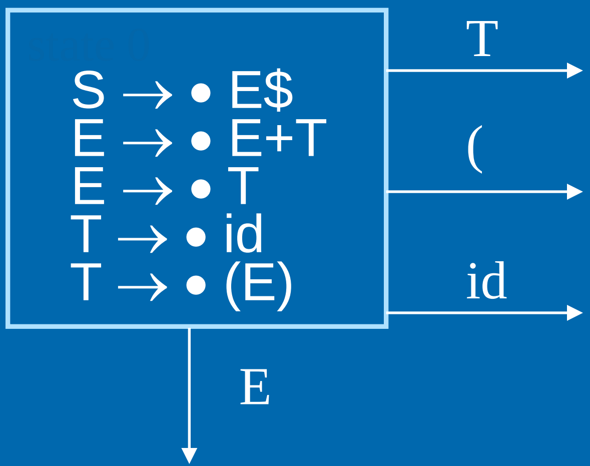
1. $S \rightarrow E\$$
2. $E \rightarrow E+T$
3. $E \rightarrow T$
4. $T \rightarrow id$
5. $T \rightarrow (E)$

- closure0({ T →(• E) }
- = { T →(• E) ,
  E →• E + T ,
  E →• T ,
  T →• id ,
  T →• (E) }

When shift (, some possible answers of tree:

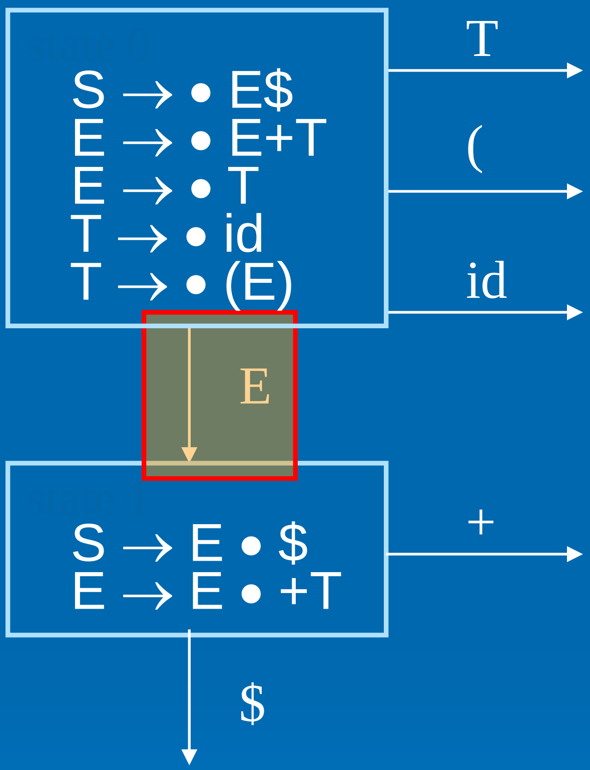


LR(0) Tracing Example(1)



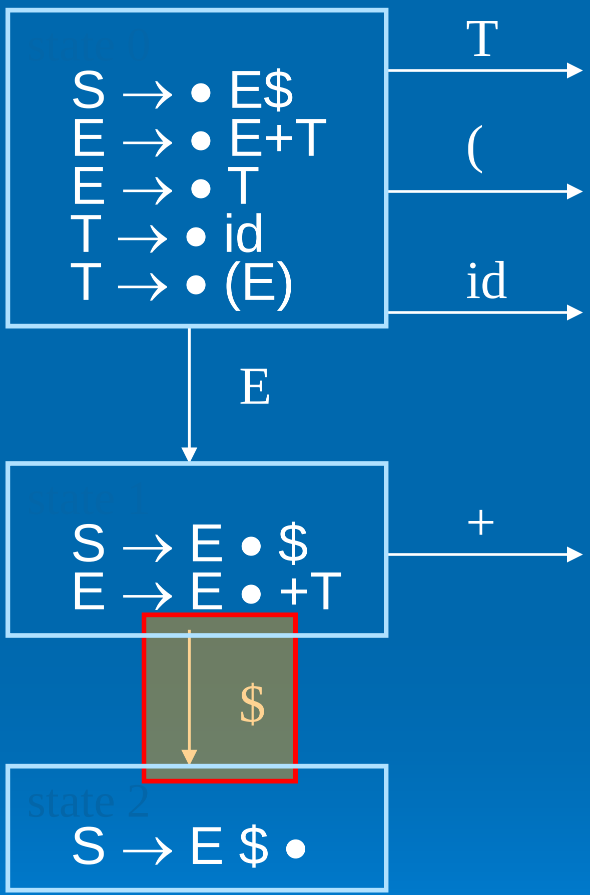
closure0({ S → • E$ }) =
{ S → • E$,
E → • E+T,
E → • T,
T → • id,
T → • (E) }

LR(0) Tracing Example(2)



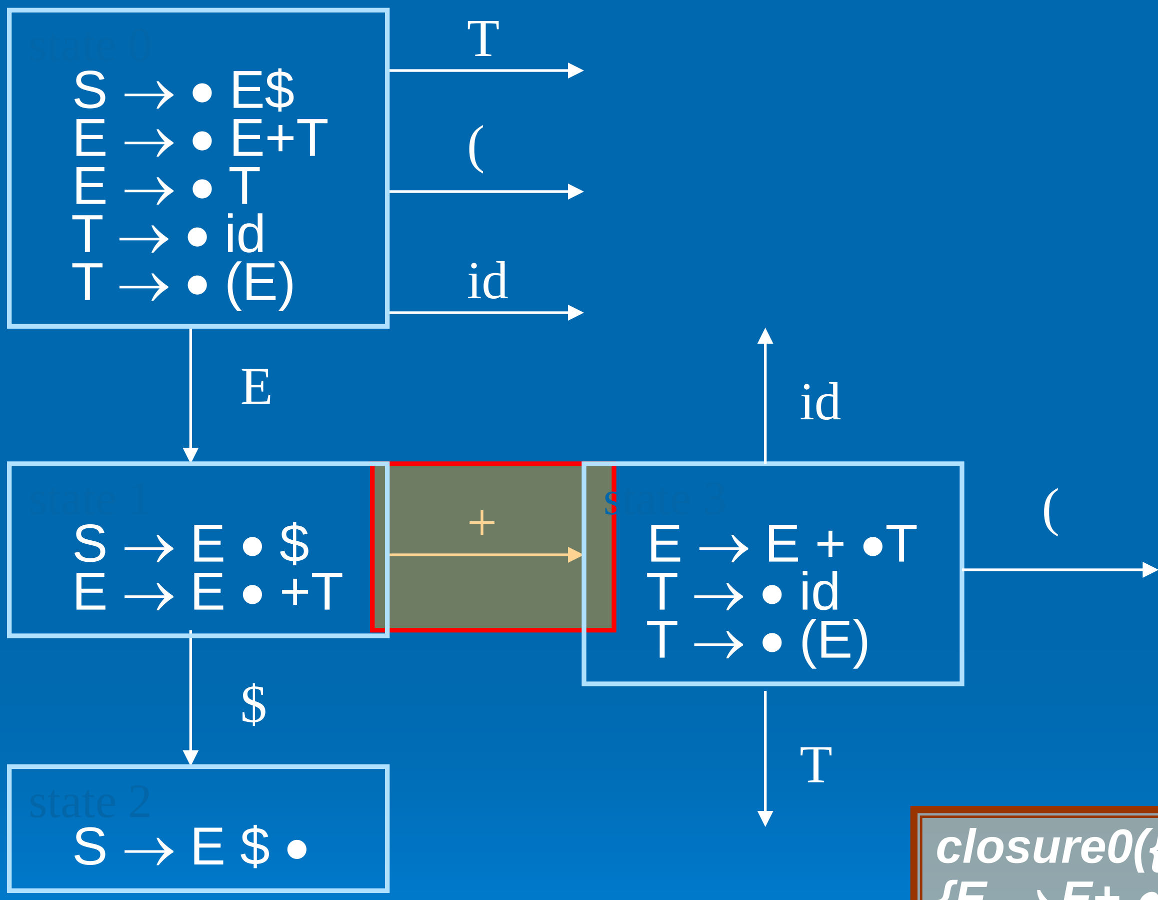
closure0({ S → E • $, E → E • +T }) =itself

