Compilation

0368-3133

Tutorial 3:

Bottom-Up Parsing

Bottom-Up Parsing

- Builds the derivation tree from the bottom to the top
- Produces the rightmost derivation
- If not sure what rule to use delay the decision and continue reading
- Can handle left recursion

Bottom-Up Parsing – Simple Example

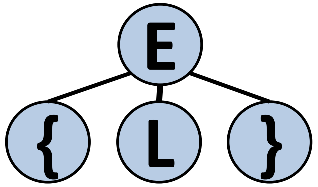
E → ID = INT

E → { L }

L → E

L → L ; E

- Input: { x = 8 ; y = 7 }
- What rule should we choose when we read x?



Bottom-Up Parsing – Simple Example

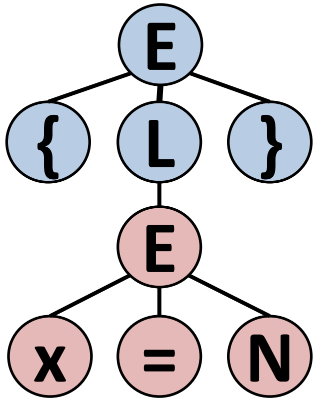
E → ID = INT

E → { L }

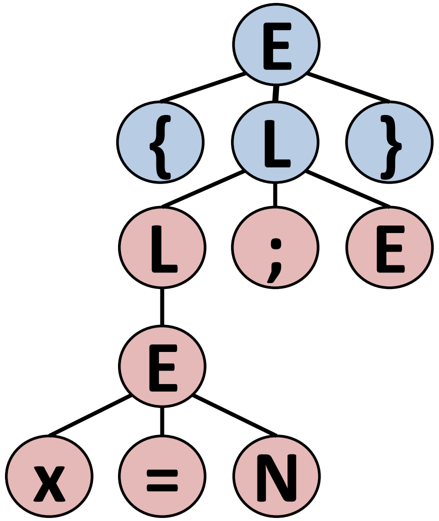
L → E

L → L ; E

- Input: { x = 8 ; y = 7 }
- What rule should we choose when we read x?
- We need to delay the decision until we read ;



or



Bottom-Up Parsing – Simple Example

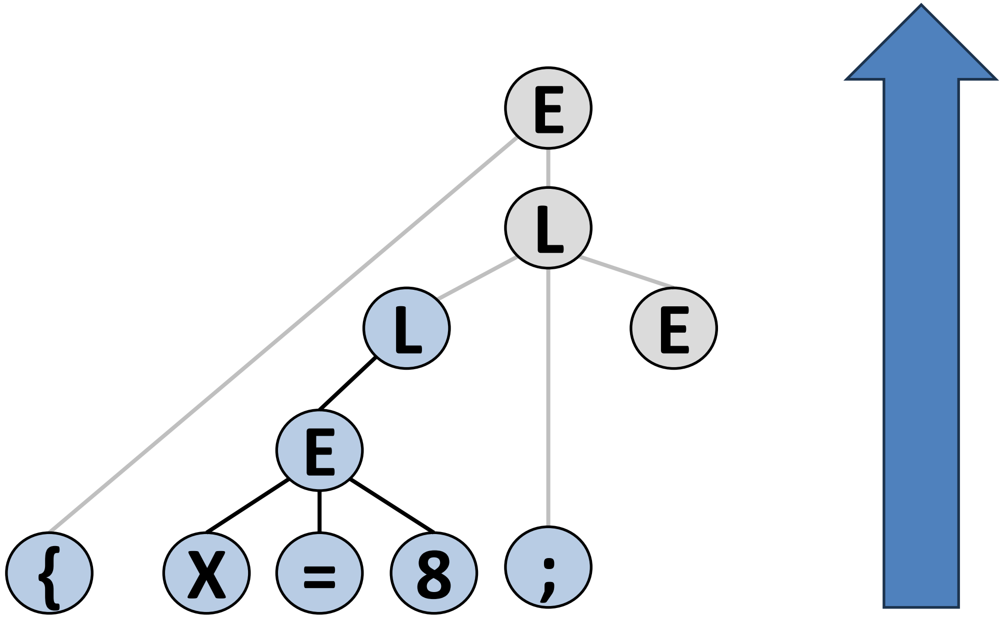
E → ID = INT

E → { L }

L → E

L → L ; E

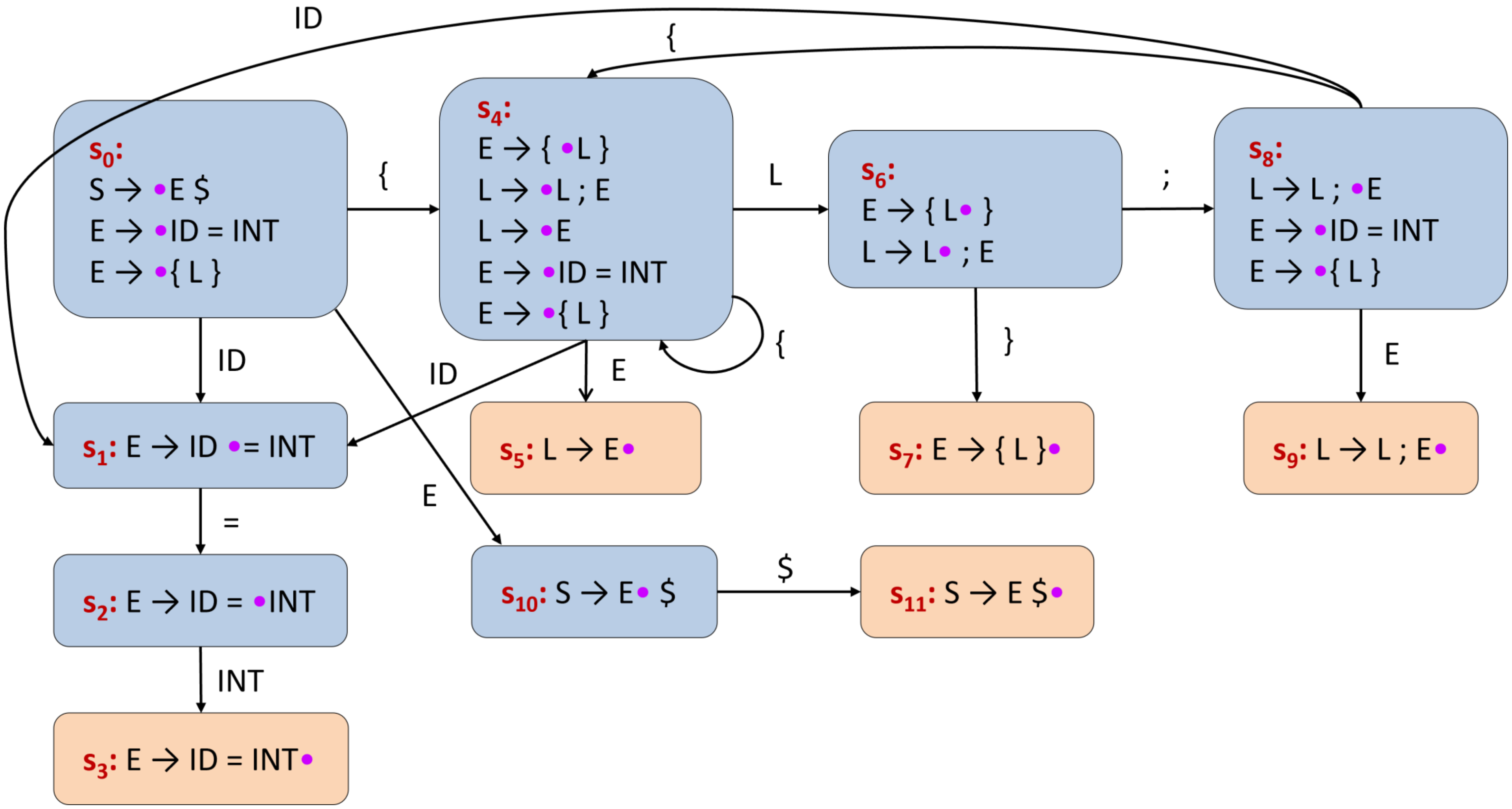
- Input: { x = 8 ; y = 7 }
- What rule should we choose when we read x?
- We need to delay the decision until we read ;



Today: LR(0) Parser

- Scans the input from left to right
- Produces the rightmost derivation
- No lookahead

Today: LR(0) Parser



One Initial Rule

- Ensure there is only one non recursive initial rule
- If not add the rule S’ → S
- Add $ to the end of the initial rule

S → E $

E → ID = INT

E → { L }

L → E

L → L ; E

LR(0) Item

- **LR(0) item** is of the form:

  $A \rightarrow \alpha \bullet \beta$

  – The dot/marker gives us the current location (a local view)

- An LR(0) item with the dot at the end is called *reduce* item:

  $A \rightarrow \alpha \beta \bullet$

- Otherwise, it is a *shift* item:

  $A \rightarrow \bullet \alpha \beta$

  $A \rightarrow \alpha \bullet \beta$

LR(0) Item Closure Set

Given an LR(0) item I we define its *closure set* S inductively as follows:

- Base: I ∋ S
- Inductive step: If A → α •Bβ ∋ S then

  for each rule B → γ also B → •γ ∋ S

LR(0) Item Closure Set

$S \rightarrow E\ \$$

$E \rightarrow ID = INT$

$E \rightarrow \{\ L\ \}$

$L \rightarrow E$

$L \rightarrow L\ ;\ E$

The closure set of $S \rightarrow \bullet E\ \$$:

- $S \rightarrow \bullet E\ \$$
- $E \rightarrow \bullet ID = INT$
- $E \rightarrow \bullet \{\ L\ \}$

LR(0) Parser

S → E $

E → ID = INT

E → { L }

L → E

L → L ; E

- Let’s construct the *LR(0) automaton* for this CFG

LR(0) Automaton

- We start with the initial LR(0) item (that comes from the initial rule):

  S → •E $

- The initial state is the closure of this item, which contains:

  S → •E $

  E → •ID = INT

  E → •{ L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E

s_0:

$S \rightarrow \bullet E\ \$$

$E \rightarrow \bullet ID = INT$

$E \rightarrow \bullet \{\ L\ \}$

LR(0) Automaton

- From s_0, if we recognize ID, then the next state contains:

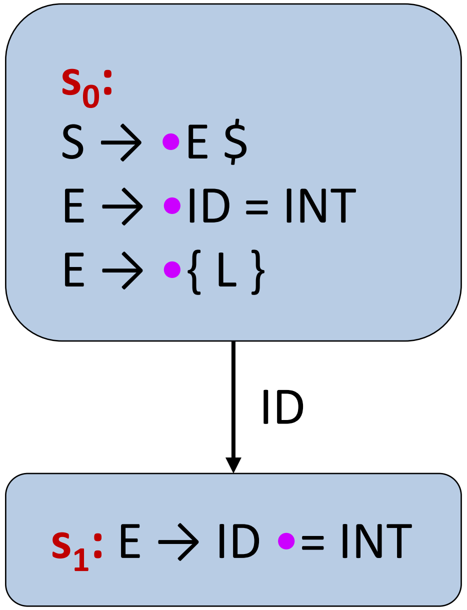
  E → ID • = INT

- The next state is the closure of this item:

  E → ID • = INT

s_0:
S → • E $
E → • ID = INT
E → • { L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
ID
s1: E → ID •= INT

LR(0) Automaton

- From s_1, if we recognize =, then the next state contains:

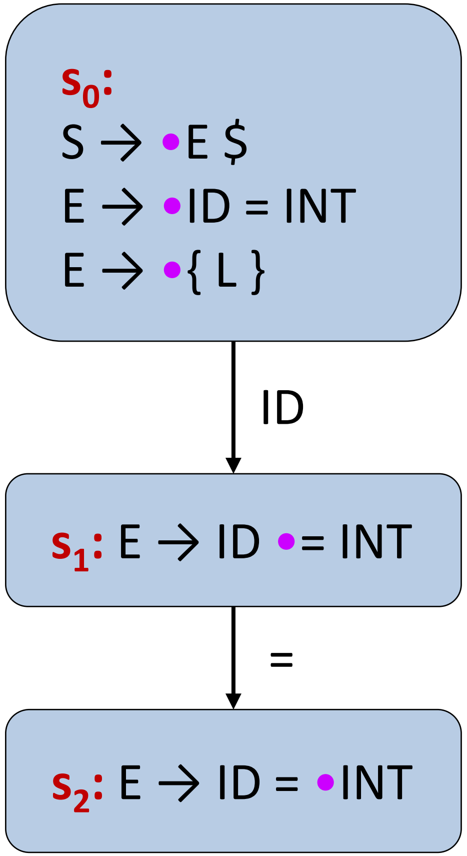
  E → ID = •INT

- The next state is the closure of this item:

  E → ID = •INT

s_1: E → ID •= INT

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
ID
s1: E → ID •= INT
=
s2: E → ID = •INT

LR(0) Automaton

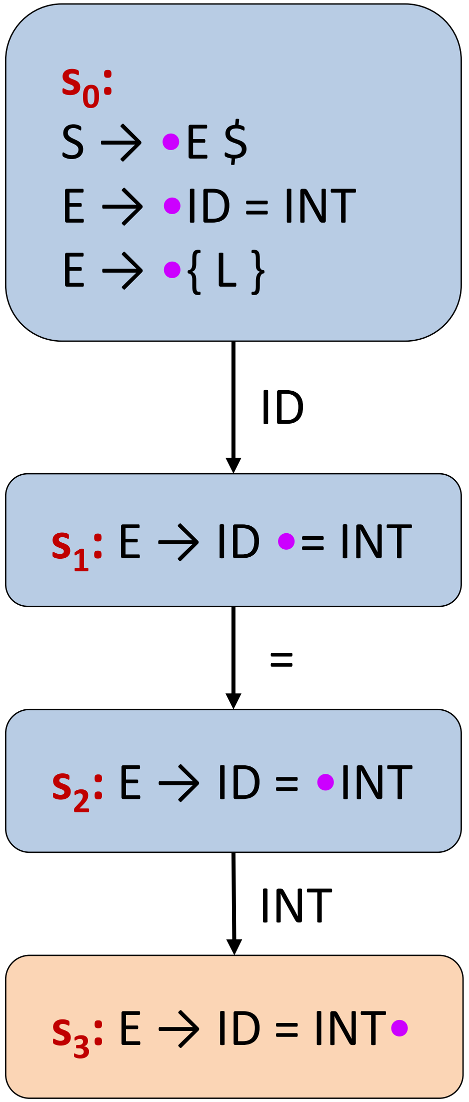
- From s_2, if we recognize INT, then the next state contains

  E → ID = INT •

- Which is a reduce state

s_2: E → ID = •INT

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
ID
s1: E → ID •= INT
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_0, if we recognize {, then the next state contains:

  E → { •L }

- The next state is the closure of this item:

  E → { •L }

  L → •L ; E

  L → •E

  E → •ID = INT

  E → •{ L }

s_0:

S → •E $

E → •ID = INT

E → •{ L }

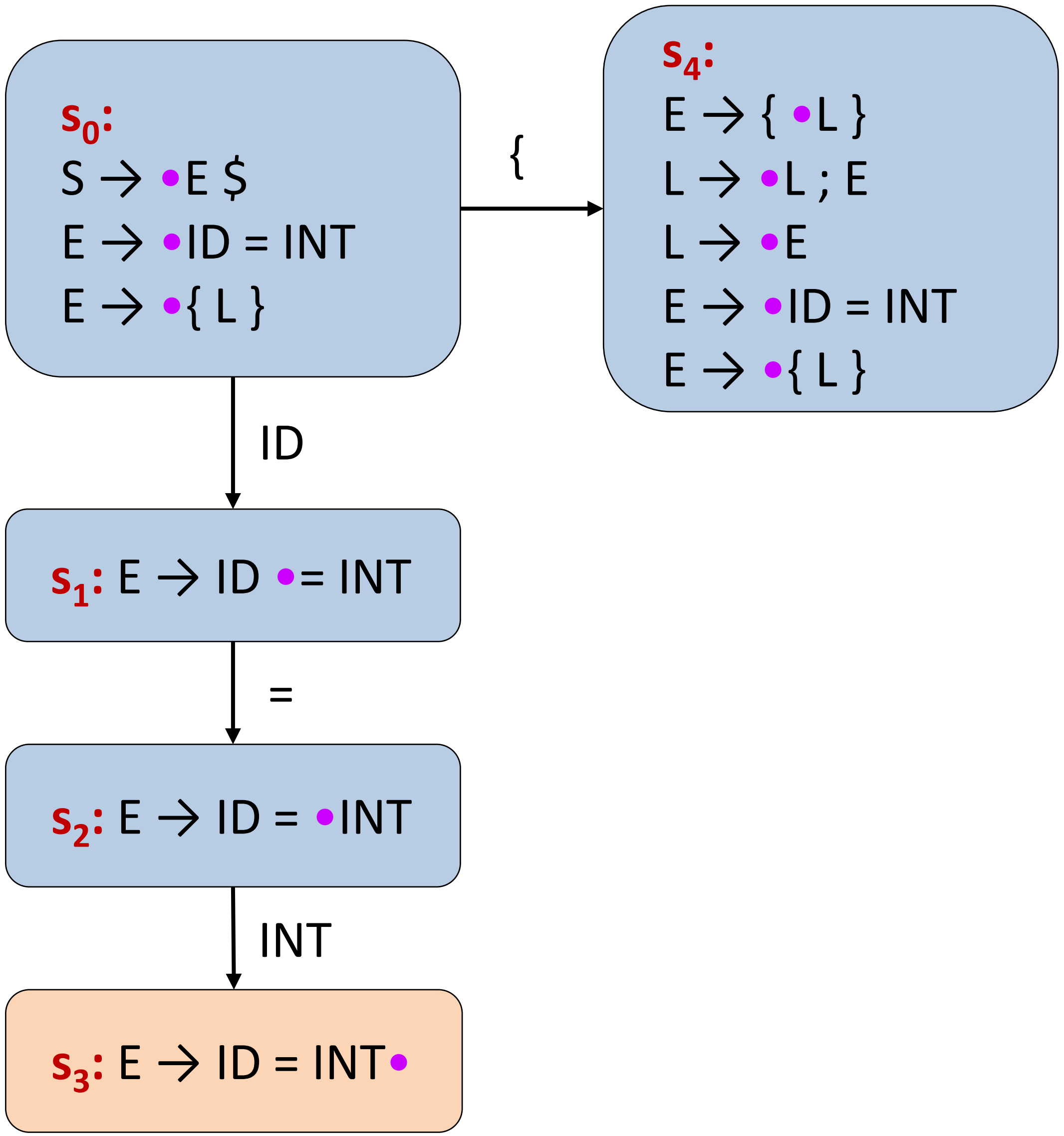
S → E $

E → ID = INT

E → { L }

L → E

L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
ID
s1: E → ID •= INT
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_4, if we recognize {, then the next state contains:

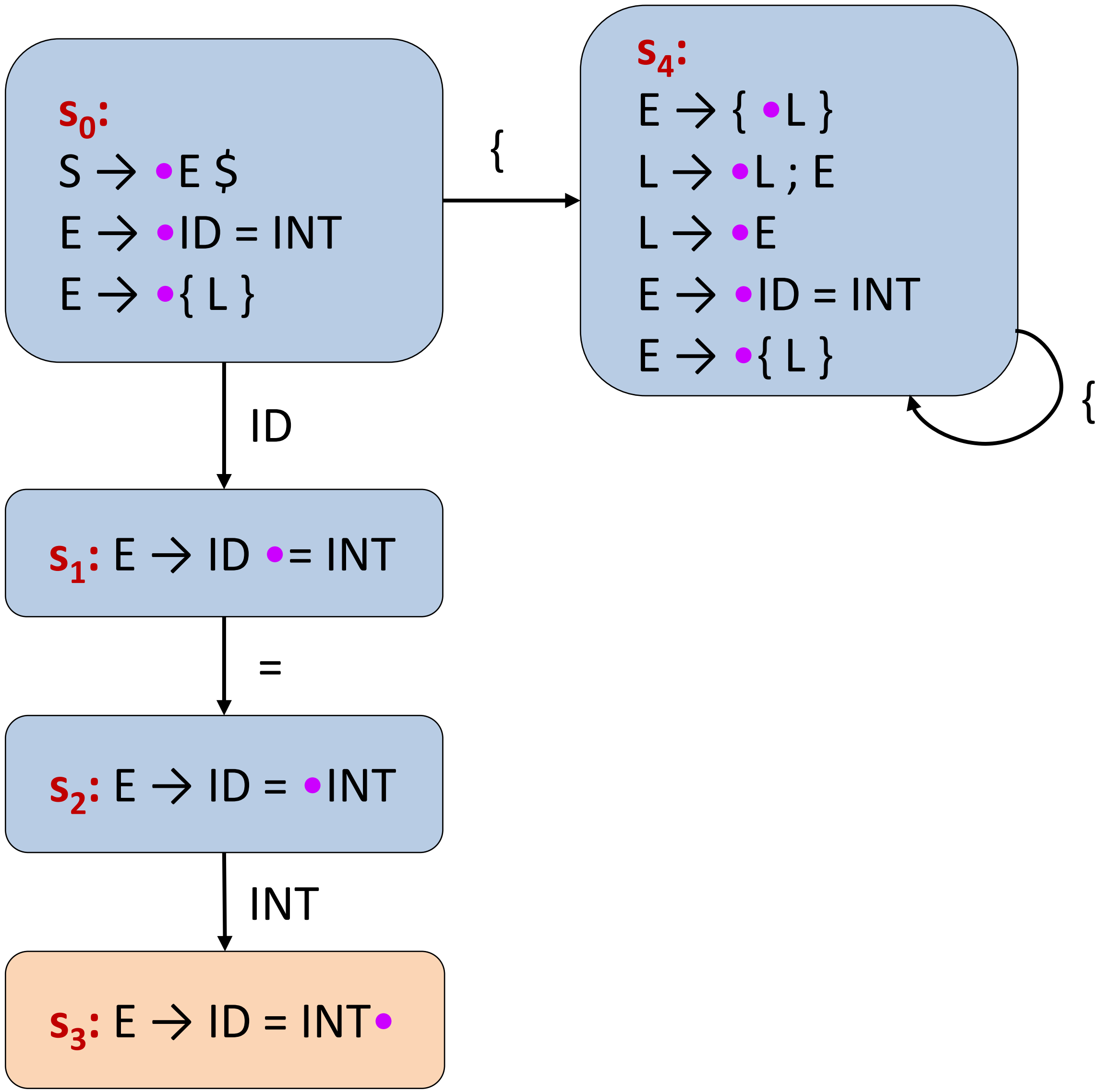
  E → { •L }

- The next state is the closure of this item:

$$\left.\begin{array}{l} E \rightarrow \{\ \bullet L\ \} \\ L \rightarrow \bullet L\ ;\ E \\ L \rightarrow \bullet E \\ E \rightarrow \bullet ID = INT \\ E \rightarrow \bullet \{\ L\ \} \end{array}\right\} s_4$$

s_4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
{
ID
s1: E → ID •= INT
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_4, if we recognize ID, then the next state contains:

  $E \rightarrow ID \bullet = INT$

- The next state is the closure of this item:

  $\left. E \rightarrow ID \bullet = INT \right\} s_1$

s_4:

$E \rightarrow \{ \bullet L \}$

$L \rightarrow \bullet L \, ; E$

$L \rightarrow \bullet E$

$E \rightarrow \bullet ID = INT$

$E \rightarrow \bullet \{ L \}$

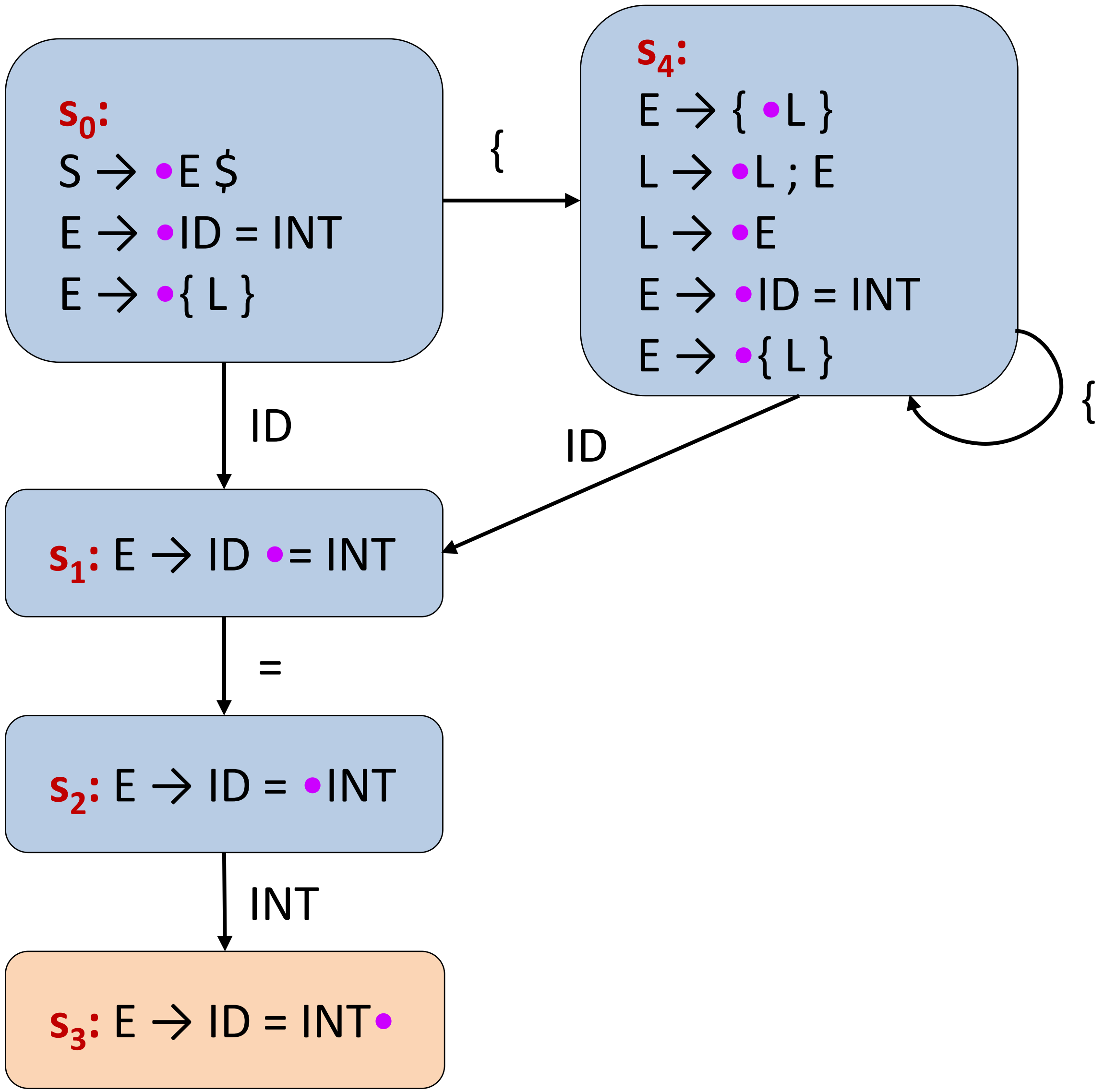
$S \rightarrow E \, \$$

$E \rightarrow ID = INT$

$E \rightarrow \{ L \}$

$L \rightarrow E$

$L \rightarrow L \, ; E$


s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
{
ID
ID
s1: E → ID •= INT
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_4, if we recognize E, then the next state contains:

  L → E•

- Which is a reduce state

s_4:

E → { •L }

L → •L ; E

L → •E

E → •ID = INT

E → •{ L }

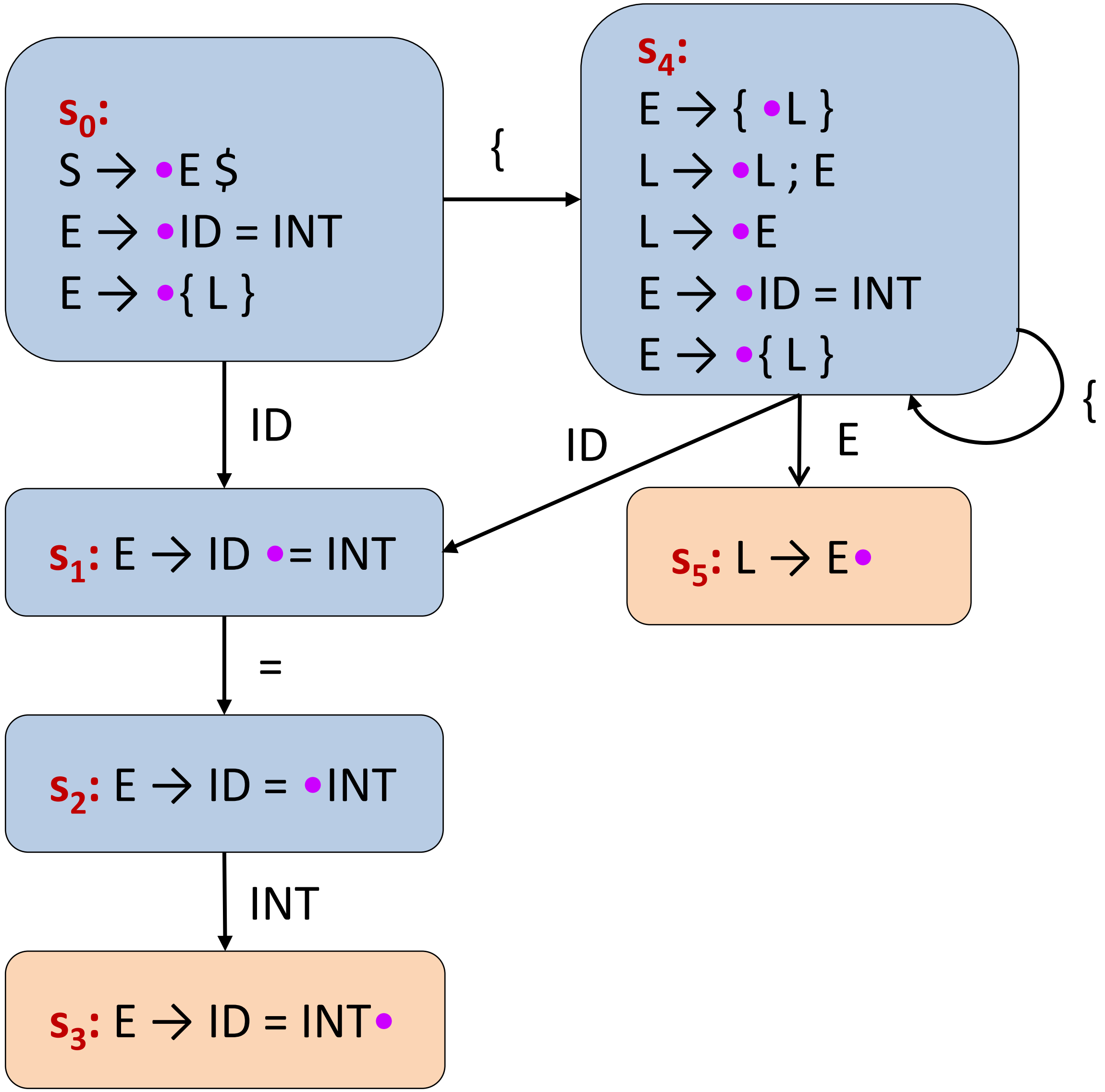
S → E $

E → ID = INT

E → { L }

L → E

L → L ; E



s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
{
ID
ID
E
s1: E → ID •= INT
s5: L → E•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_4, if we recognize L, then the next state contains:

  E → { L• }

  L → L• ; E

- The next state is the closure of these items:

  E → { L• }

  L → L• ; E

s_4:

E → { •L }

L → •L ; E

L → •E

E → •ID = INT

E → •{ L }

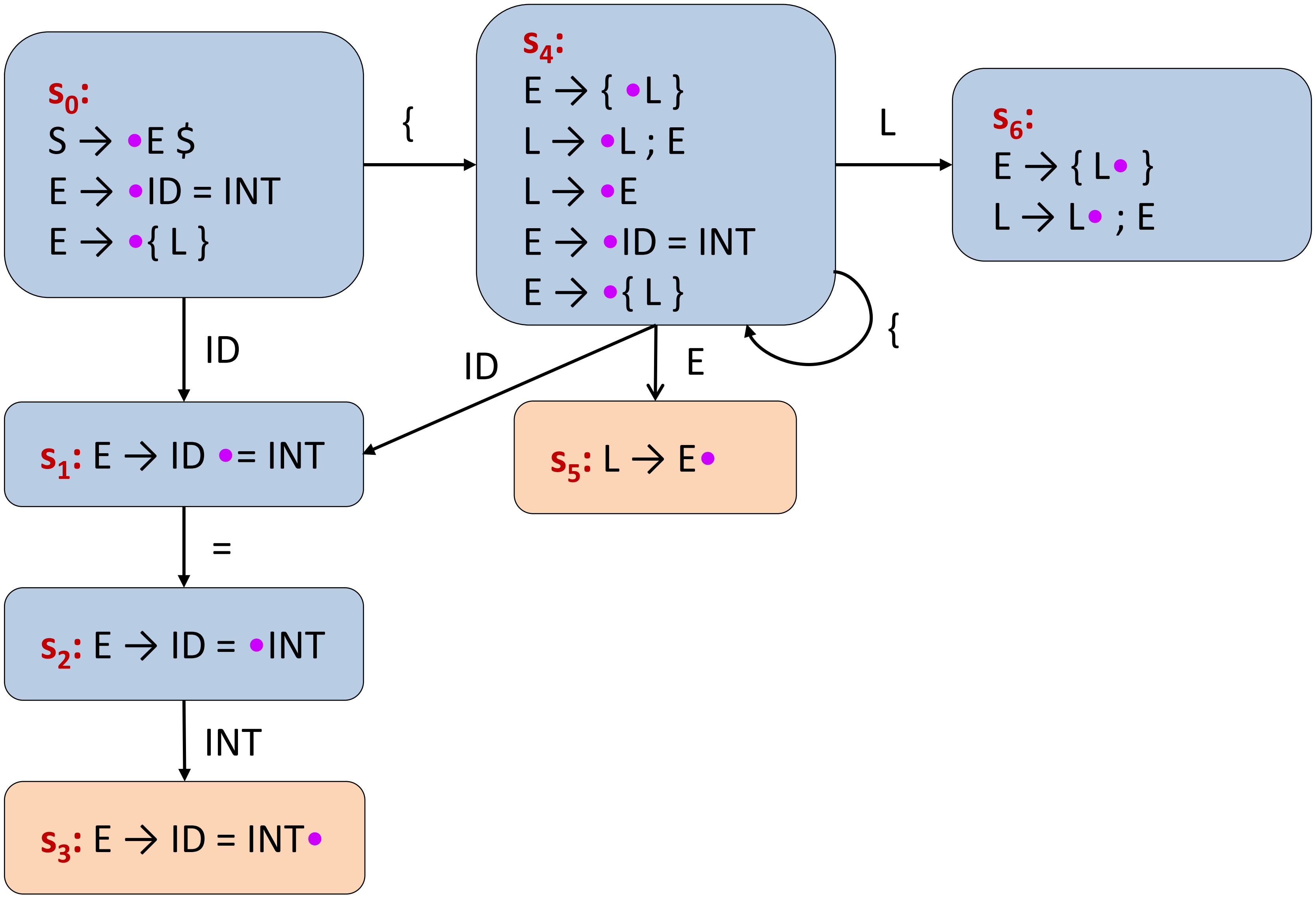
S → E $

E → ID = INT

E → { L }

L → E

L → L ; E



s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
{
ID
ID
E
s1: E → ID •= INT
s5: L → E•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