LR(0) Tracing Example(3)



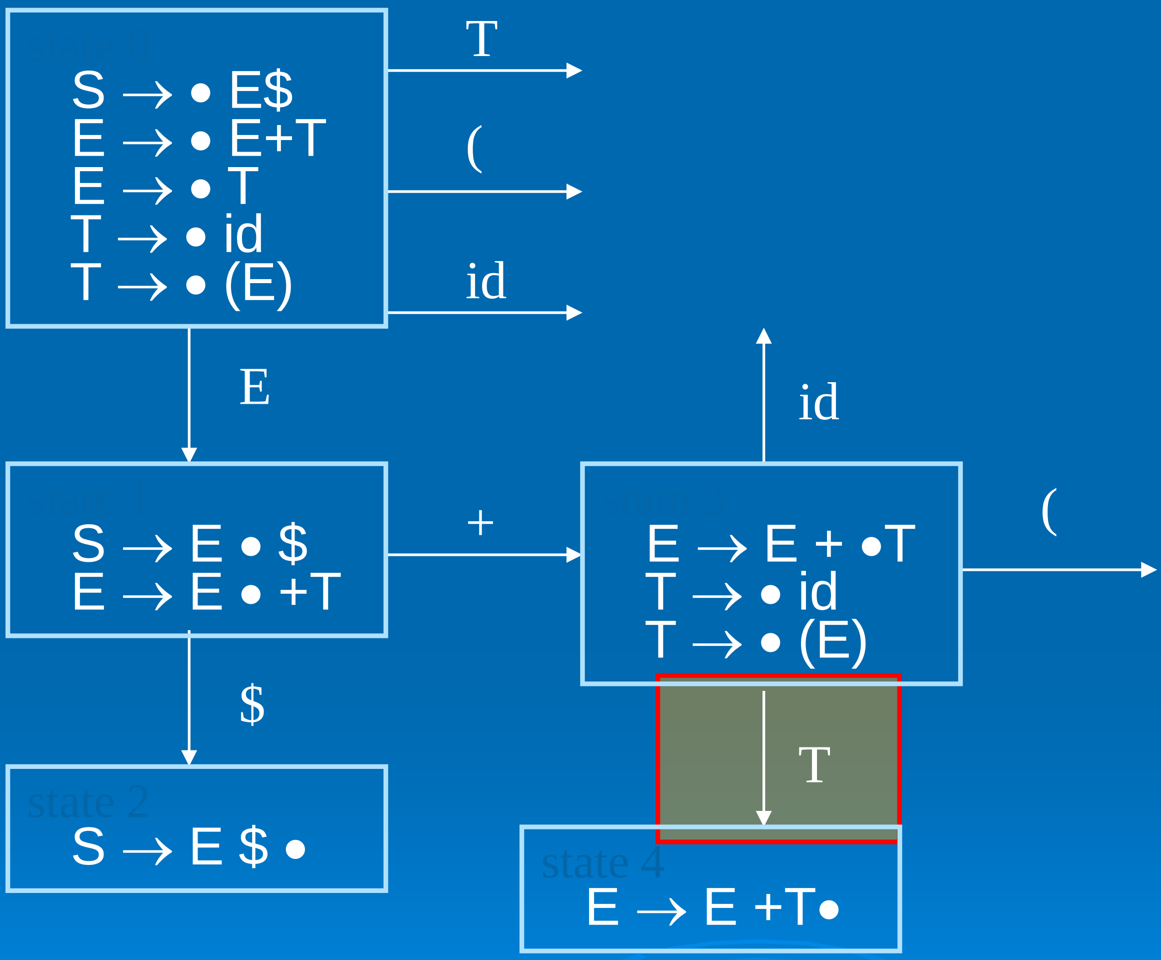
closure0({ S → E $ •}) =itself

LR(0) Tracing Example(4)



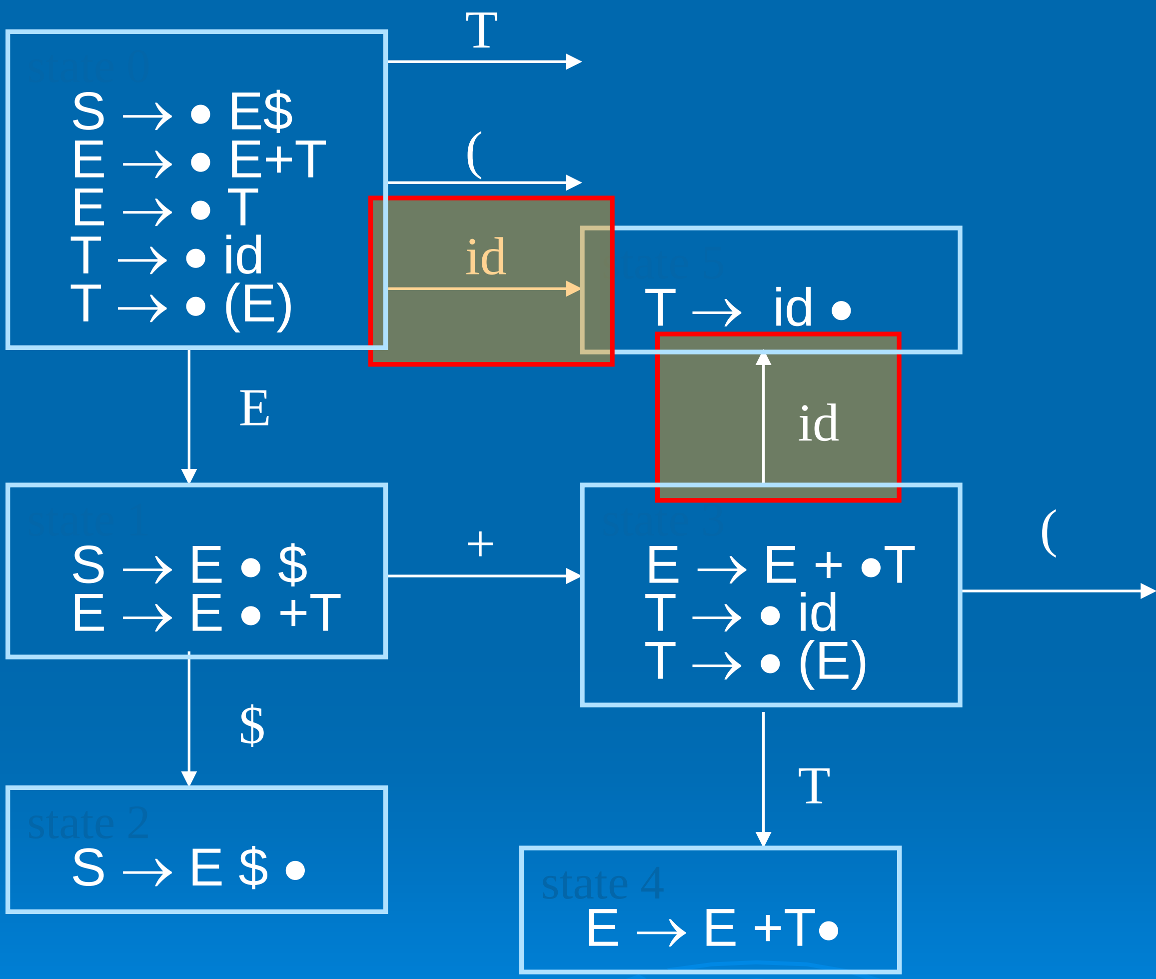
closure0({E → E+ • T}) =
{E → E+ • T,
T → • id,
T → • (E) }

LR(0) Tracing Example(5)