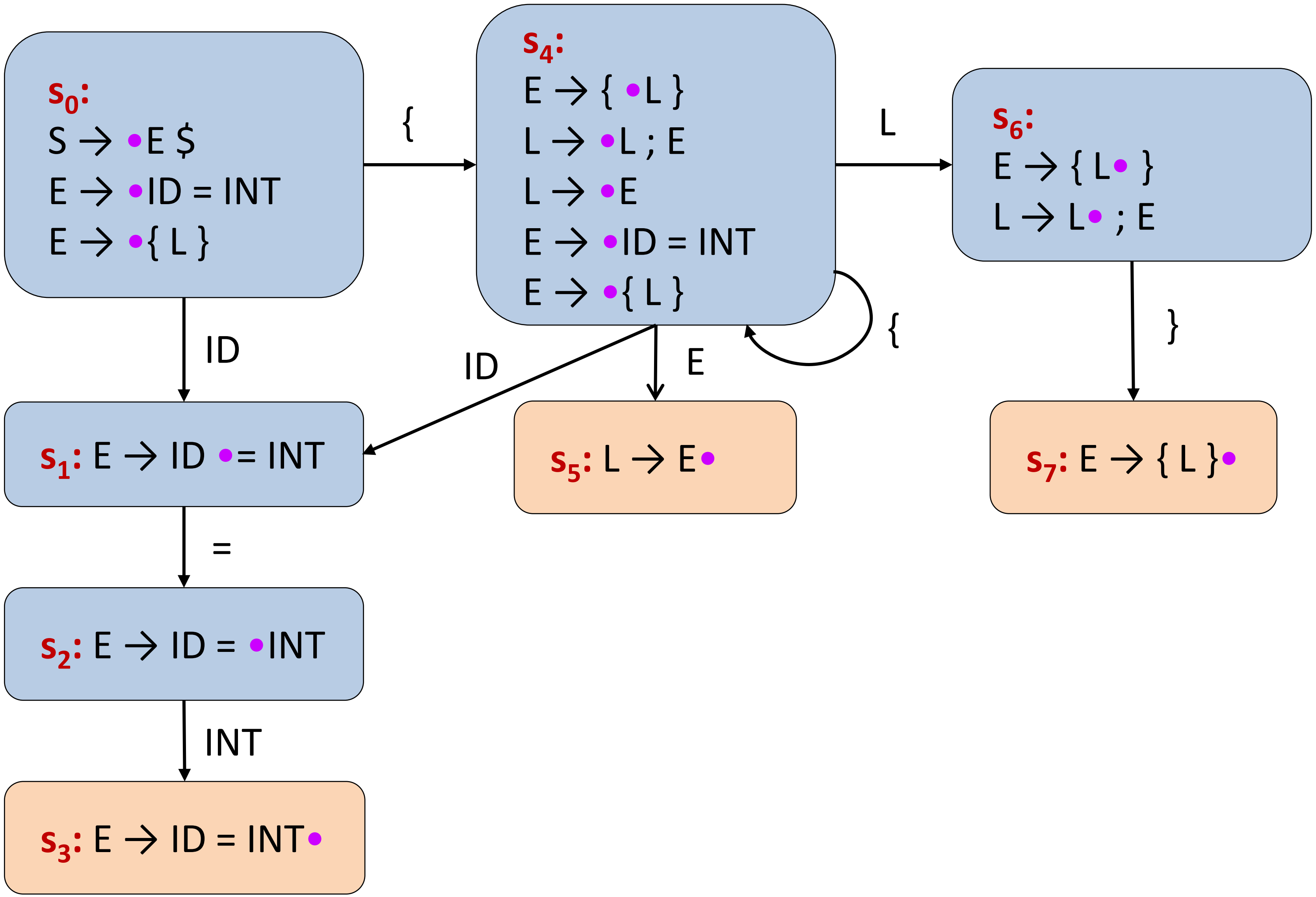
- From s_6, if we recognize }, then the next state contains:

  E → { L }•

- Which is a reduce state

s_6:
E → { L • }
L → L • ; E

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E



s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
ID
ID
E
{
}
s1: E → ID •= INT
s5: L → E•
s7: E → { L }•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_6, if we recognize ;, then the next state contains:

  L → L ; •E

- The next state is the closure of this item:

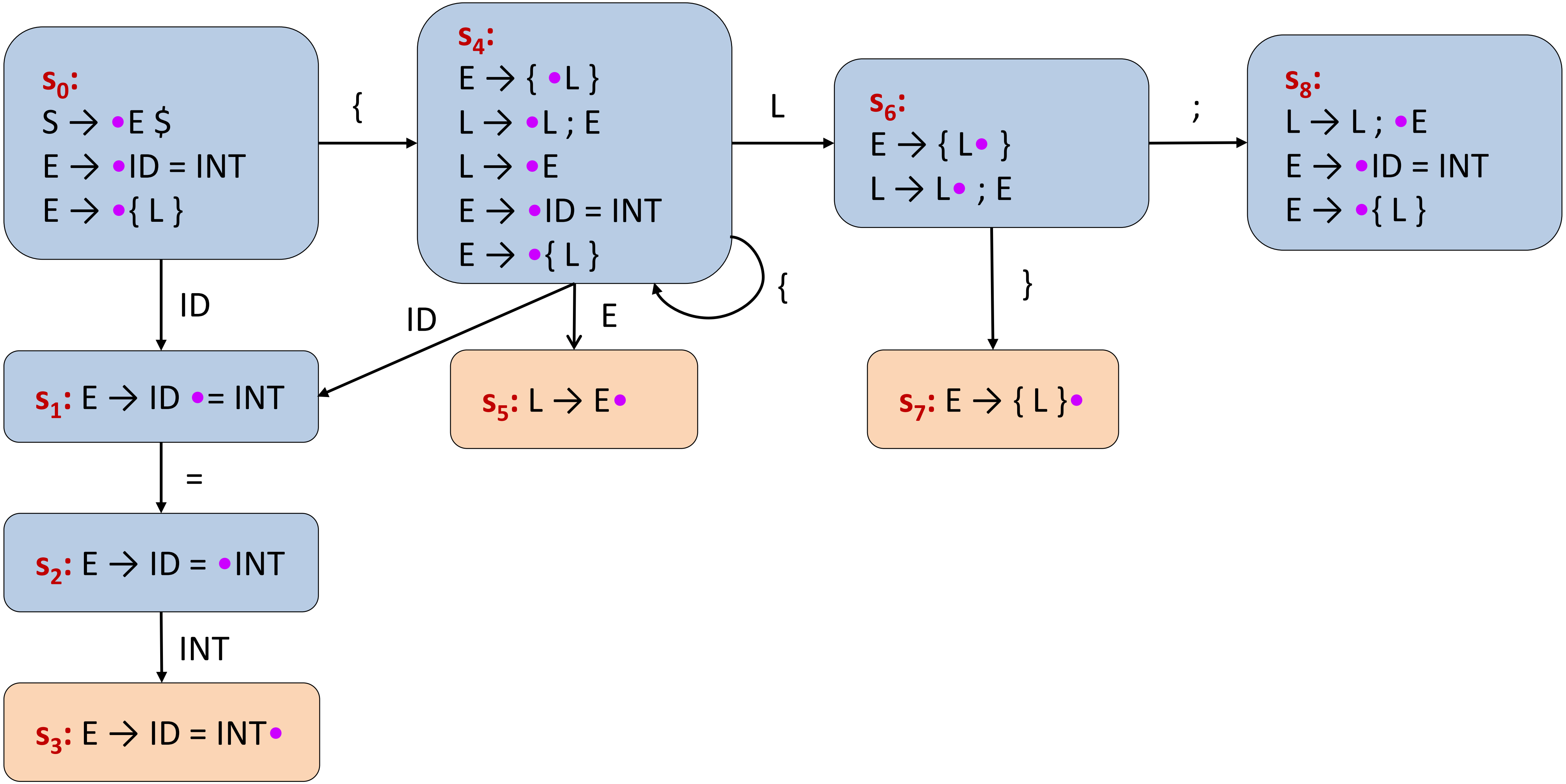
  L → L ; •E

  E → •ID = INT

  E → •{ L }

s_6:
E → { L• }
L → L• ; E

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E



s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
;
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
ID
ID
E
{
}
s1: E → ID •= INT
s5: L → E•
s7: E → { L }•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

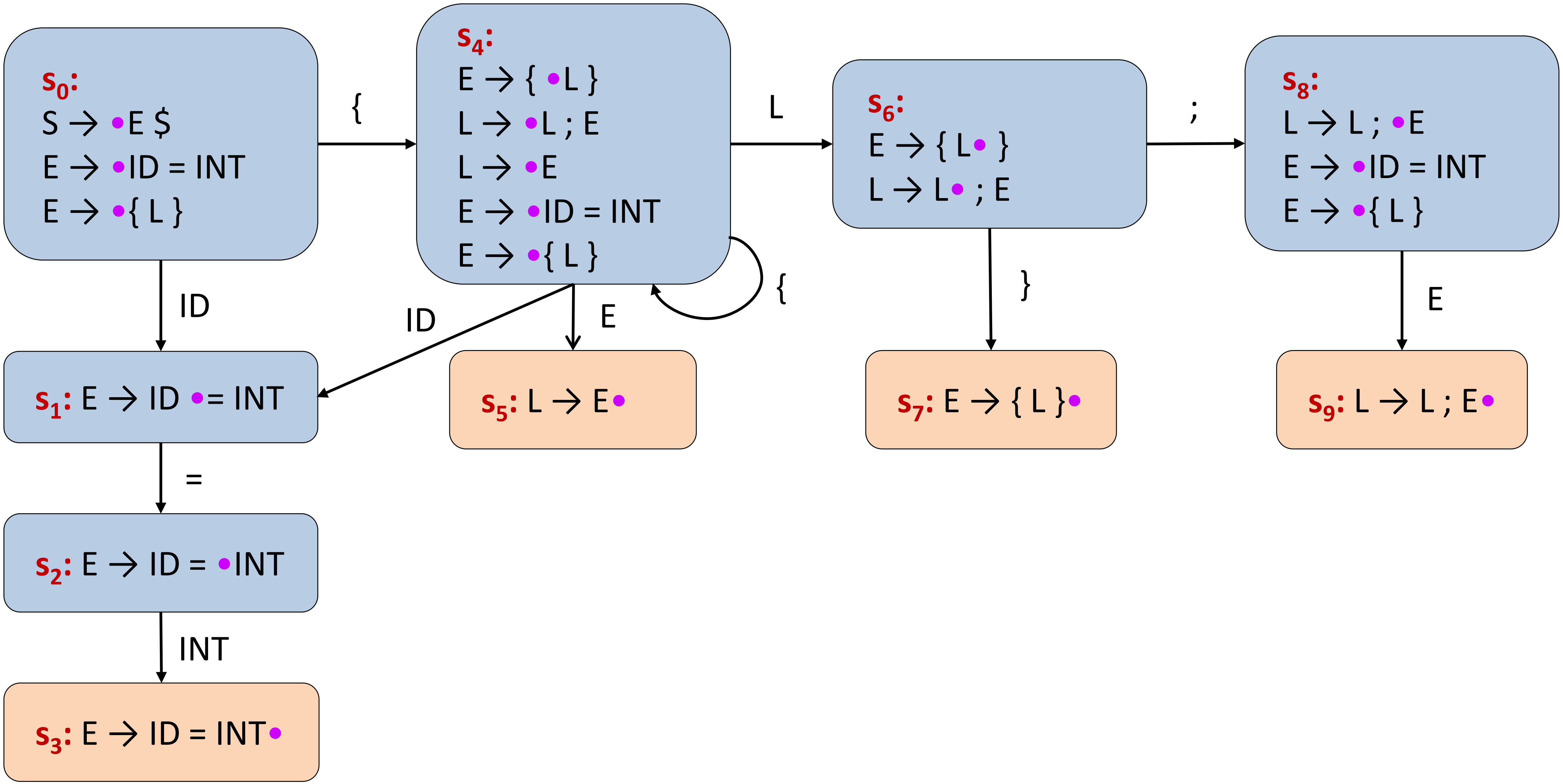
- From s_8, if we recognize E, then the next state contains:

  L → L ; E •

- which is a reduce state

s_8:
L → L ; • E
E → • ID = INT
E → • { L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
;
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
ID
ID
E
{
}
E
s1: E → ID •= INT
s5: L → E•
s7: E → { L }•
s9: L → L ; E•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_8, if we recognize {, then the next state contains:

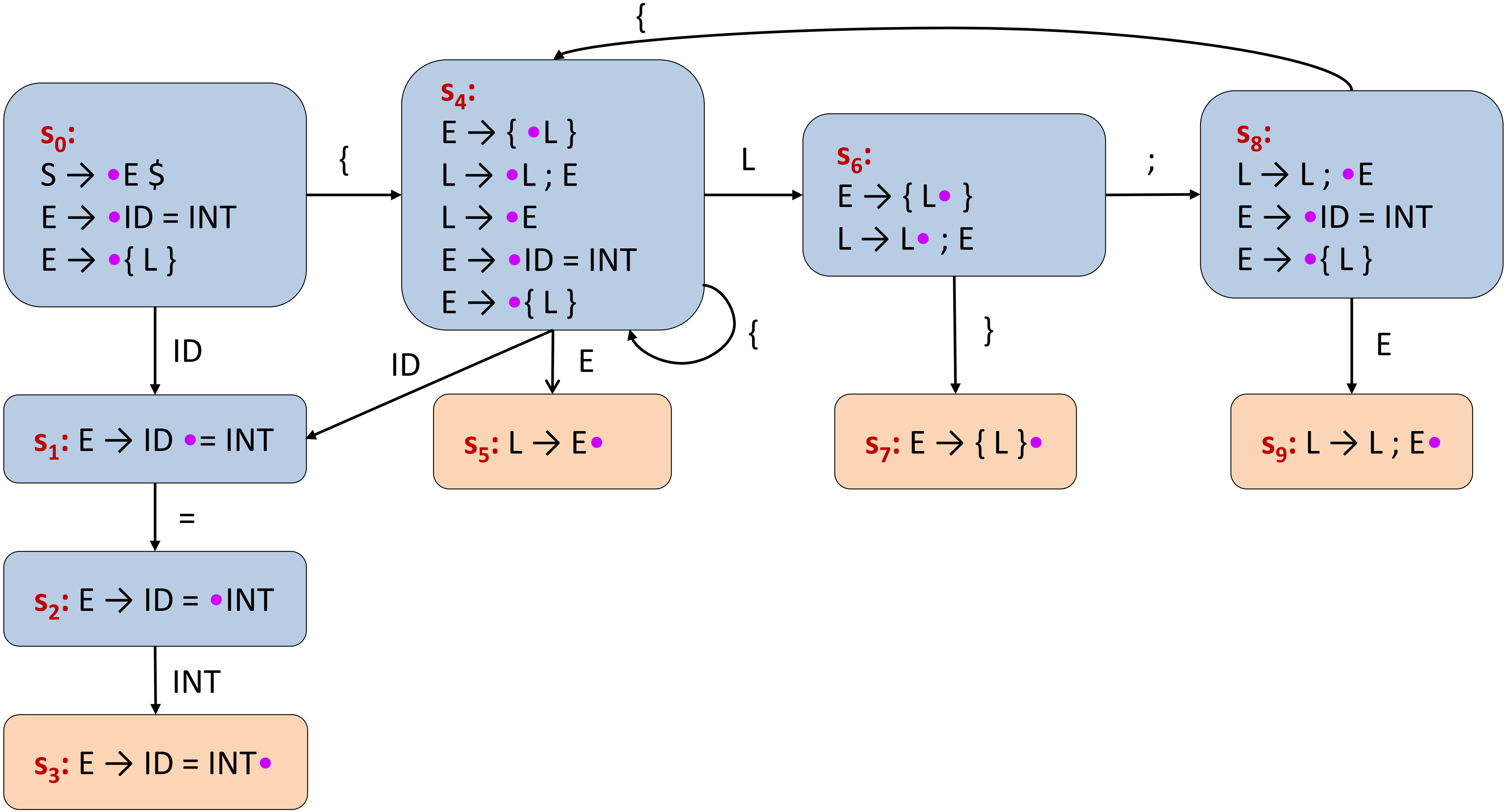
  E → {• L }

- The next state is the closure of this item:

$$\left.\begin{array}{l} E \rightarrow \{ \bullet L \} \\ L \rightarrow \bullet L \, ; E \\ L \rightarrow \bullet E \\ E \rightarrow \bullet ID = INT \\ E \rightarrow \bullet \{ L \} \end{array}\right\} s_4$$

s_8:
L → L ; •E
E → •ID = INT
E → •{ L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
s6:
E → { L• }
L → L• ; E
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
s1: E → ID •= INT
s2: E → ID = •INT
s3: E → ID = INT•
s5: L → E•
s7: E → { L }•
s9: L → L ; E•
{
{
{
L
;
ID
ID
E
}
E
=
INT

LR(0) Automaton

- From s_8, if we recognize ID, then the next state contains:

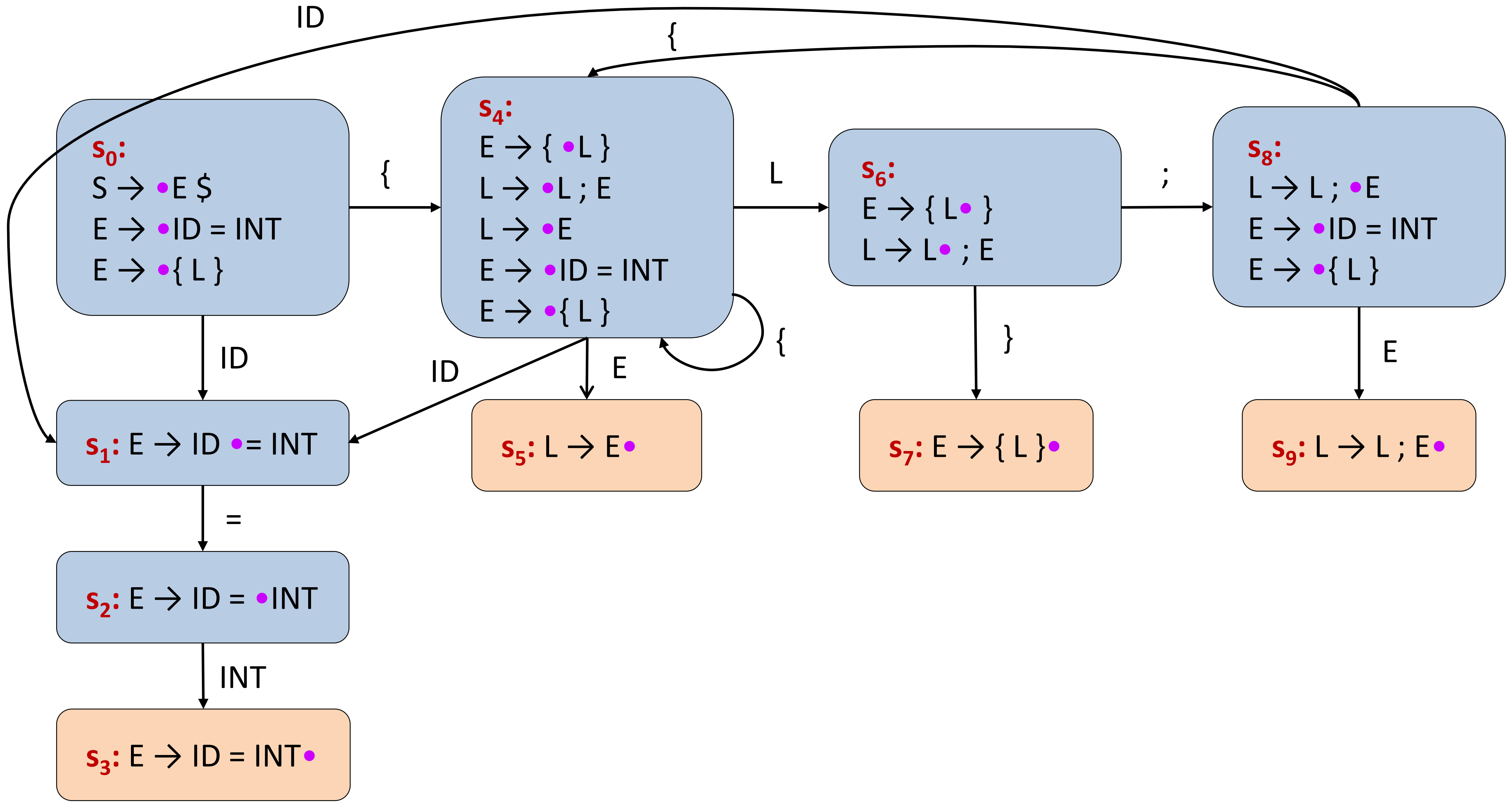
  E → ID • = INT

- The next state is the closure of this item:

  $E \rightarrow ID \bullet = INT \;\}\; s_1$

s_8:
L → L ; • E
E → • ID = INT
E → • { L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


ID
{
s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
;
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
ID
ID
E
{
}
E
s1: E → ID •= INT
s5: L → E•
s7: E → { L }•
s9: L → L ; E•
=
s2: E → ID = •INT
INT
s3: E → ID = INT•

LR(0) Automaton

- From s_0, if we recognize E, then the next state contains:

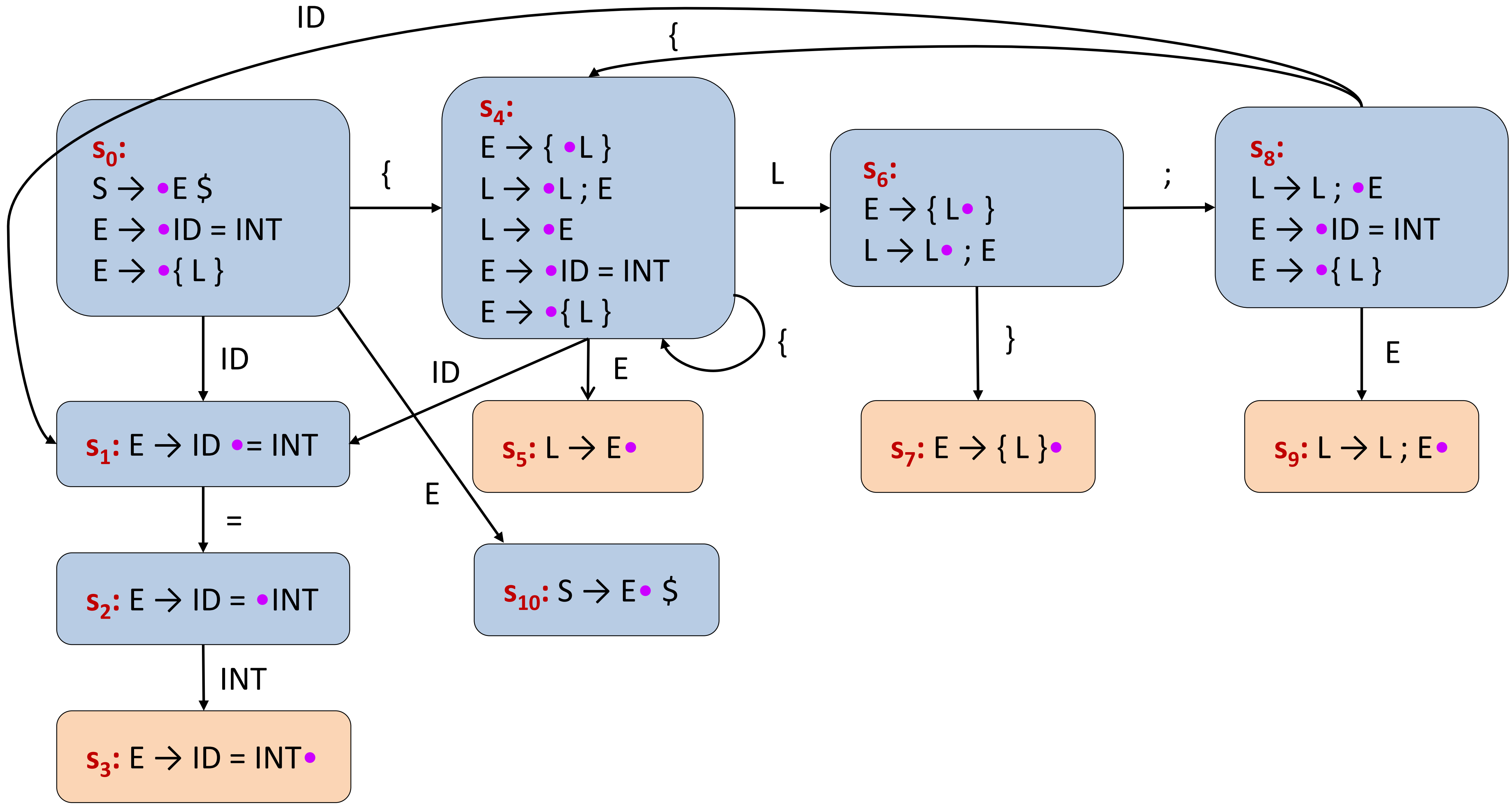
  S → E• $

- The next state is the closure of this item:

  S → E• $

s_0:
S → •E $
E → •ID = INT
E → •{ L }

S → E $
E → ID = INT
E → { L }
L → E
L → L ; E


s0:
S → •E $
E → •ID = INT
E → •{ L }
s1: E → ID •= INT
s2: E → ID = •INT
s3: E → ID = INT•
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
s5: L → E•
s6:
E → { L• }
L → L• ; E
s7: E → { L }•
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
s9: L → L ; E•
s10: S → E• $
ID
{
L
;
=
INT
E
}

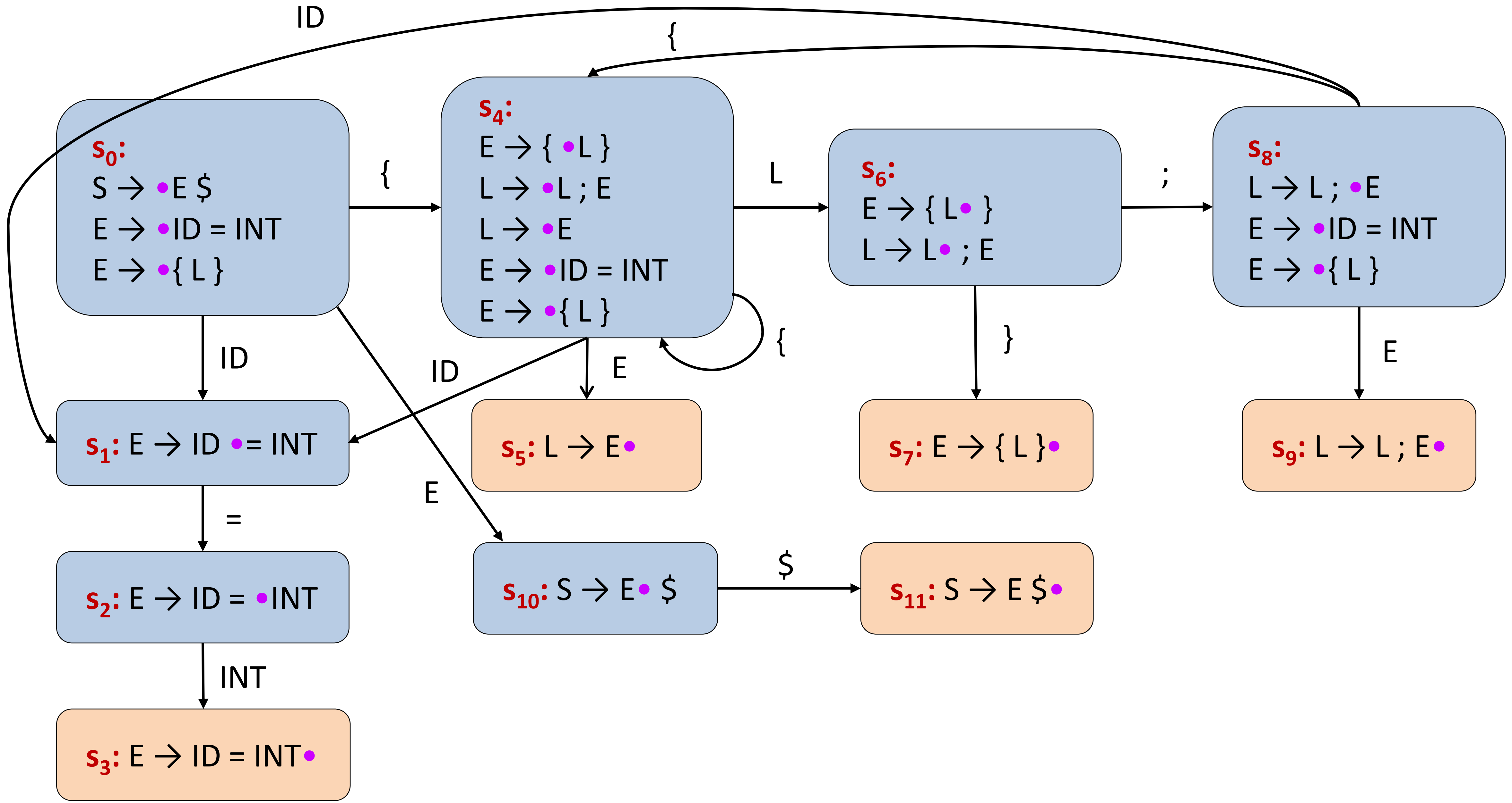
LR(0) Automaton

- From s_{10}, if we recognize \$, then the next state contains :

  $S \rightarrow E \, \$ \bullet$
- Which is a reduce state

s_{10}: $S \rightarrow E \bullet \$$

$S \rightarrow E \, \$$
$E \rightarrow ID = INT$
$E \rightarrow \{ \, L \, \}$
$L \rightarrow E$
$L \rightarrow L \, ; \, E$



s0:
S → •E $
E → •ID = INT
E → •{ L }
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
s6:
E → { L• }
L → L• ; E
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
s1: E → ID •= INT
s2: E → ID = •INT
s3: E → ID = INT•
s5: L → E•
s7: E → { L }•
s9: L → L ; E•
s10: S → E• $
s11: S → E $•
ID
{
L
;
E
}
=
INT
$

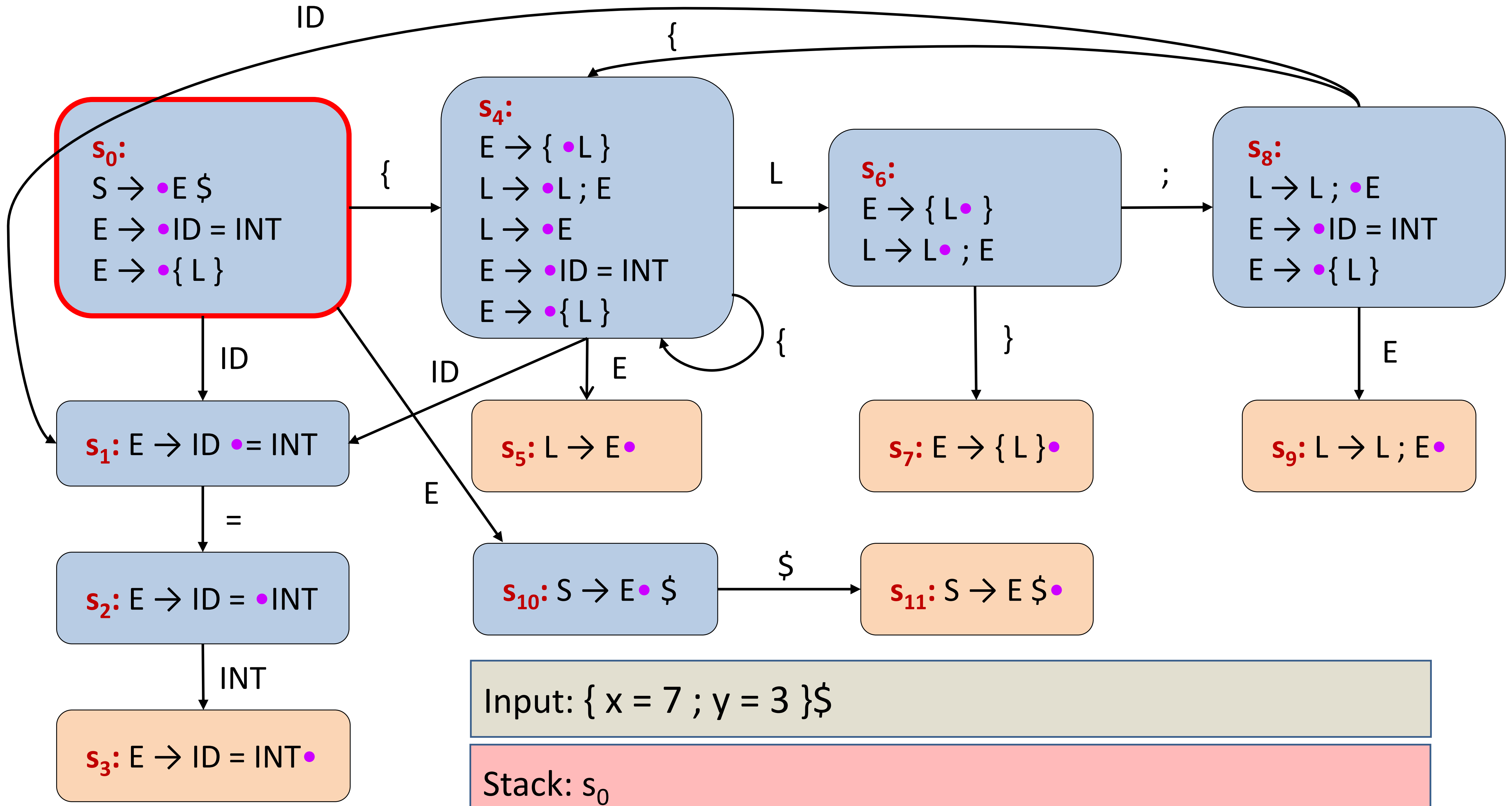
Shift & Reduce

Two possible operations:

- **Shift:** Read the next token from the input, push it onto the stack and, transition to the next state
- **Reduce:** The top of the stack matches the right-hand side of a rule, replace it with the left-hand side and transition to the appropriate state