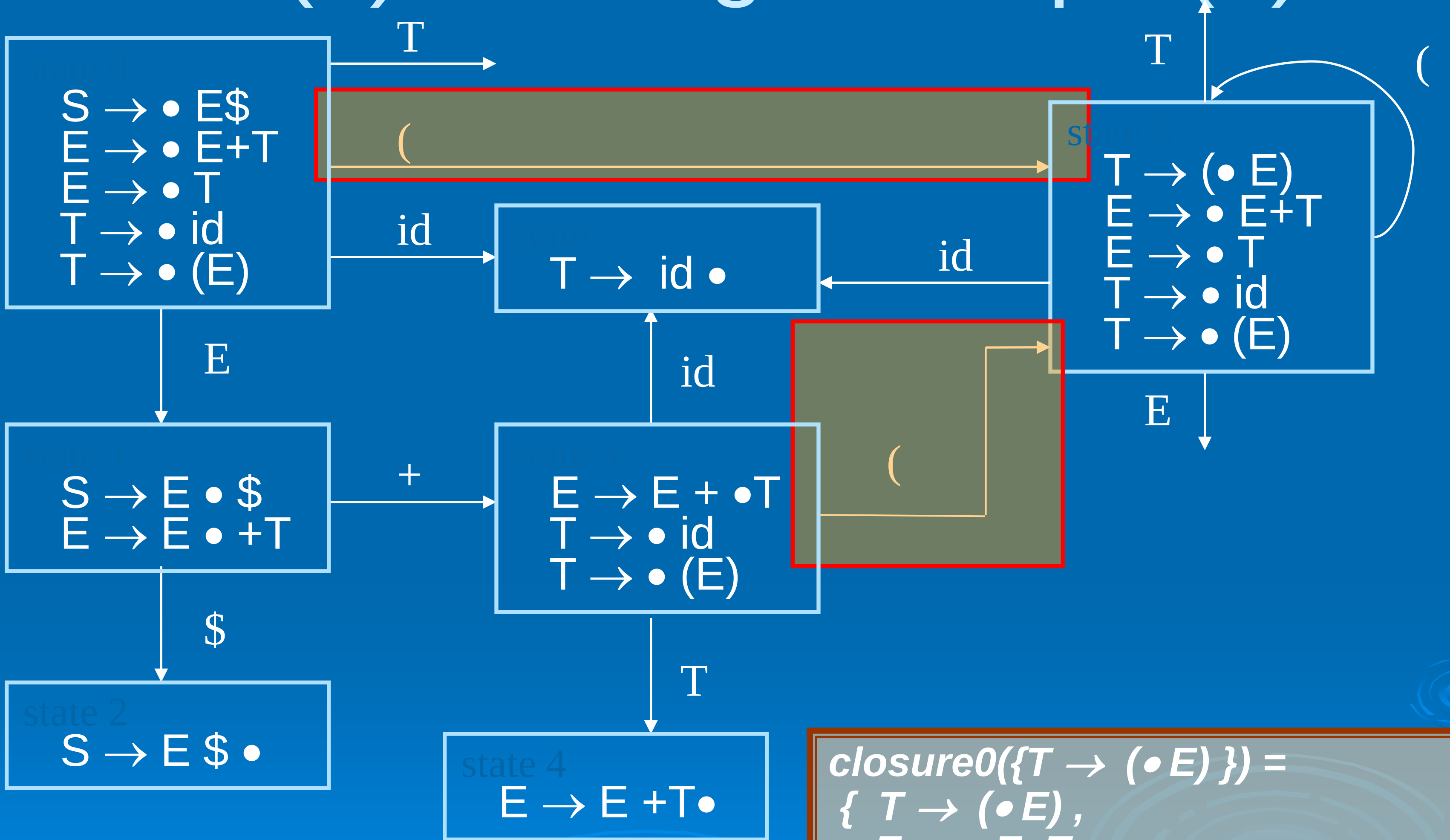
closure0({E → E+ T •}) =itself

LR(0) Tracing Example(6)



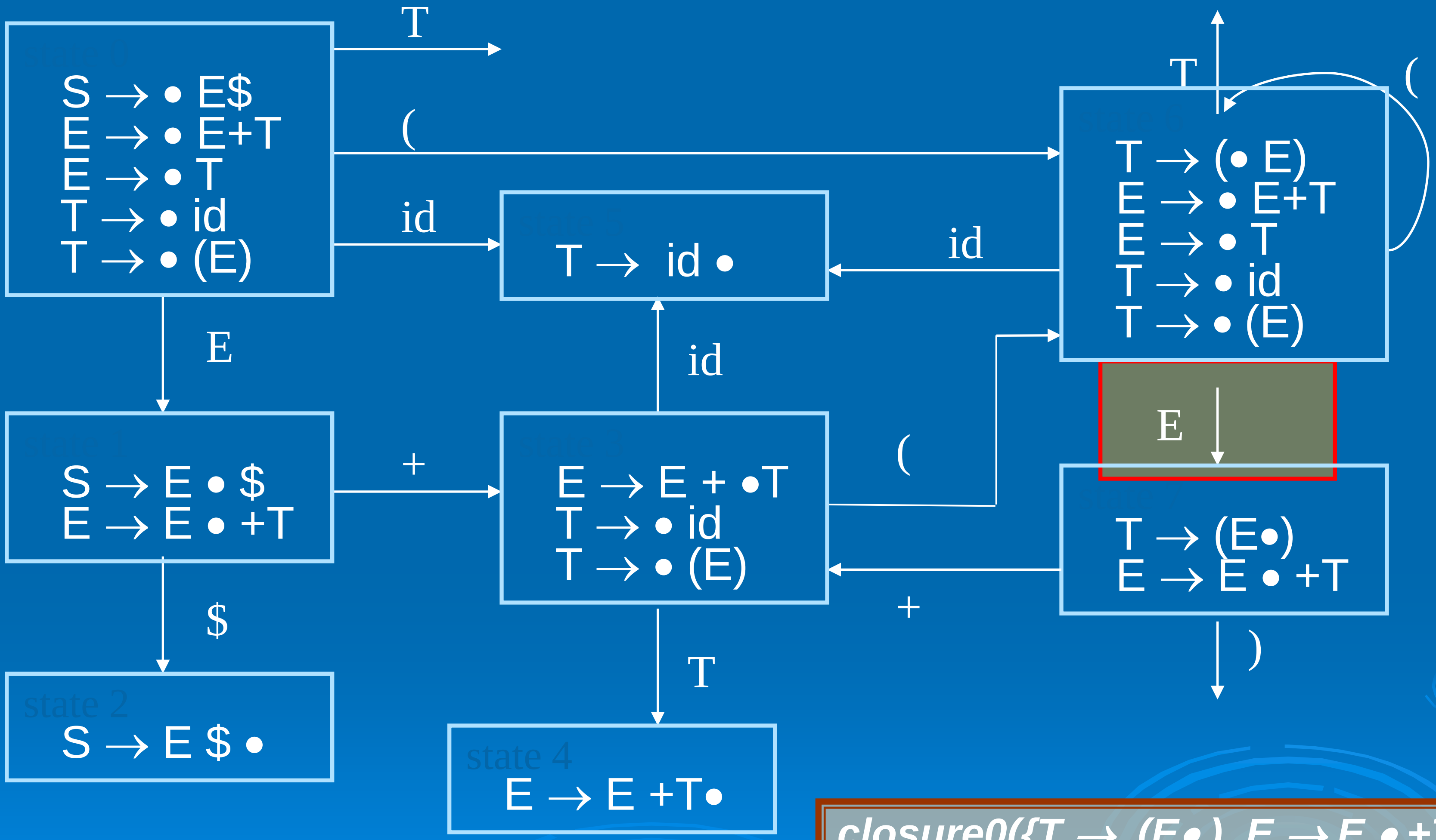
closure0({T → id • }) =itself

LR(0) Tracing Example(7)



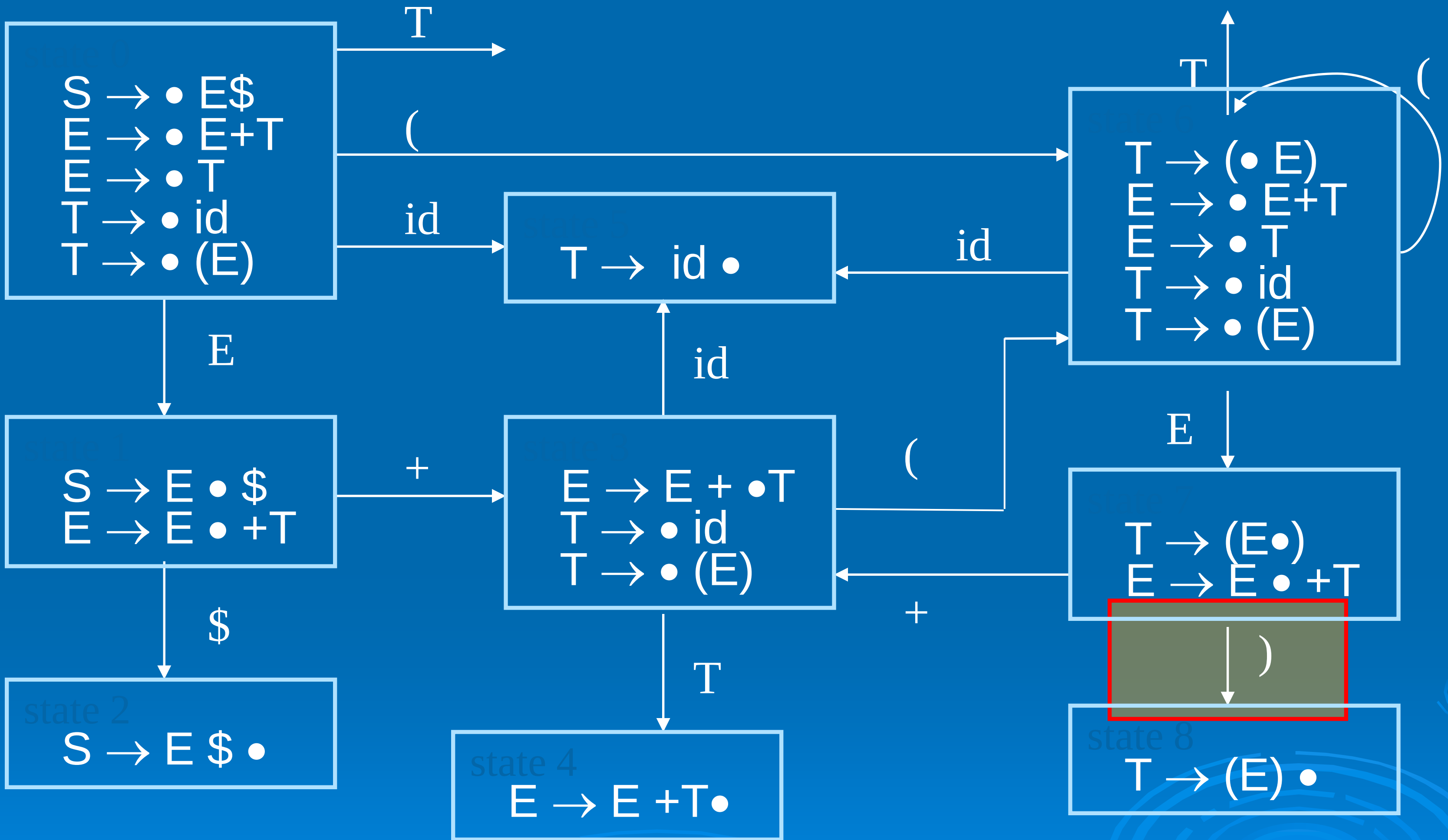
closure0({T → (• E) }) =
{ T → (• E) ,
E → • E+T,
E → • T,
T → • id,
T → • (E) }

LR(0) Tracing Example(8)



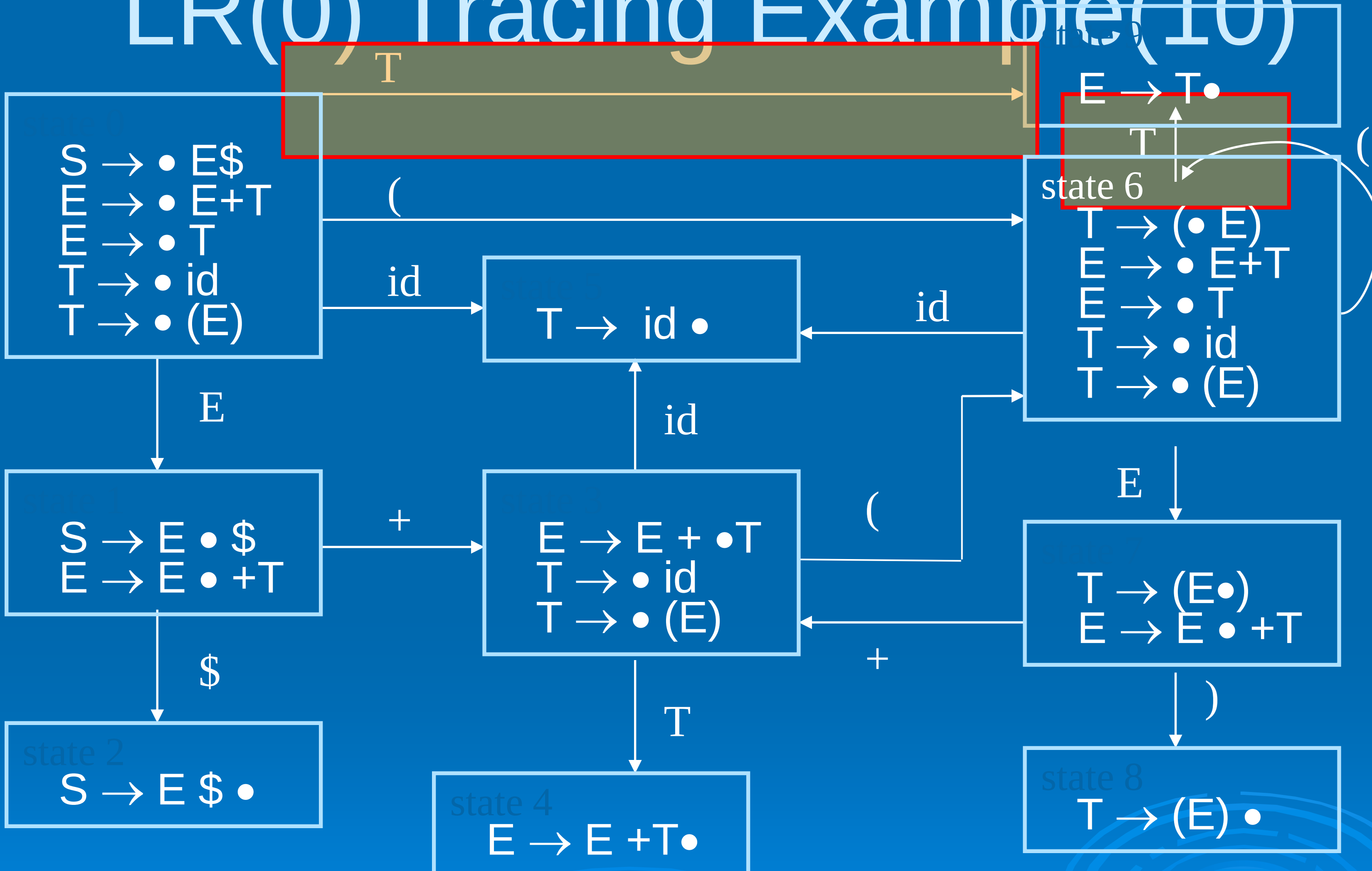
closure0({T → (E•) ,E → E •+T }) =itself

LR(0) Tracing Example(9)