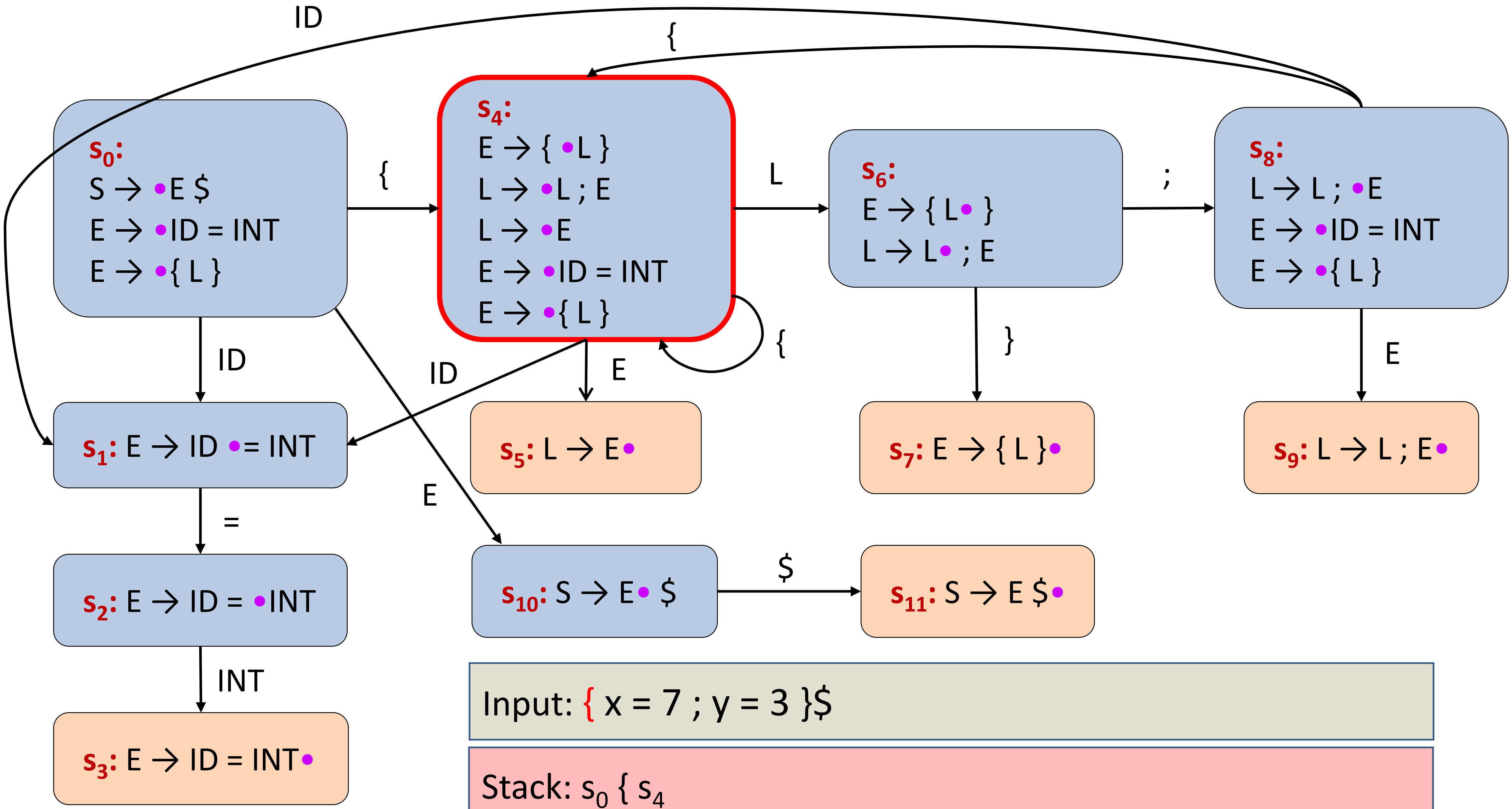
LR(0) Parser – Example Run

- Parsing the input:
  - { x = 7 ; y = 3 }$



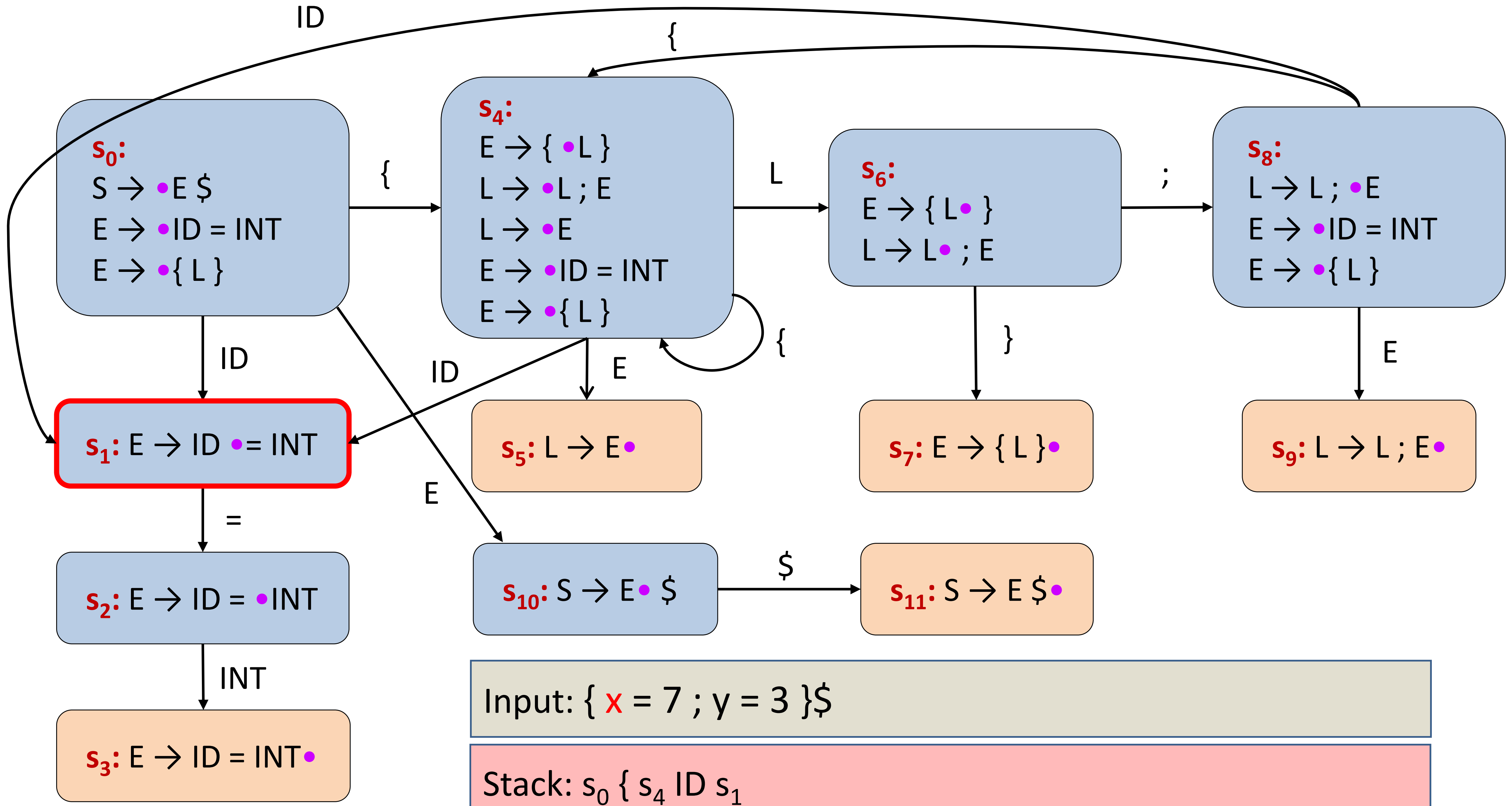
Input: { x = 7 ; y = 3 }$

Stack: s_0



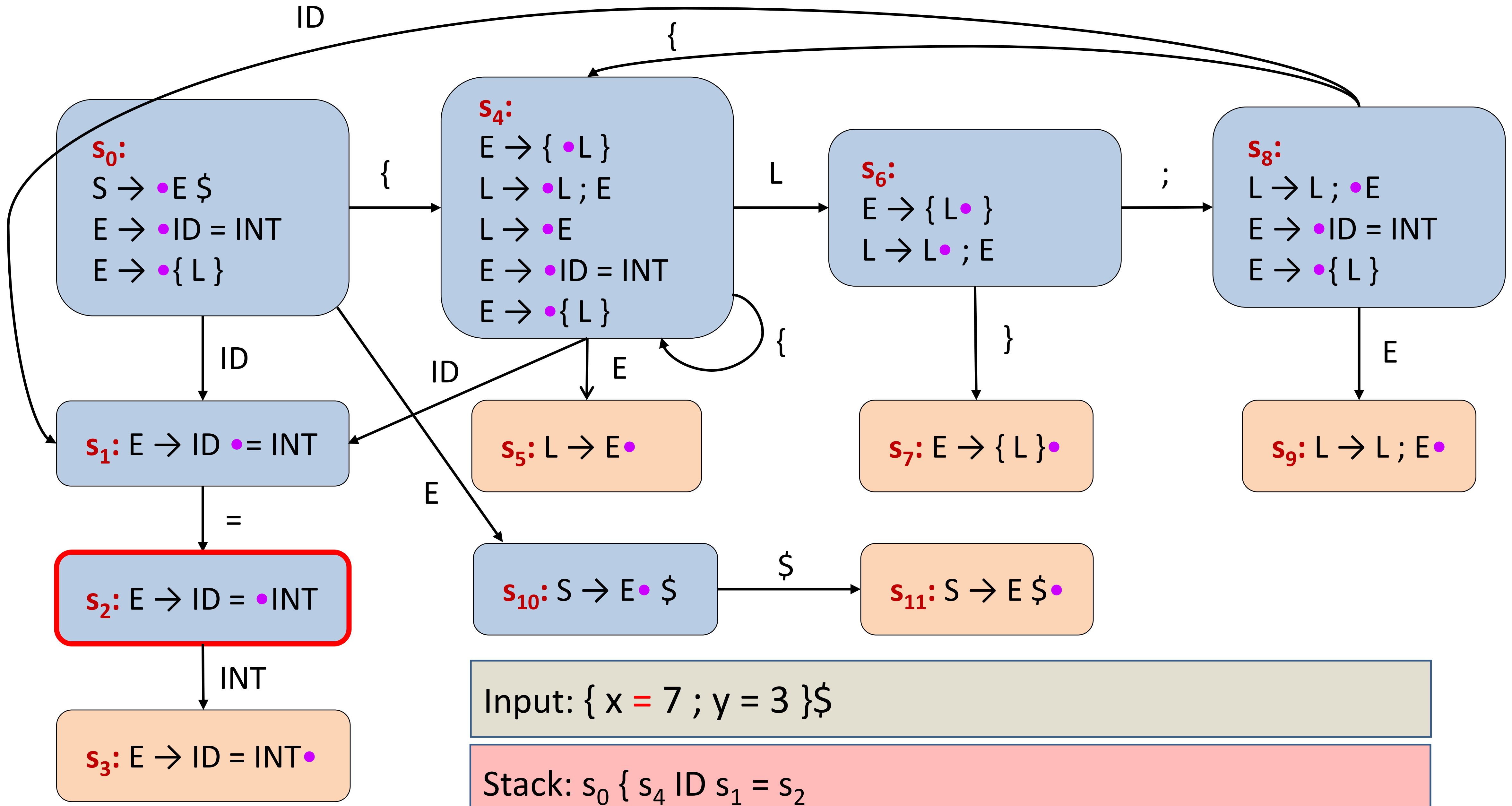
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4



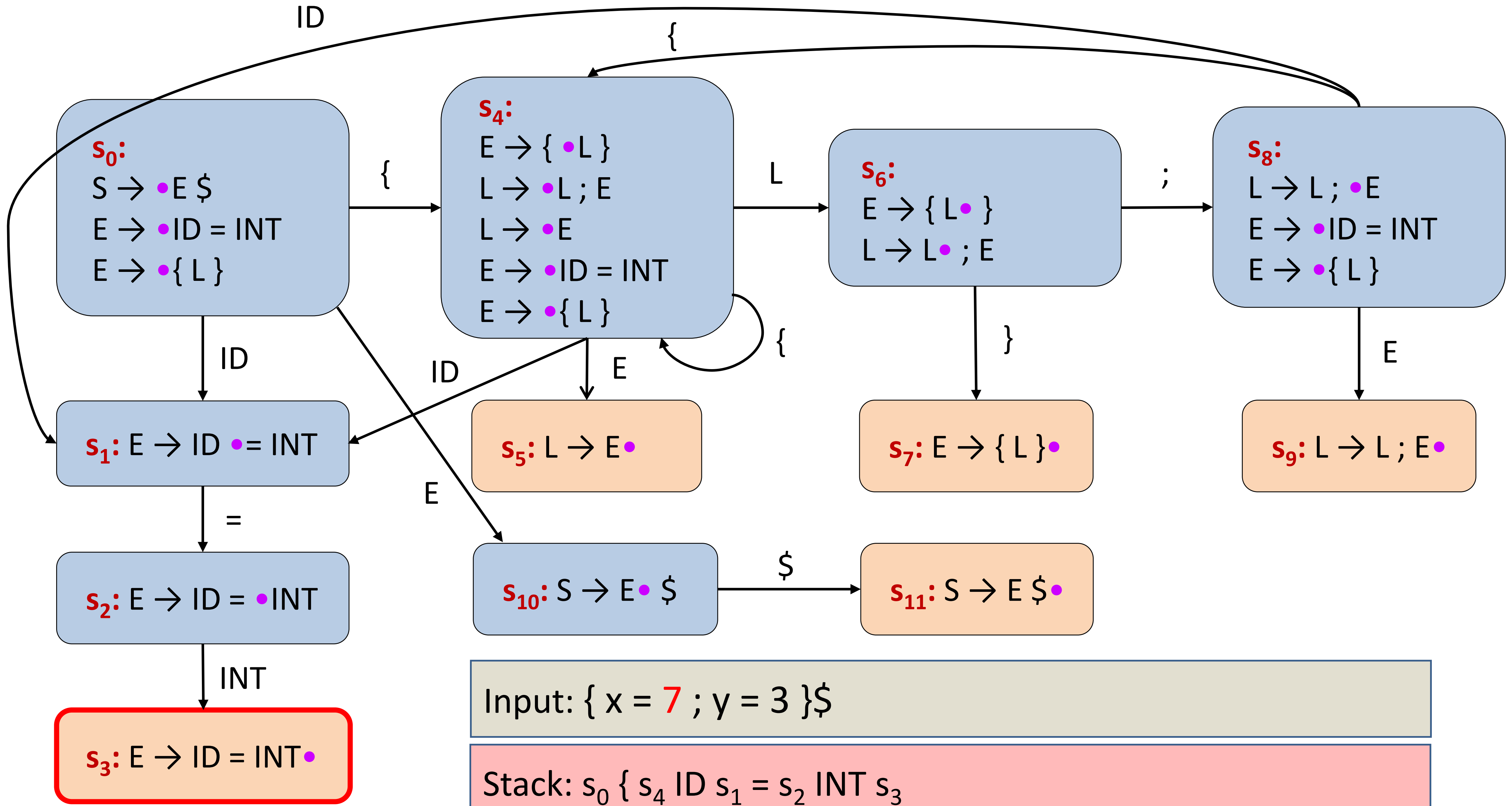
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 ID s_1



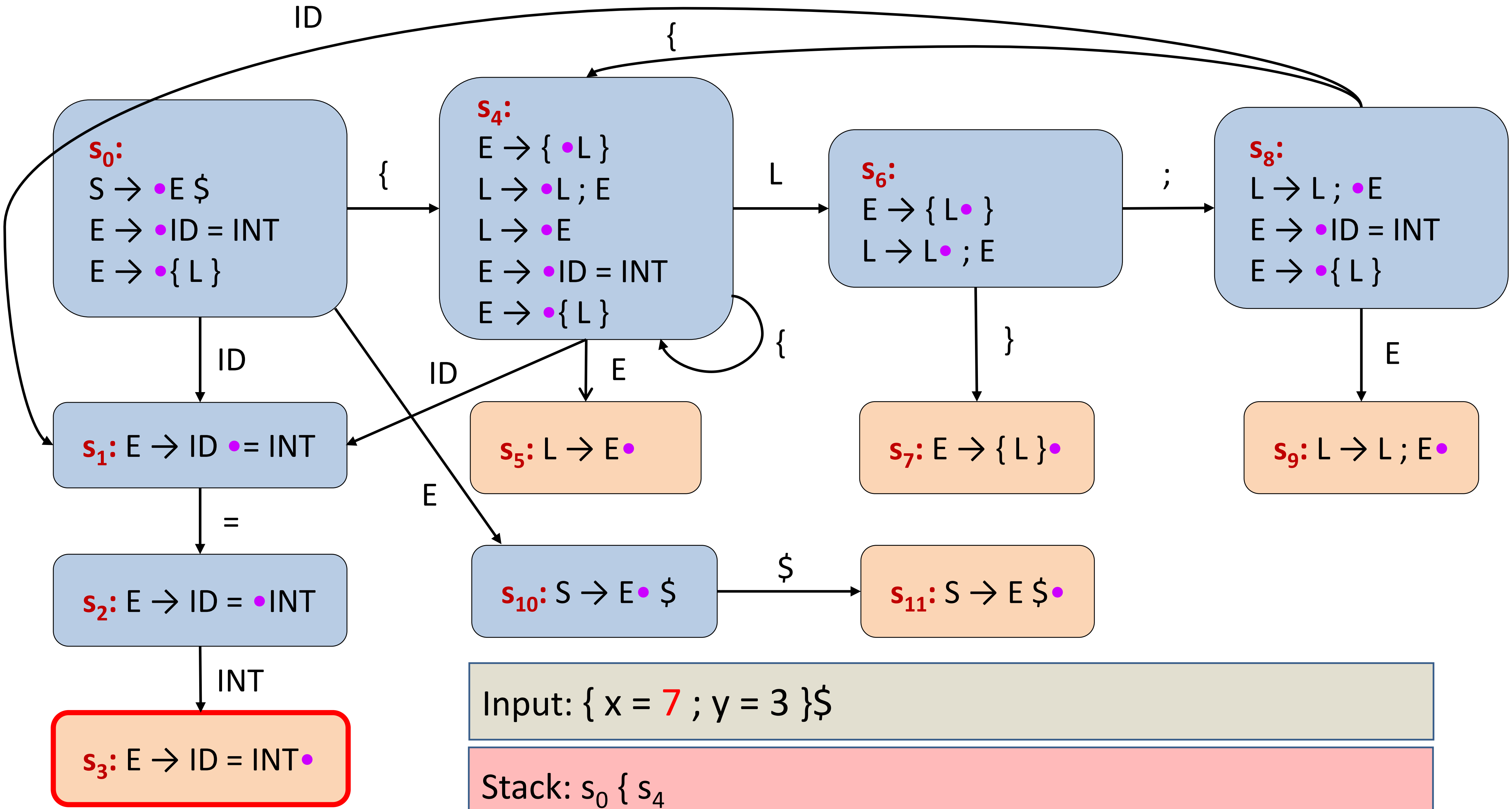
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 ID s_1 = s_2