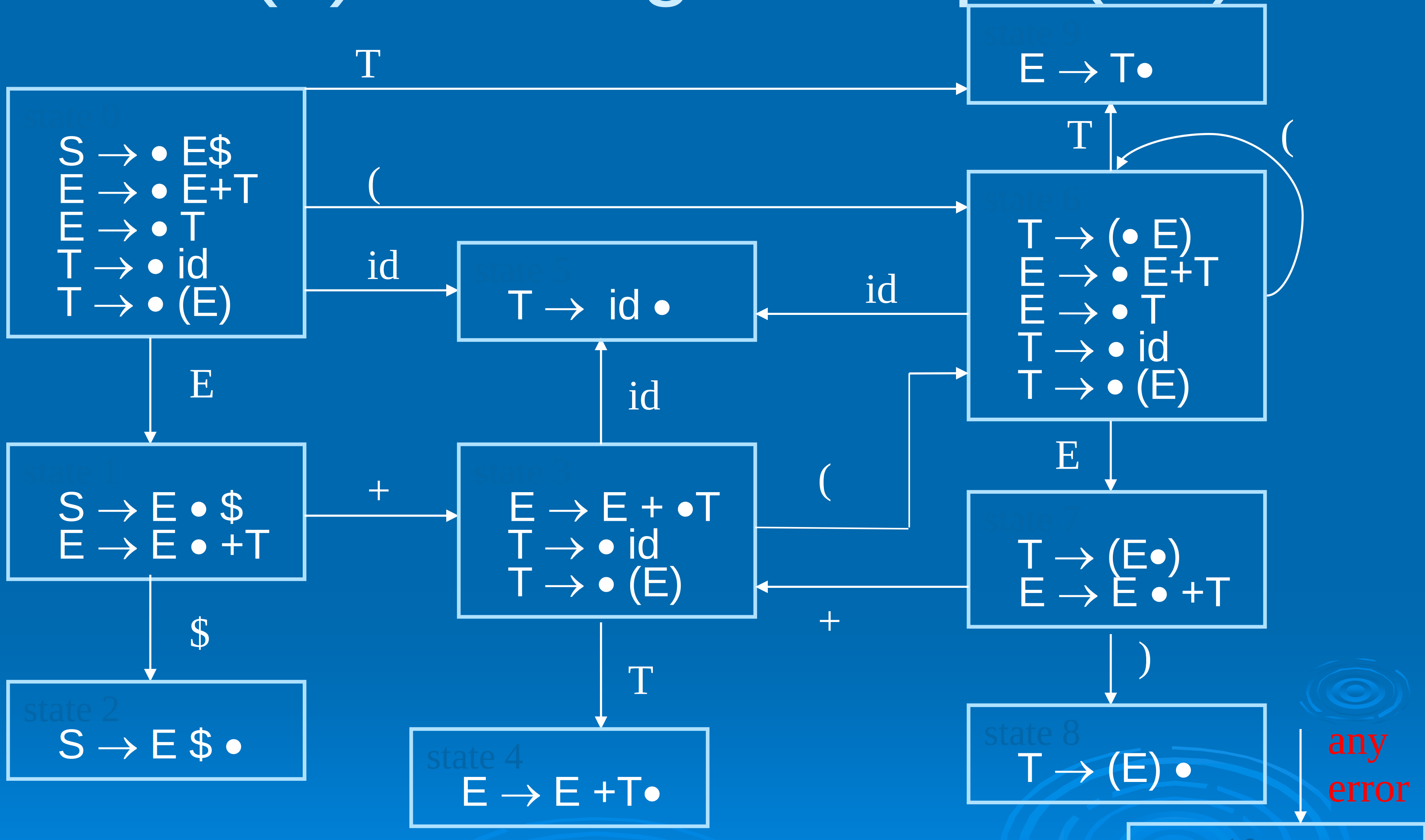
closure0({T → (E) •}) =itself

LR(0) Tracing Example(10)



closure0({E → T •}) =itself

LR(0) Tracing Example(11)

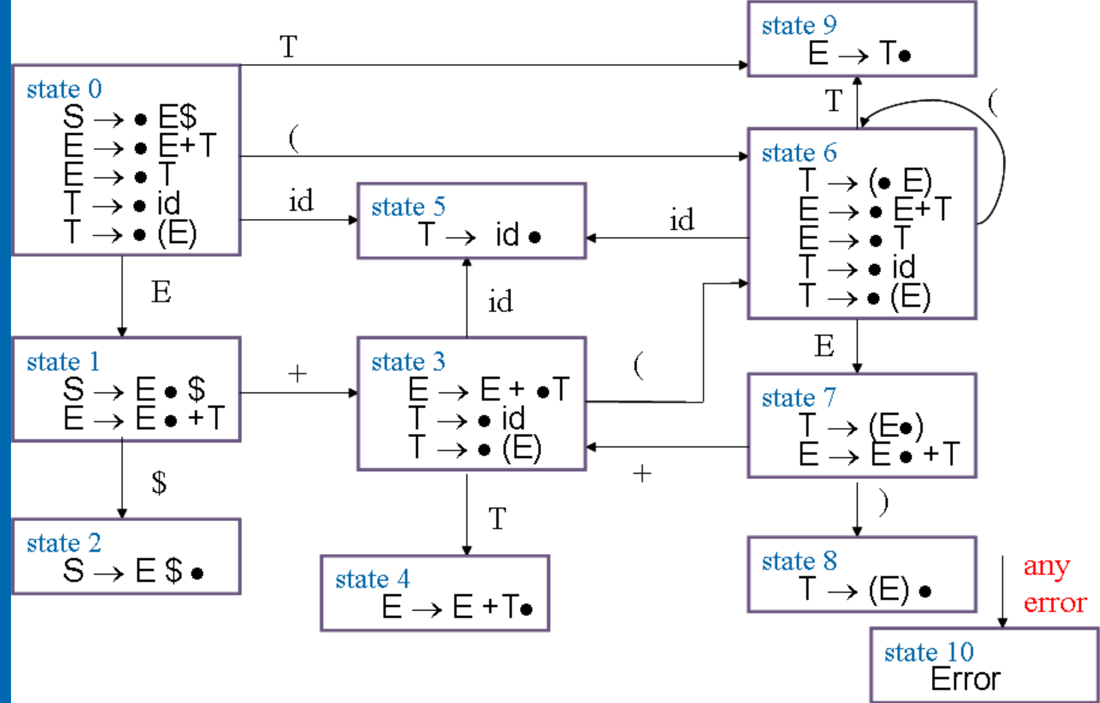


action table

Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything			A		R2	R4			R5	R3	

LR(0) Tracing Example(12)

State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								



goto table

Program Example (1)

State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

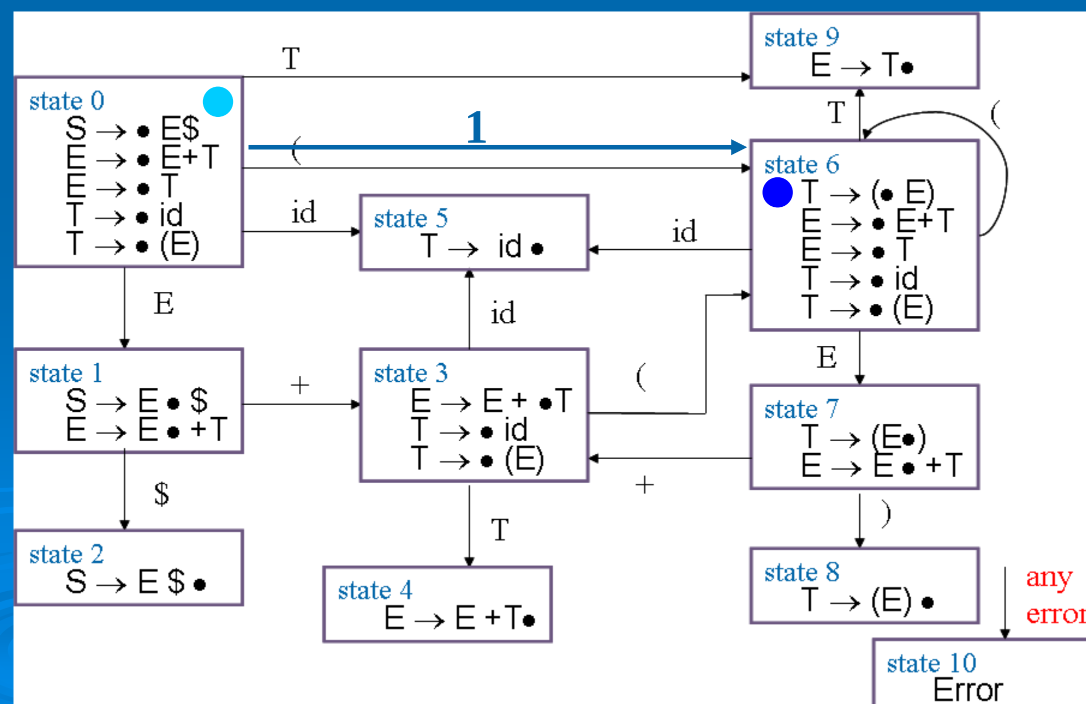
Initial :(id)$

step1:0 (id)$ shift

Tree:

(

Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	



State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

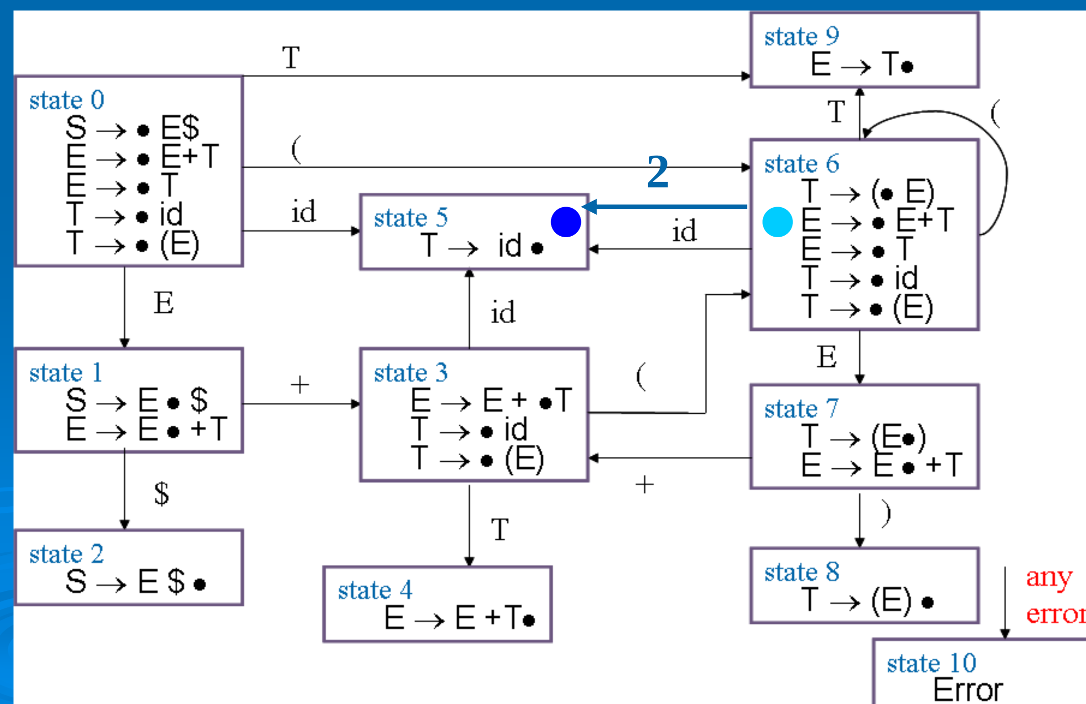
Initial :(id)$

step2:06)$

Tree:

(id

Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything			A		R2	R4			R5	R3	



State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

Program Example (3)

Initial :(id)$

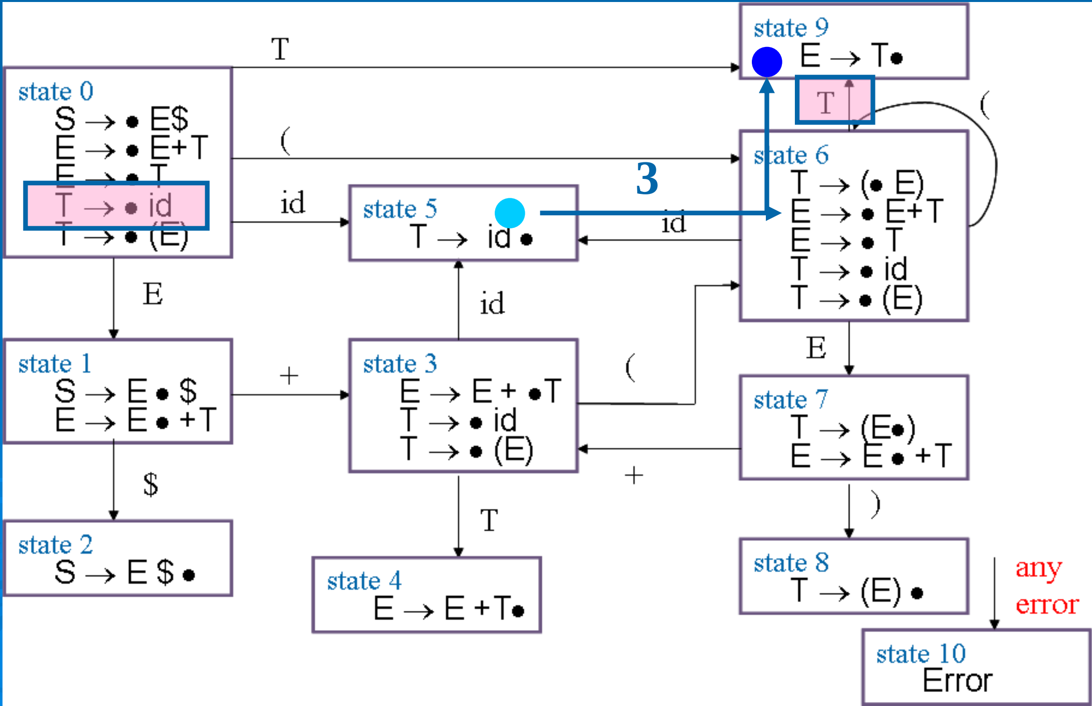
step3:065 **$** reduce 3

Tree:

(T

|

id



Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	

State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

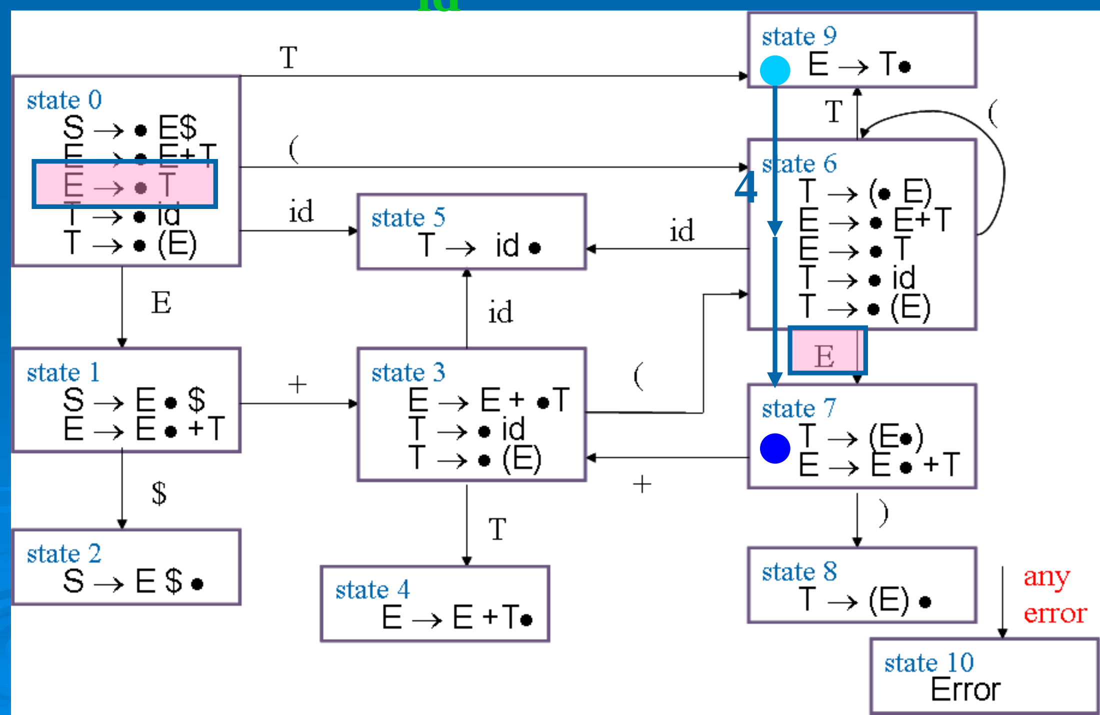
Program Example (4)

Initial :(id)$

step4:069 $ reduce 3

Tree:

(E
|
T
|
id



Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	

Program Example (5)

State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

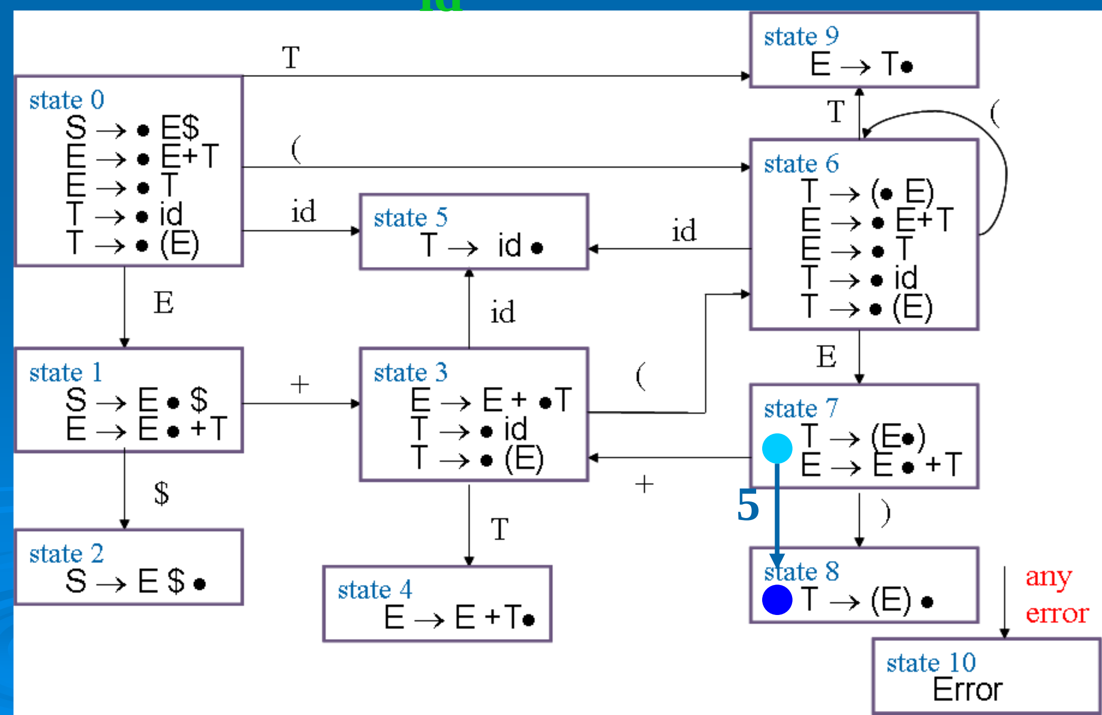
Initial :(id)$

step5:067 **$**

Tree:

(**E**)
|
T
|
id

Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	

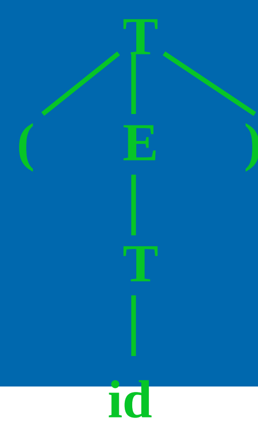


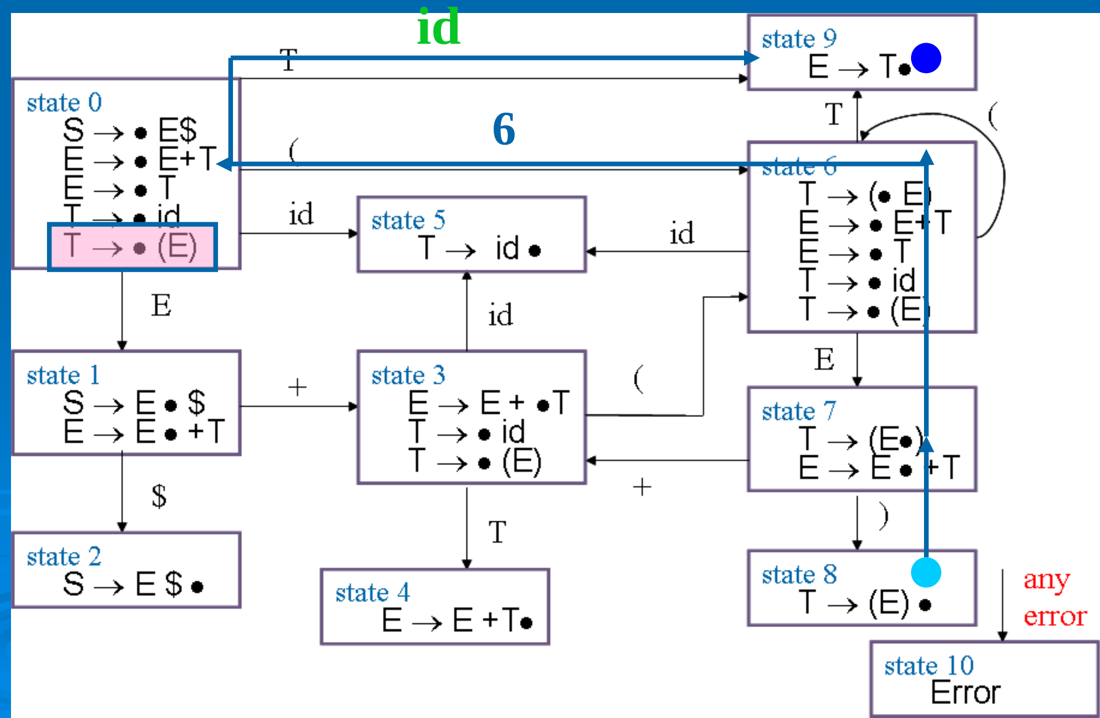
State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

Initial :(id)$

step6:0678

Tree:





Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	

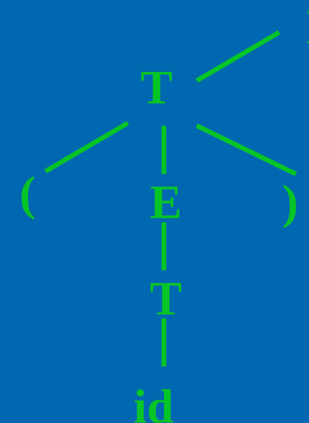
State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

Program Example (7)

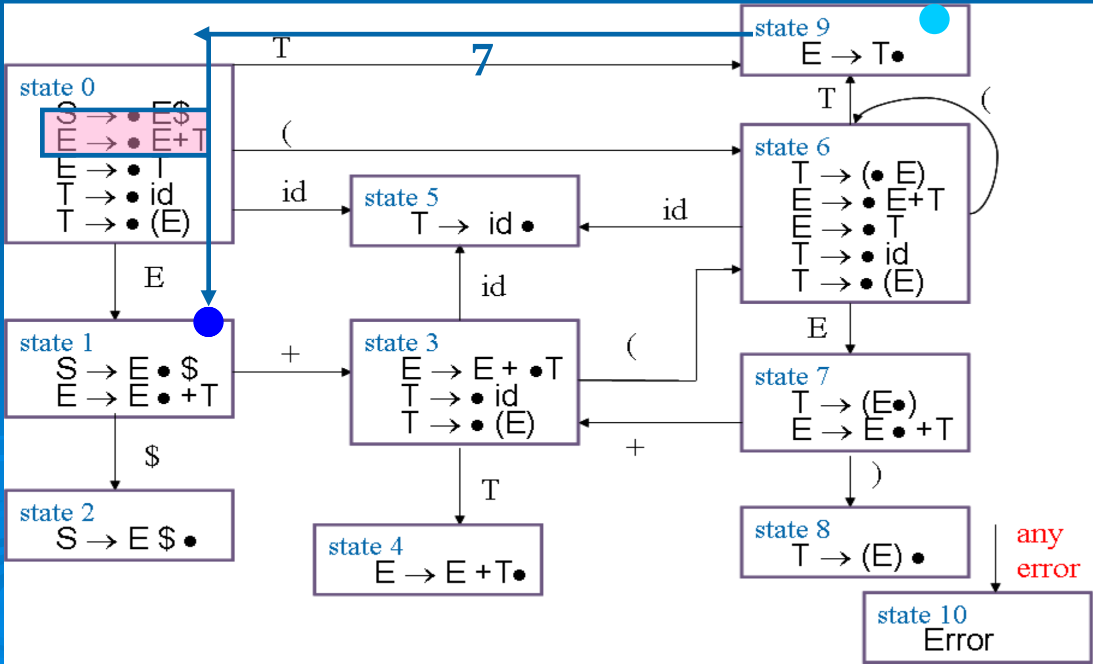
Initial :(id)$

step7:09

Tree:



Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	



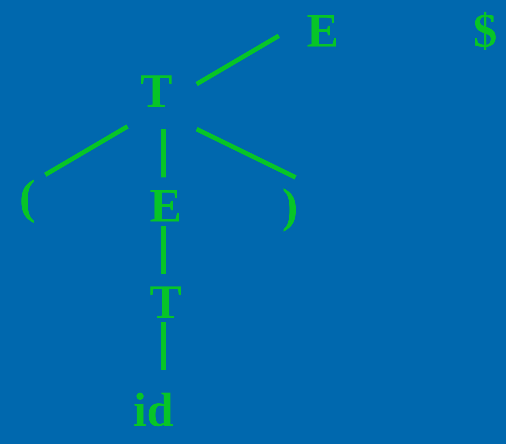
State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

Program Example (8)

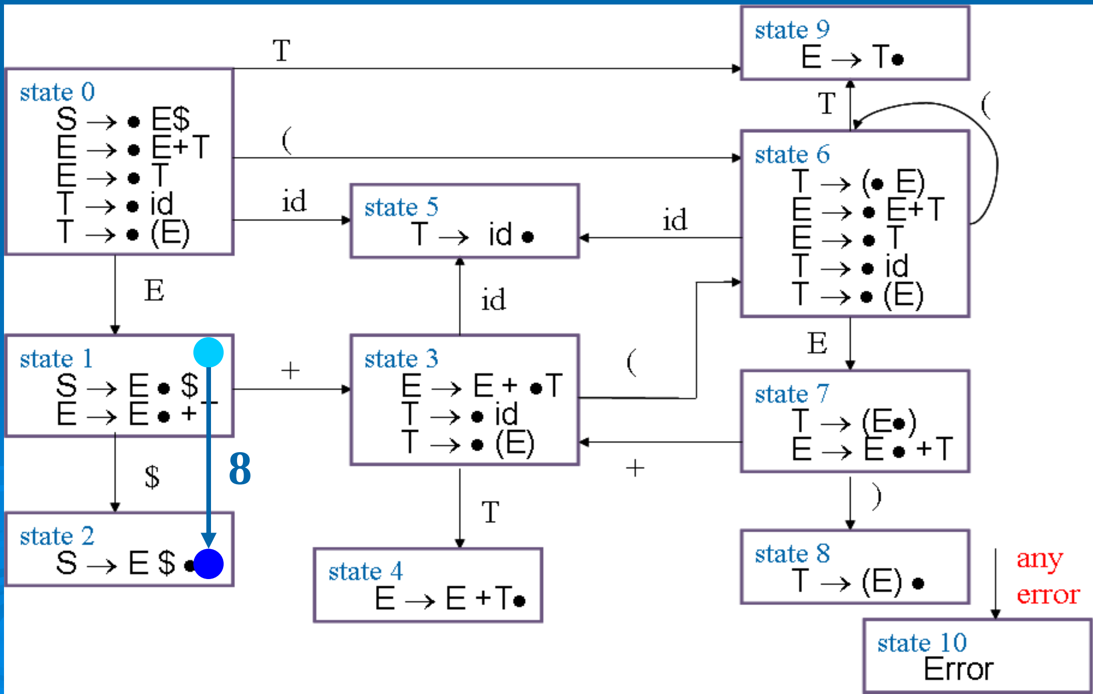
Initial :(id)$

step8:01 $ shift $

Tree:



Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	



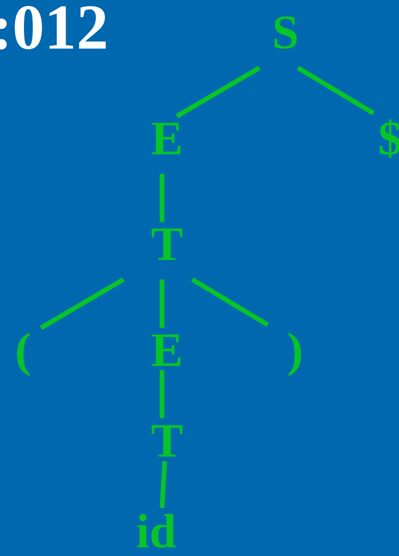
State	Symbol							
	S	E	T	+	id	(	)	$
0		1	9		5	6		
1				3				2
2								
3			4		5	6		
4								
5								
6		7	9		5	6		
7				3			8	
8								
9								
10								

Initial :(id)$

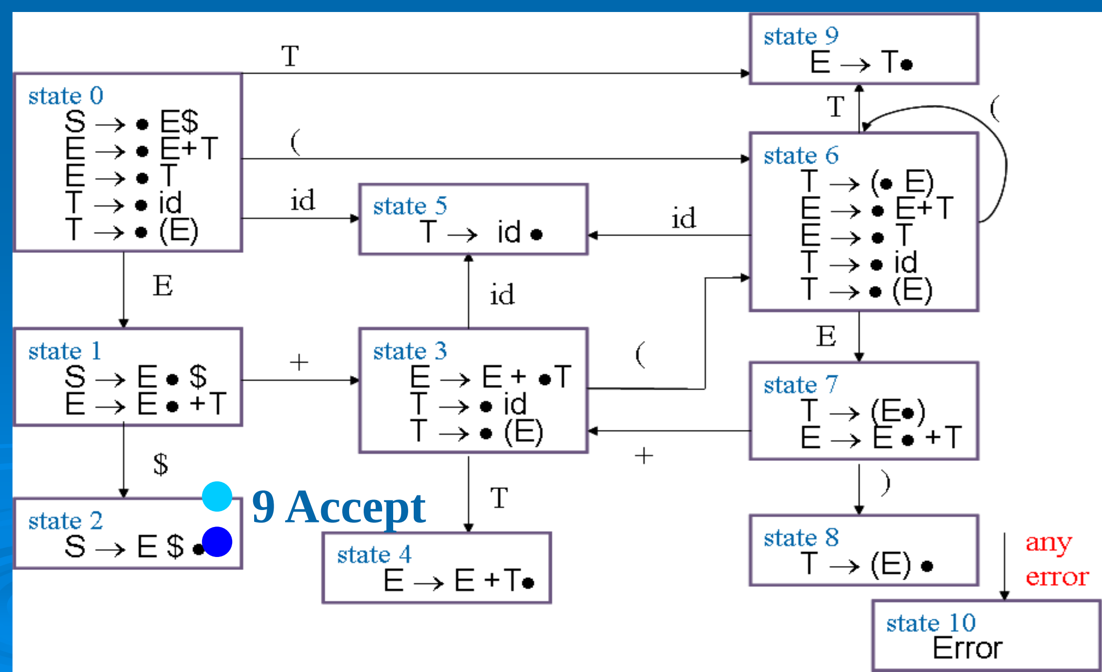
step9:012

Accept

Tree:



Symbol	State										
	0	1	2	3	4	5	6	7	8	9	10
anything	S	S	A	S	R2	R4	S	S	R5	R3	



Thank you