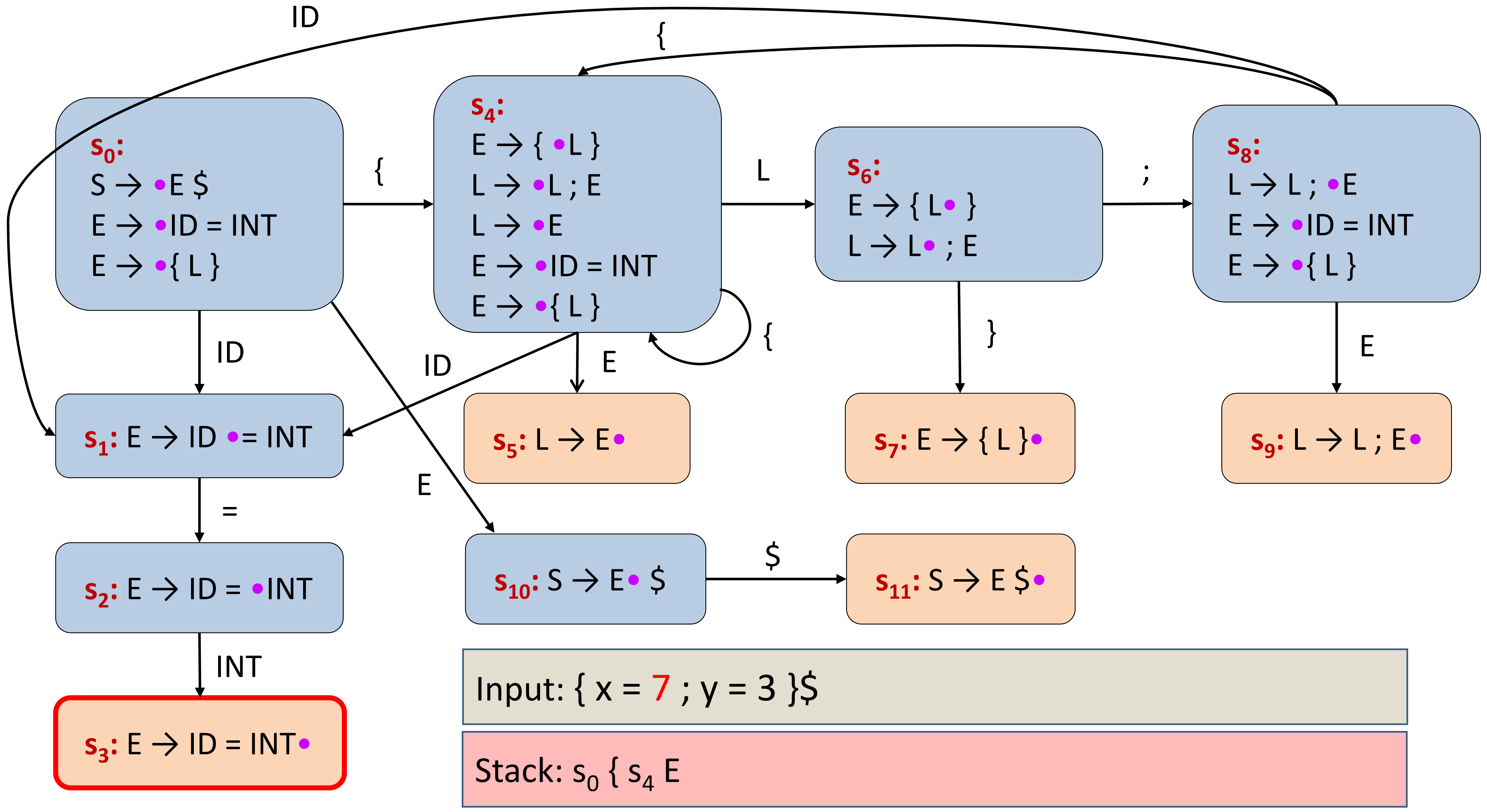
Input: { x = 7 ; y = 3 }$

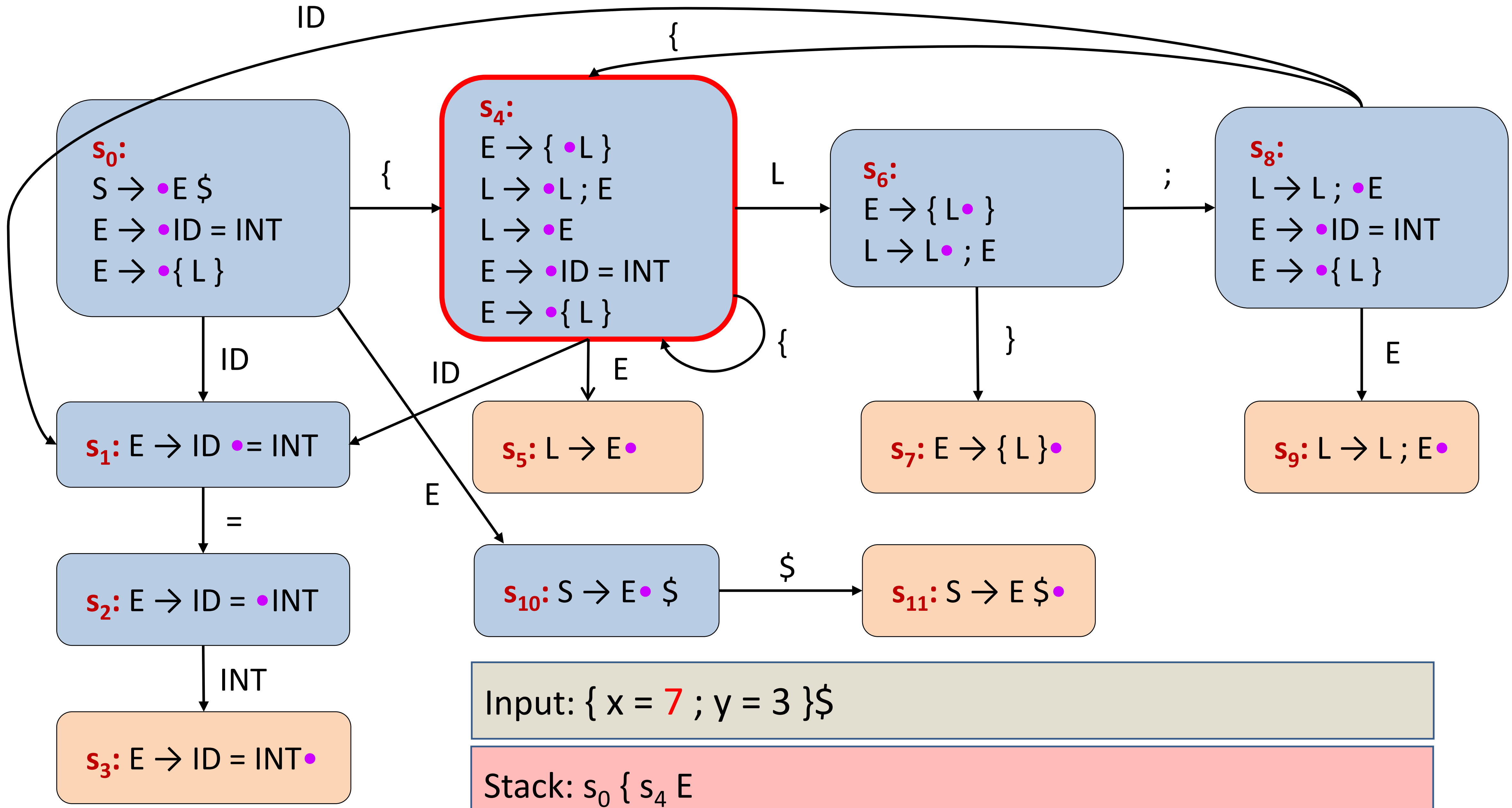
Stack: s_0 { s_4 ID s_1 = s_2 INT s_3



Input: { x = 7 ; y = 3 }$

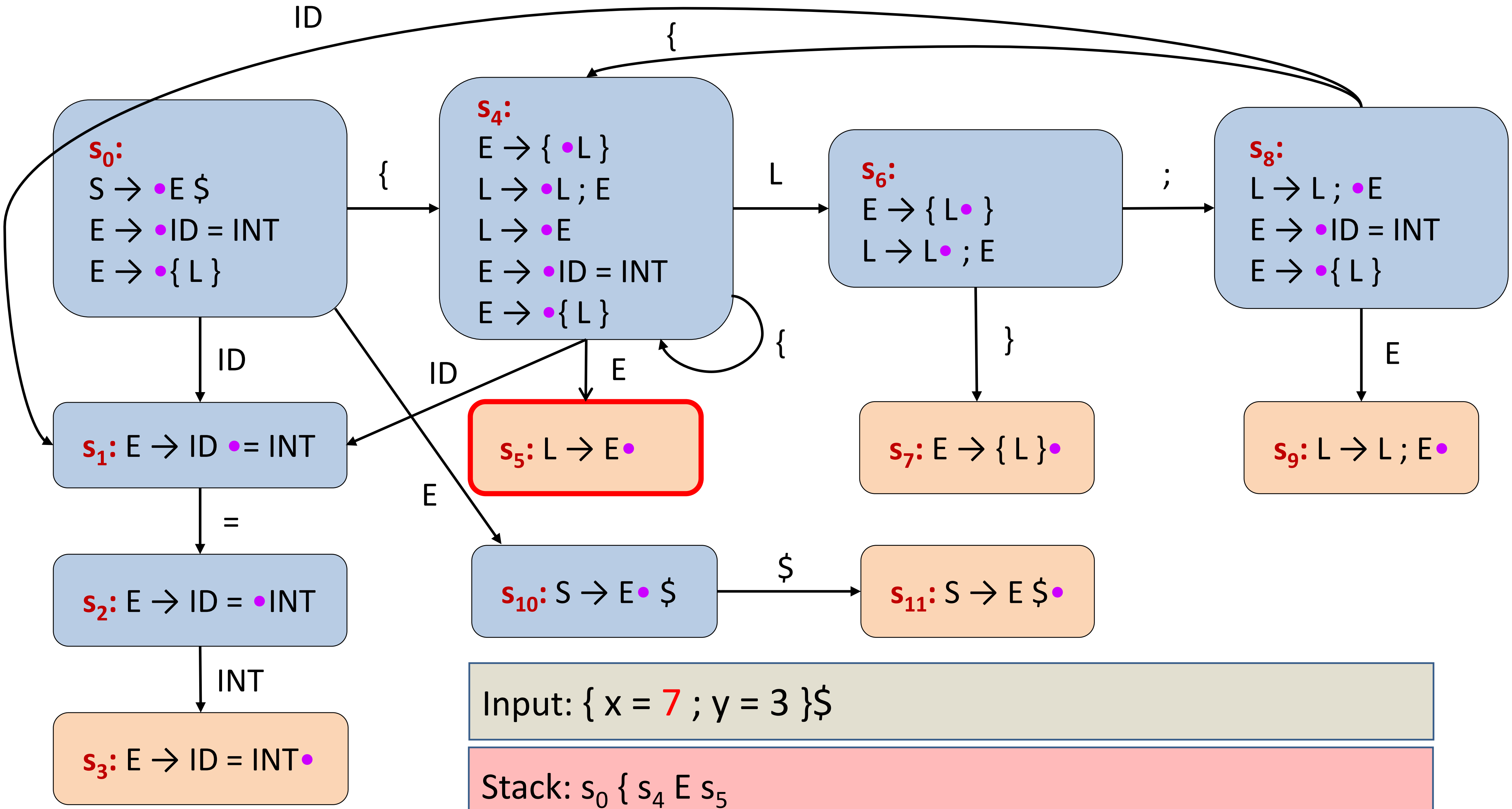
Stack: s_0 { s_4


s0:
S → •E $
E → •ID = INT
E → •{ L }
s1: E → ID •= INT
s2: E → ID = •INT
s3: E → ID = INT•
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
s5: L → E•
s6:
E → { L• }
L → L• ; E
s7: E → { L }•
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
s9: L → L ; E•
s10: S → E• $
s11: S → E $•
ID
{
L
;
}
E
=
INT
$
Input: { x = 7 ; y = 3 }$
Stack: s0 { s4 E



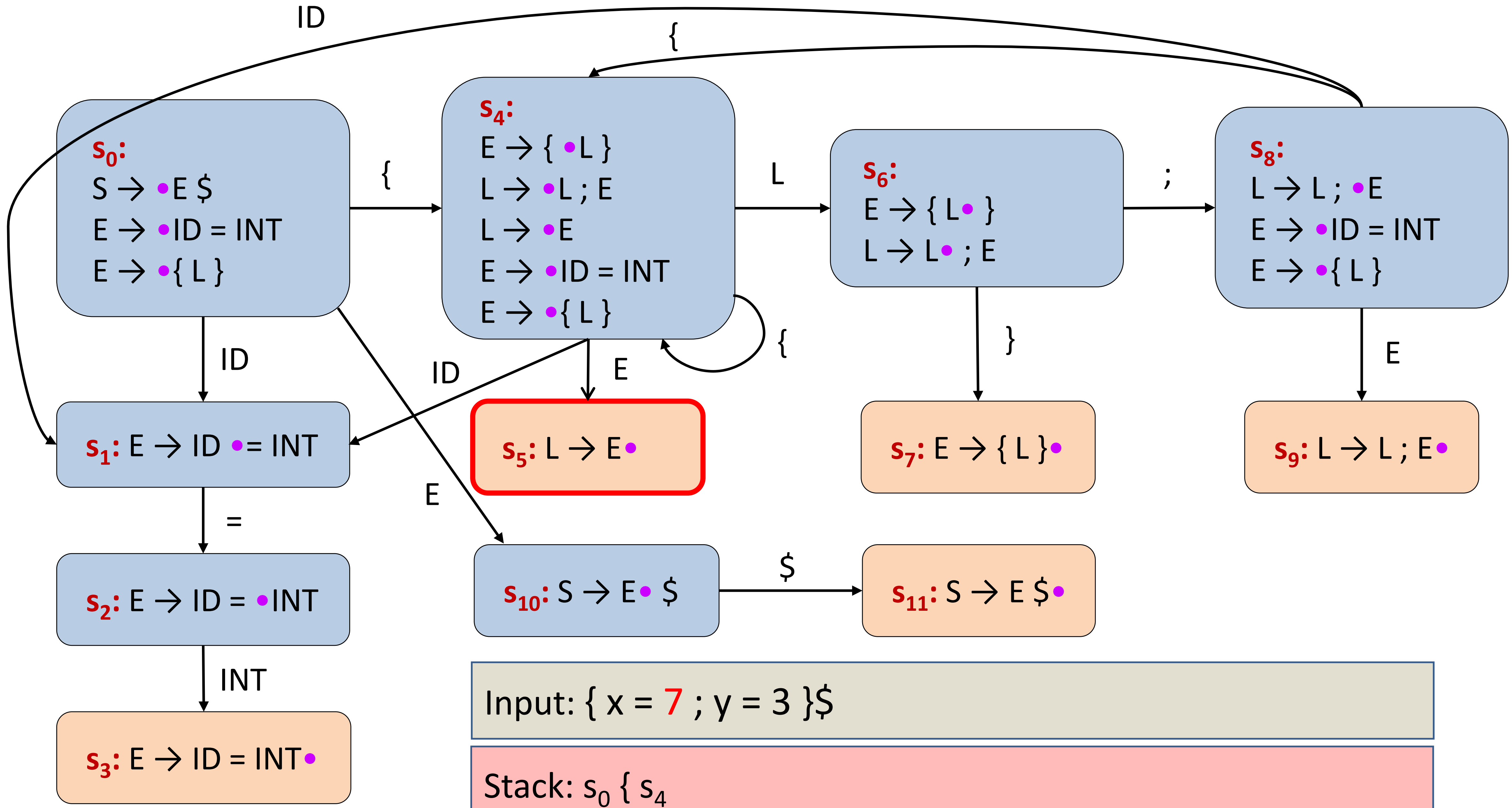
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 E



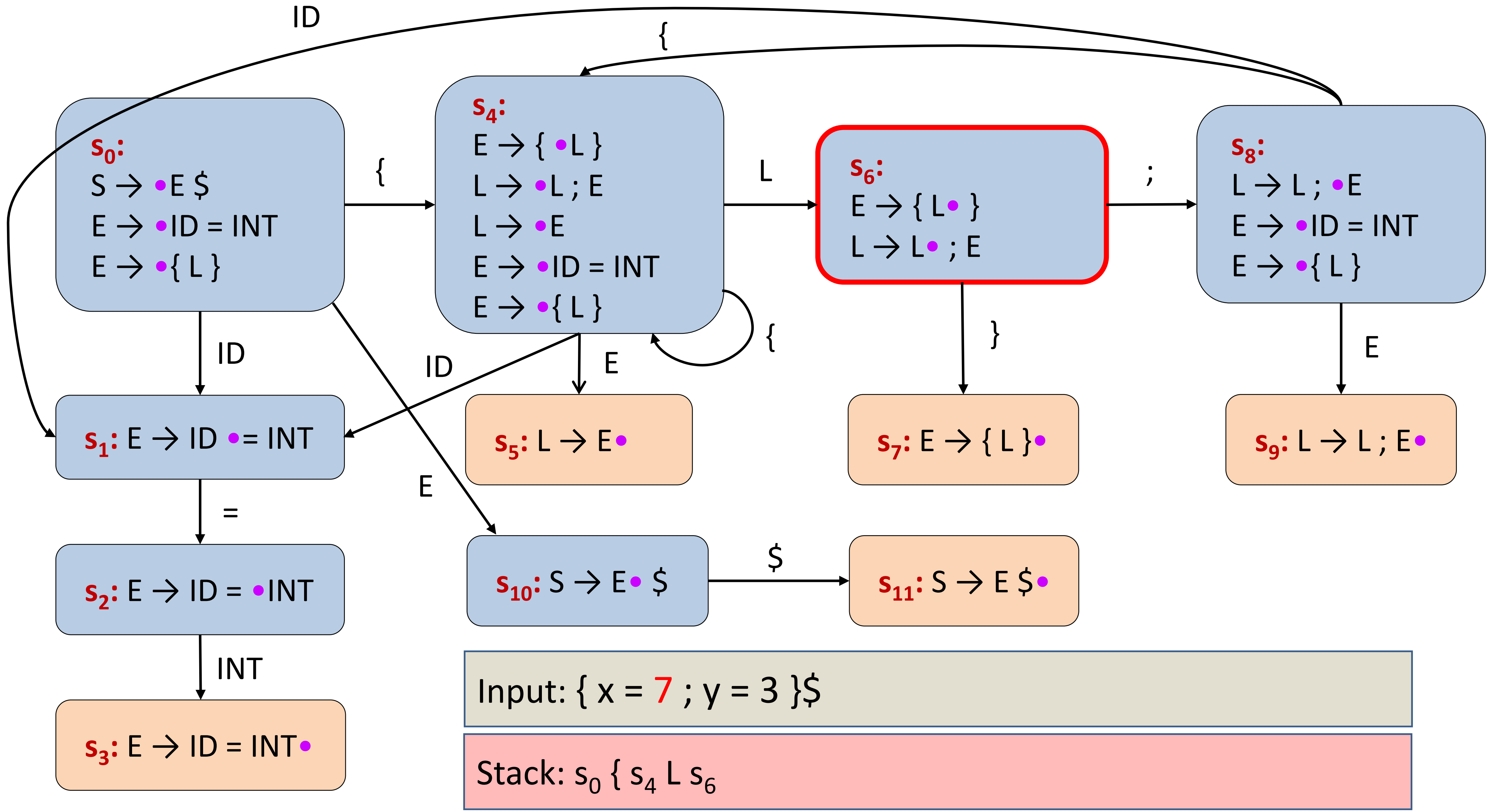
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 E s_5

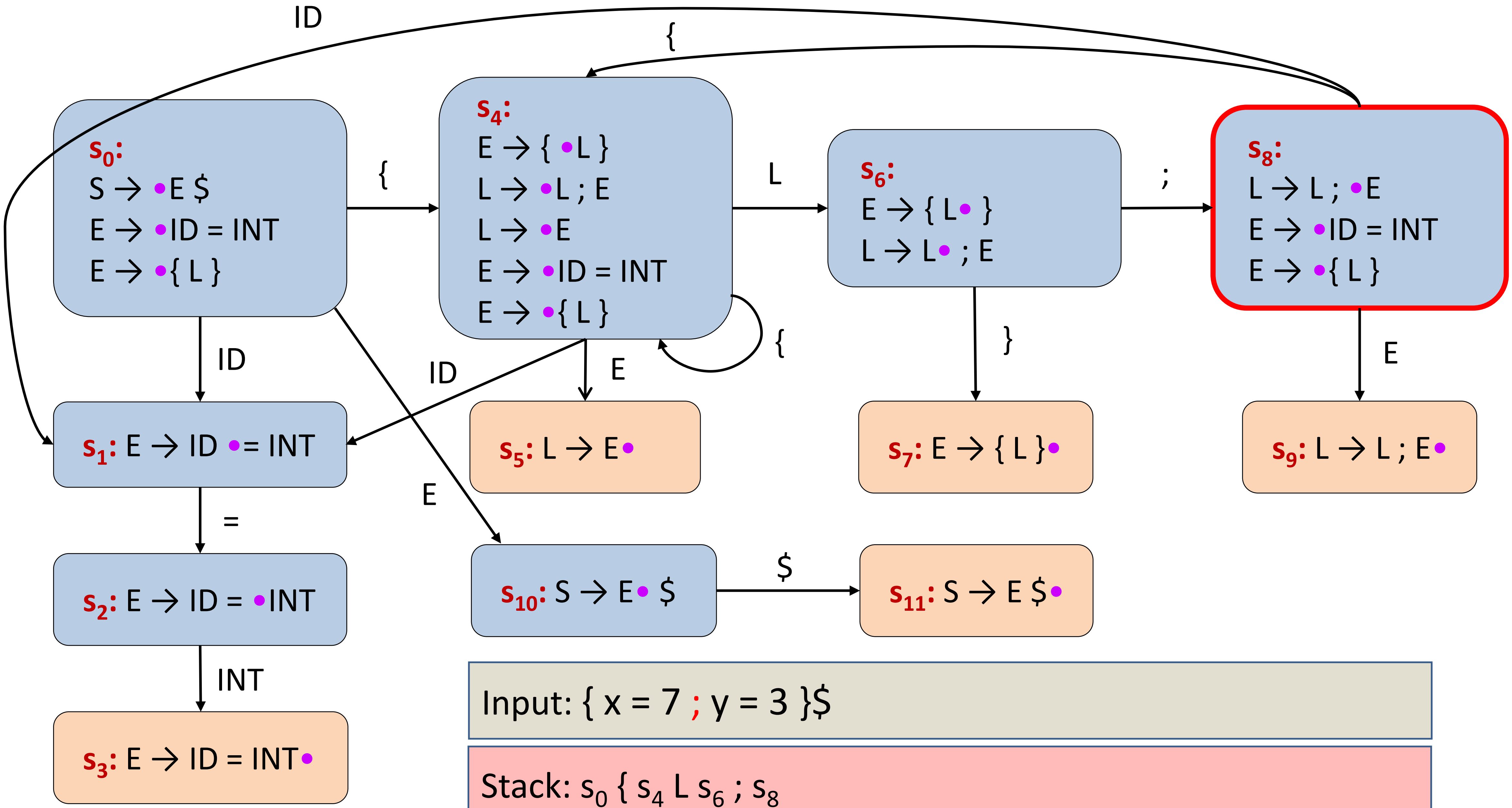


Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4

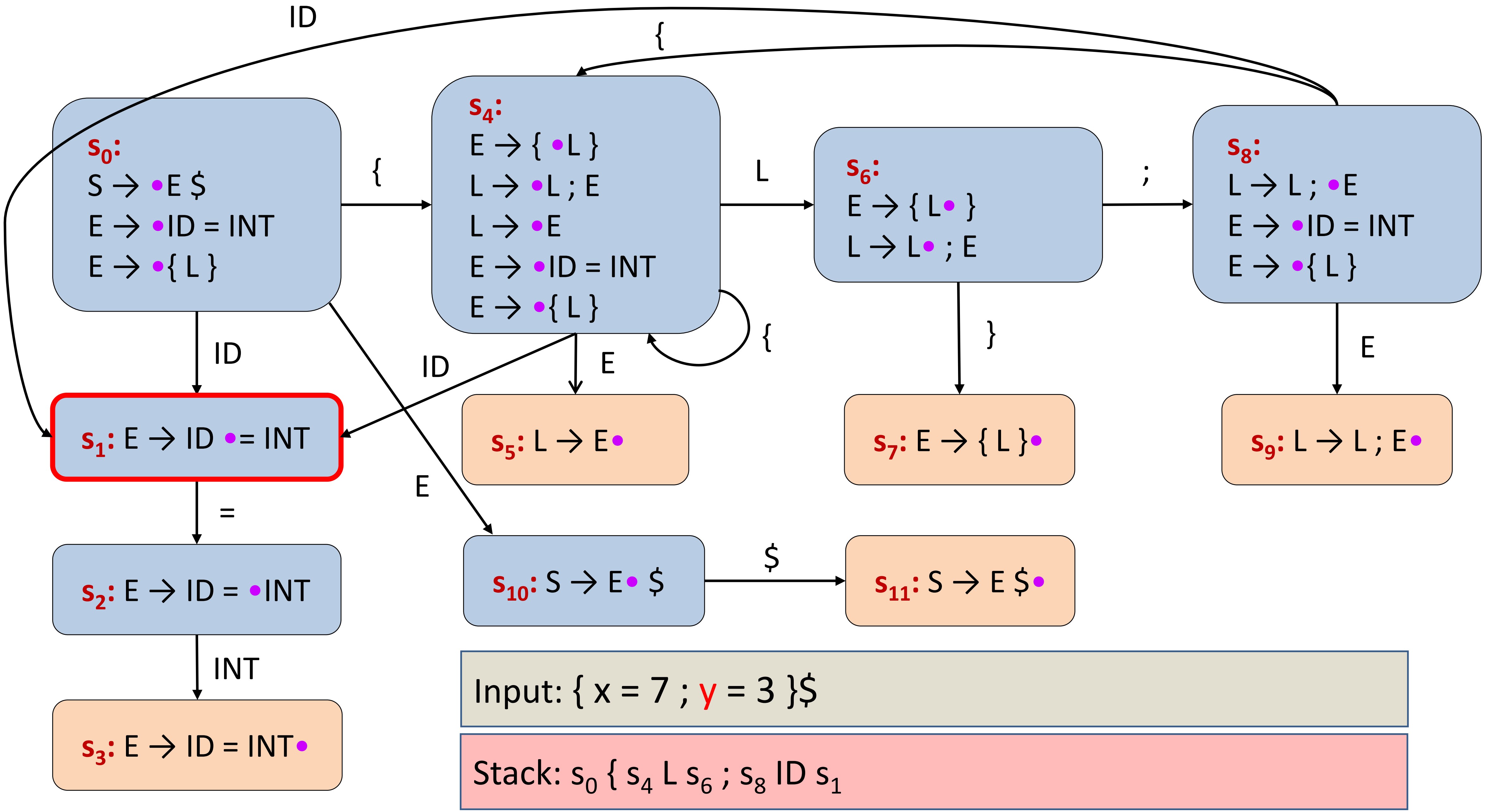


s0:
S → •E $
E → •ID = INT
E → •{ L }
s1: E → ID •= INT
s2: E → ID = •INT
s3: E → ID = INT•
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
s5: L → E•
s6:
E → { L• }
L → L• ; E
s7: E → { L }•
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
s9: L → L ; E•
s10: S → E• $
s11: S → E $•
ID
{
L
;
}
E
=
INT
$
Input: { x = 7 ; y = 3 }$
Stack: s0 { s4 L s6



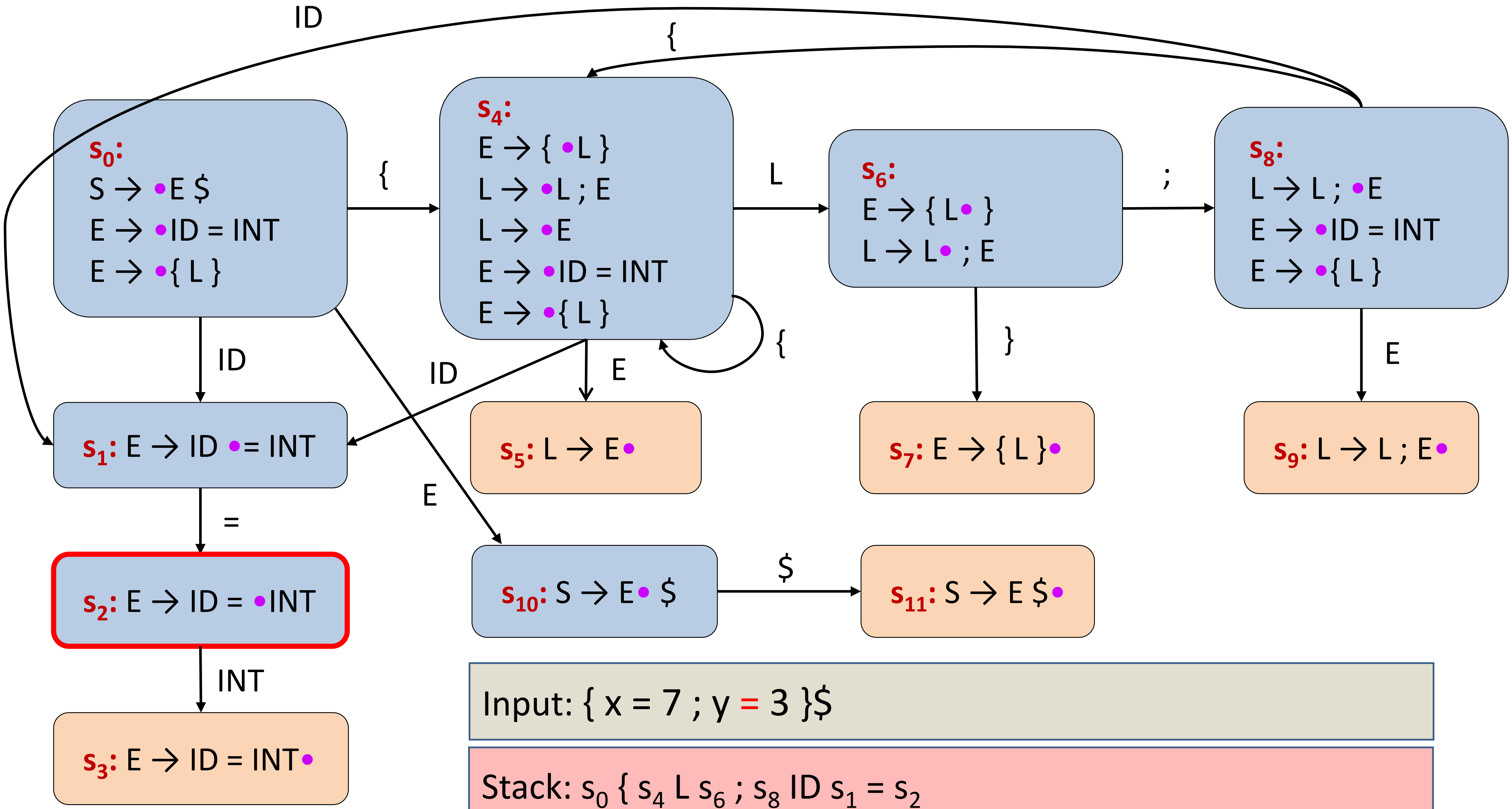
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 ; s_8



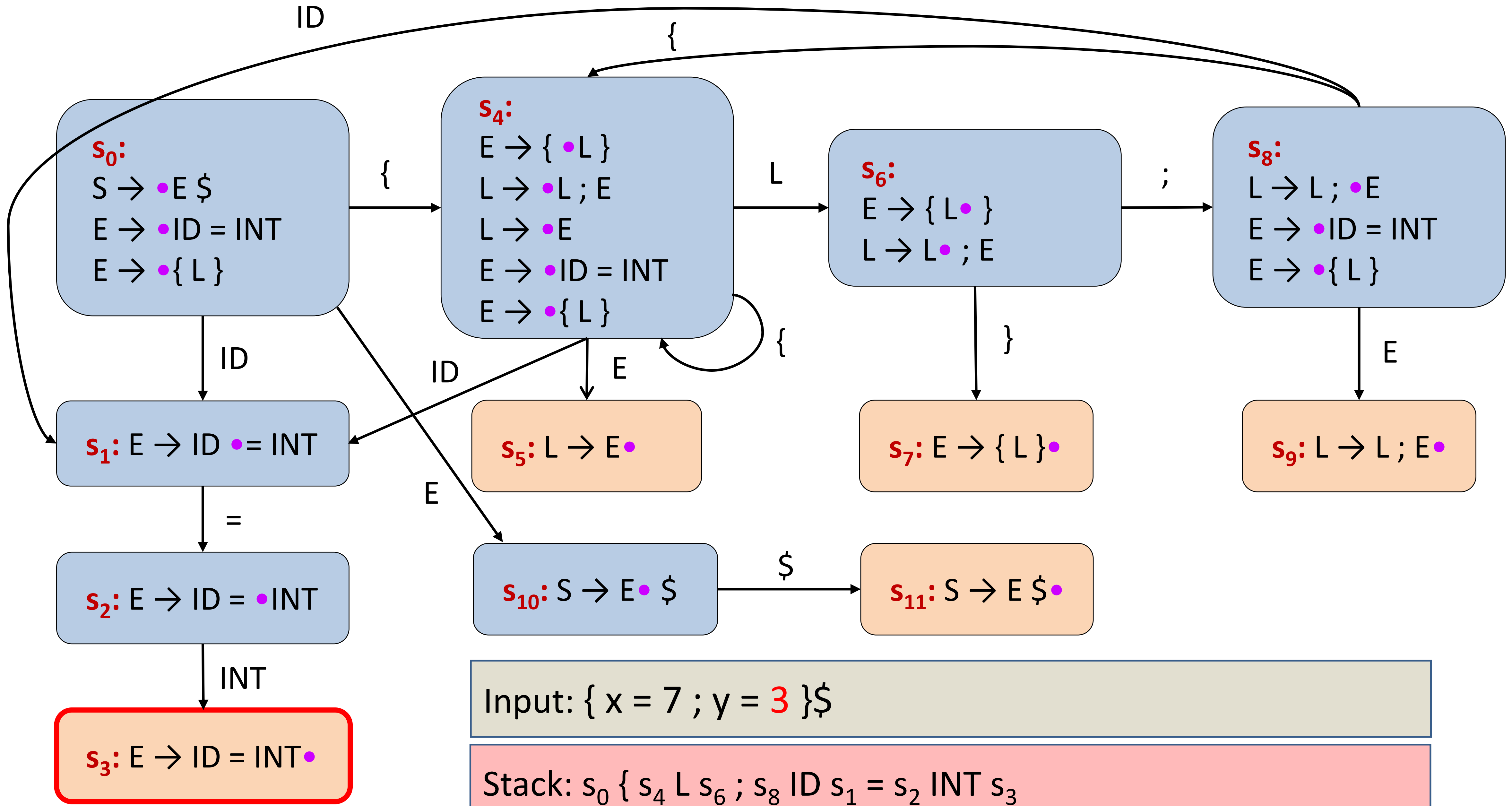
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 ; s_8 ID s_1



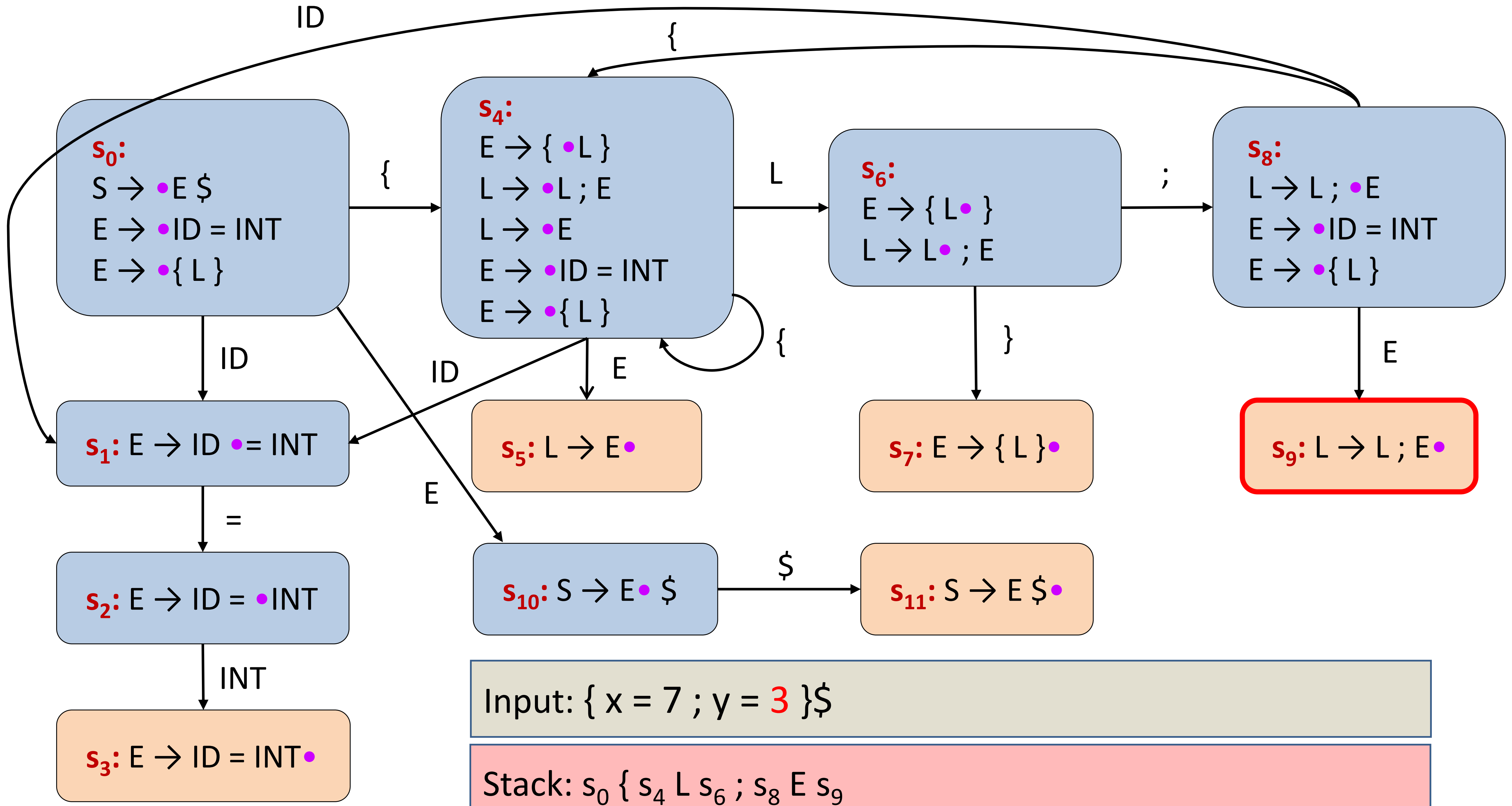
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 ; s_8 ID s_1 = s_2



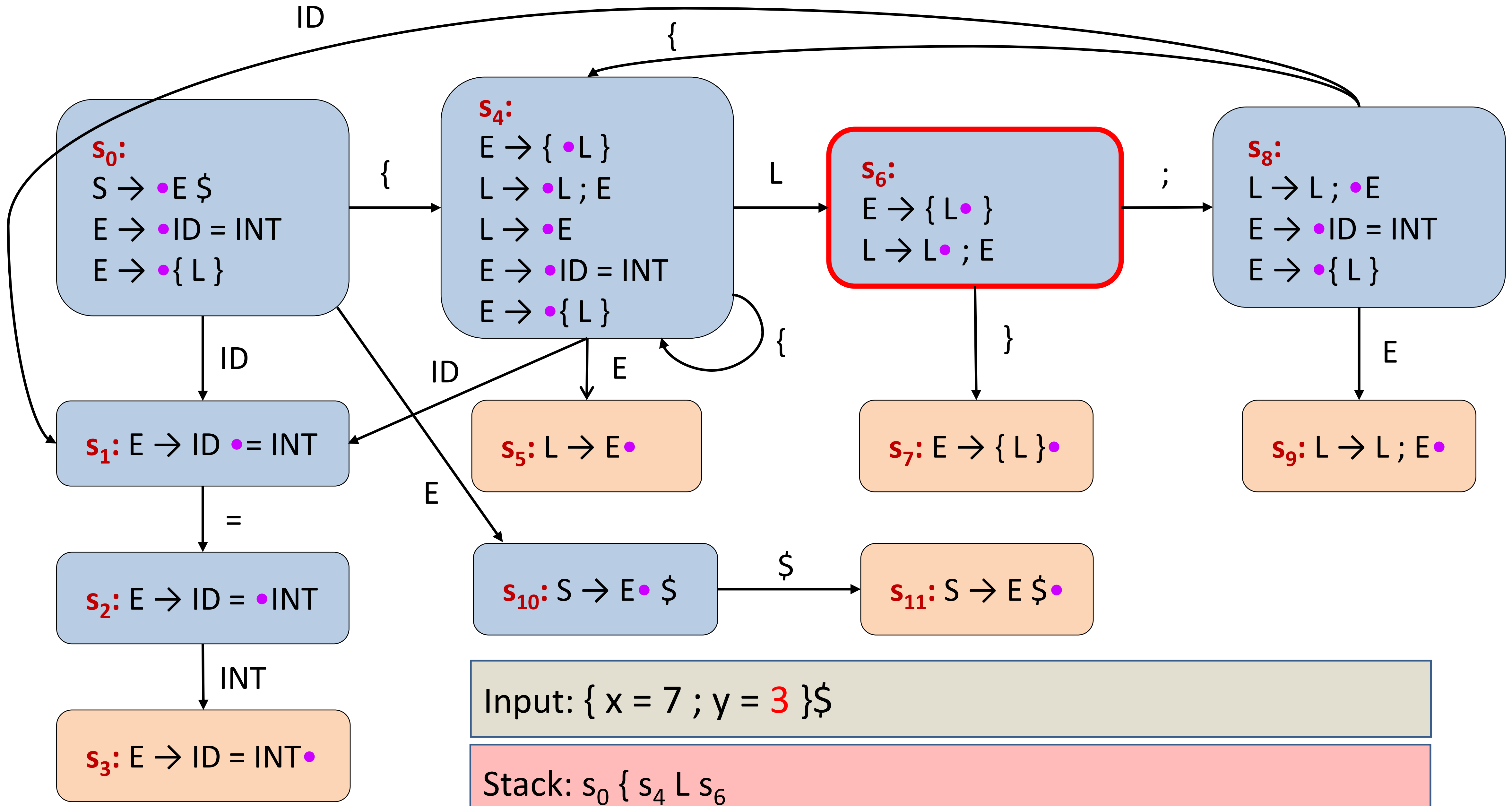
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 ; s_8 ID s_1 = s_2 INT s_3



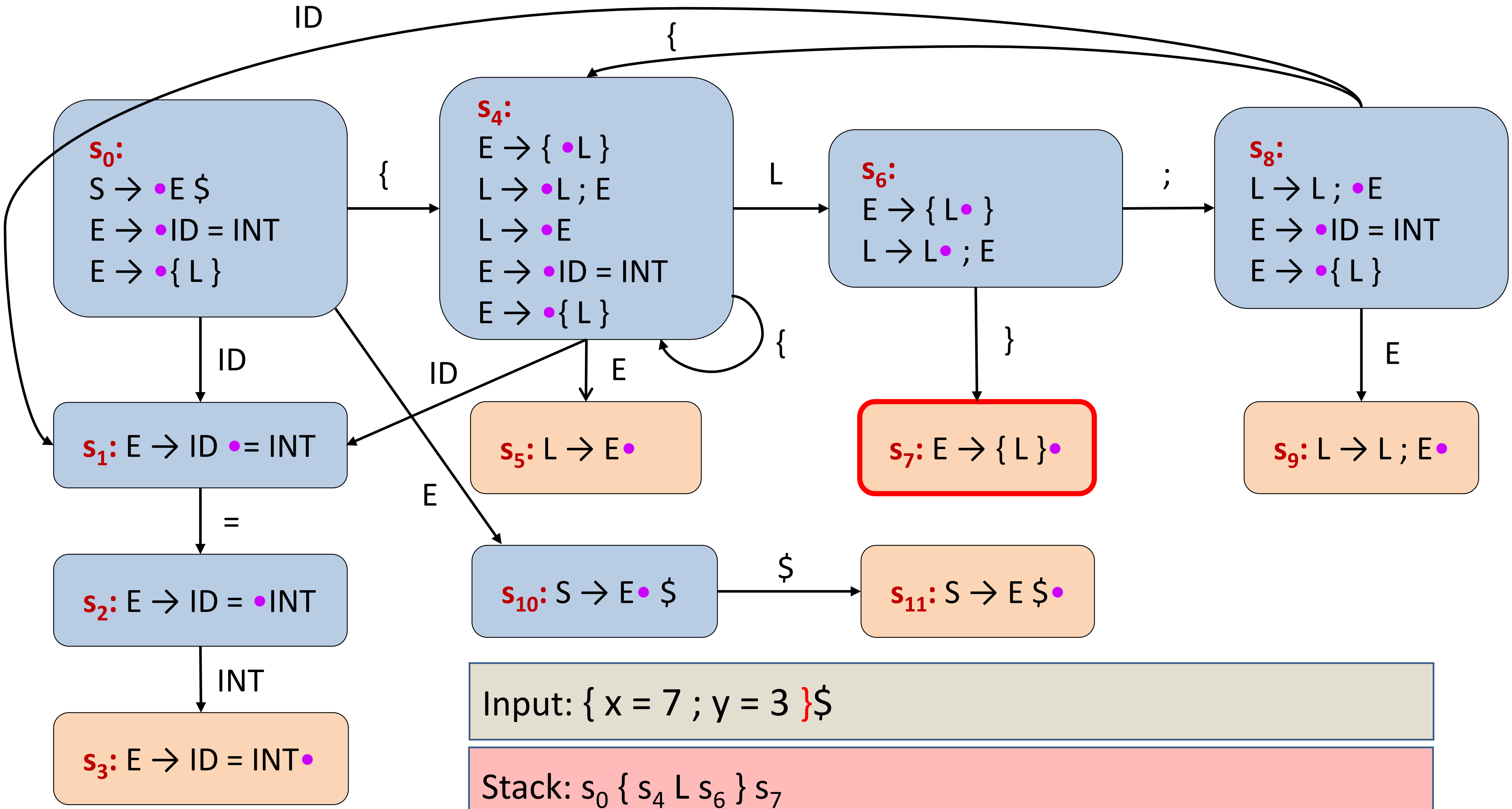
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 ; s_8 E s_9



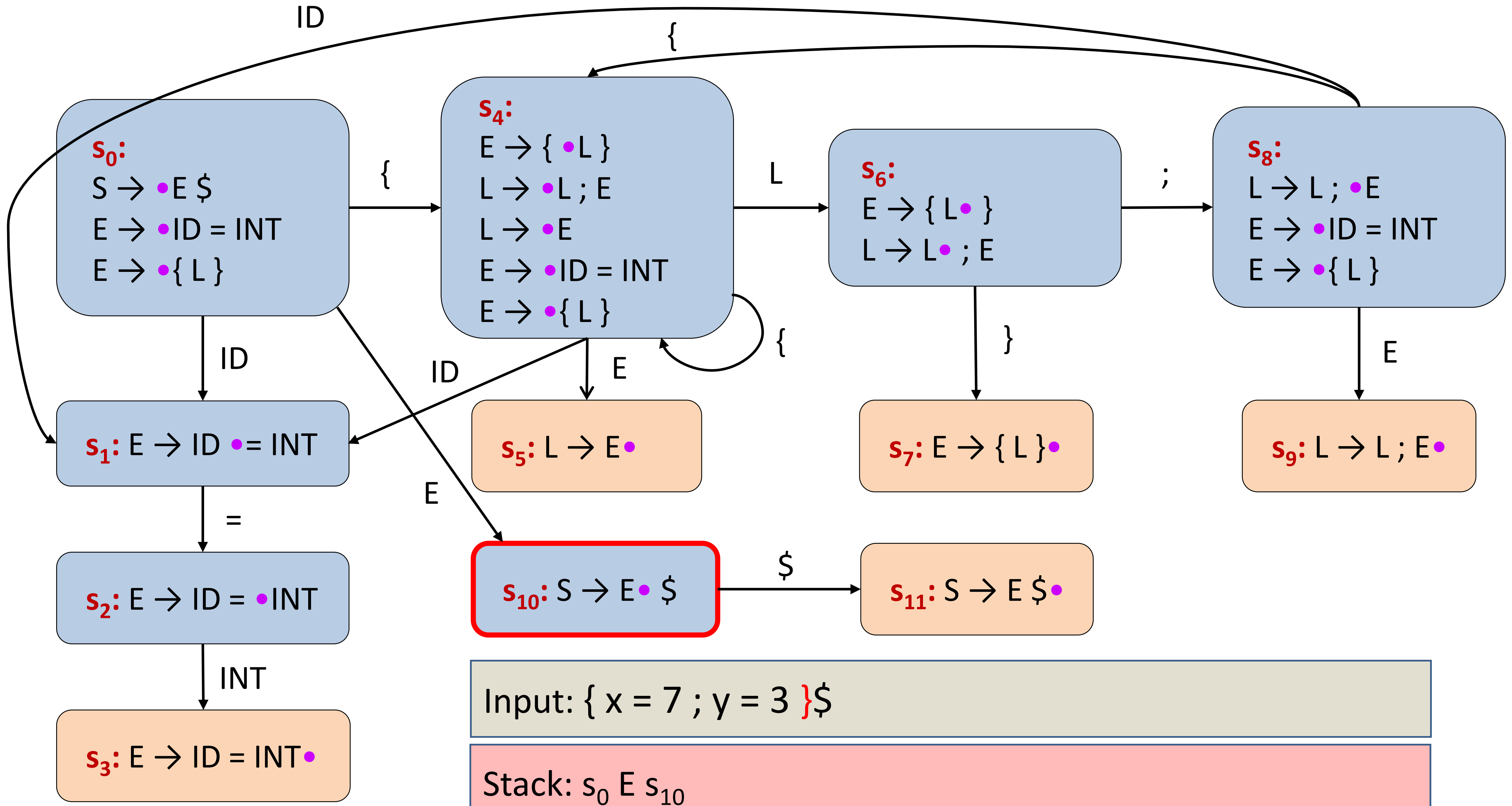
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6



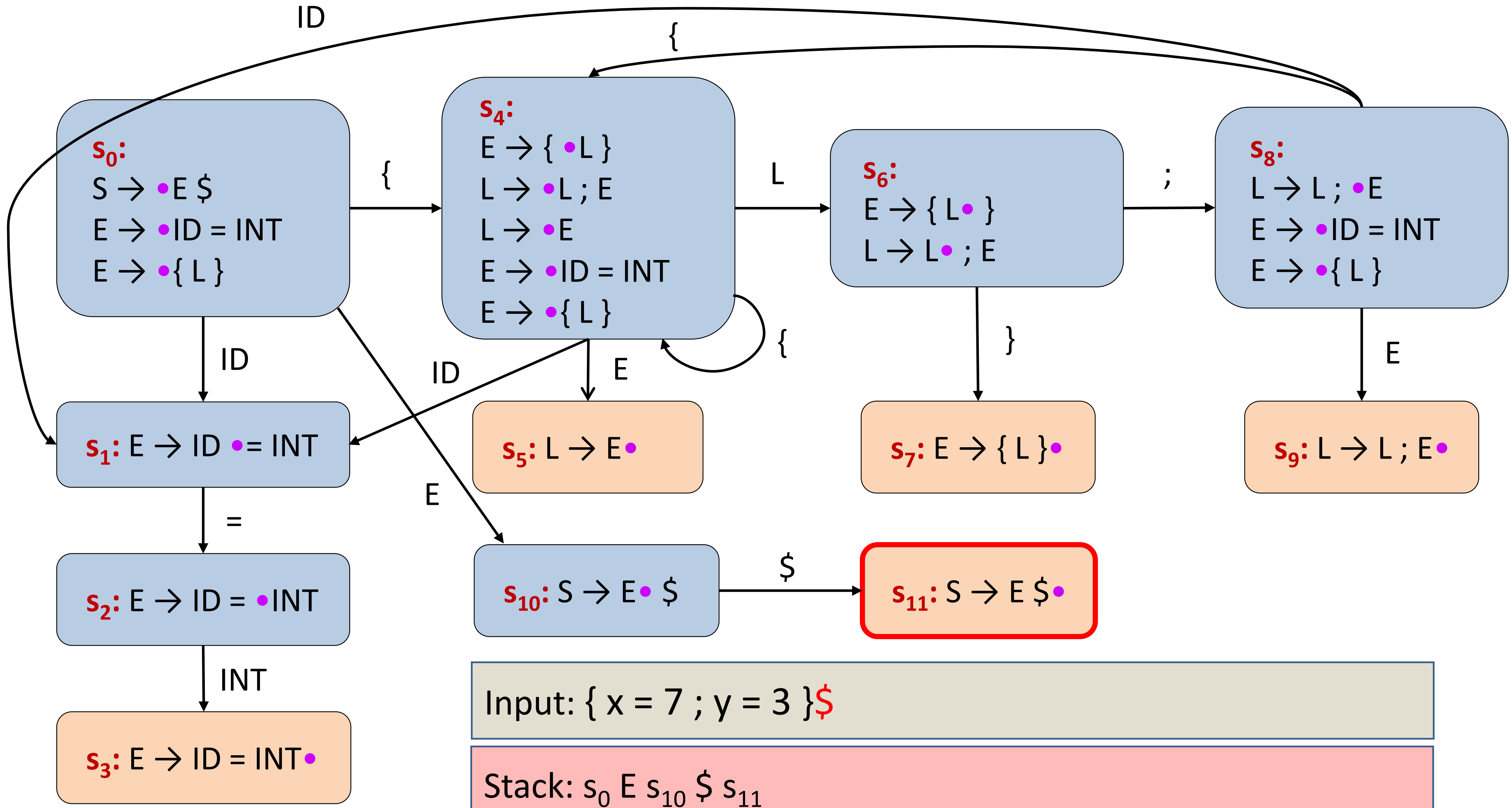
Input: { x = 7 ; y = 3 }$

Stack: s_0 { s_4 L s_6 } s_7



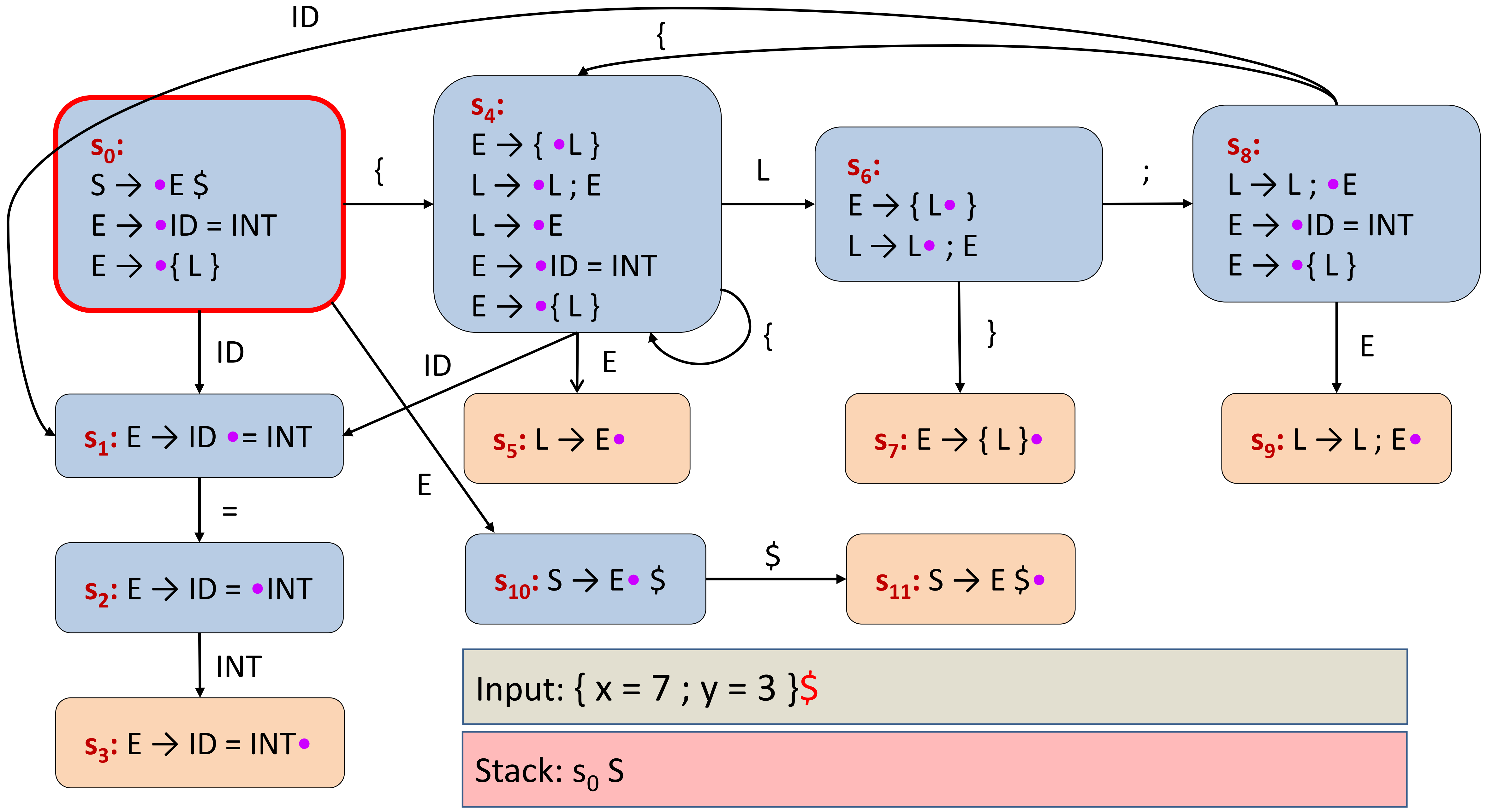
Input: { x = 7 ; y = 3 }$

Stack: s_0 E s_{10}



Input: { x = 7 ; y = 3 }$

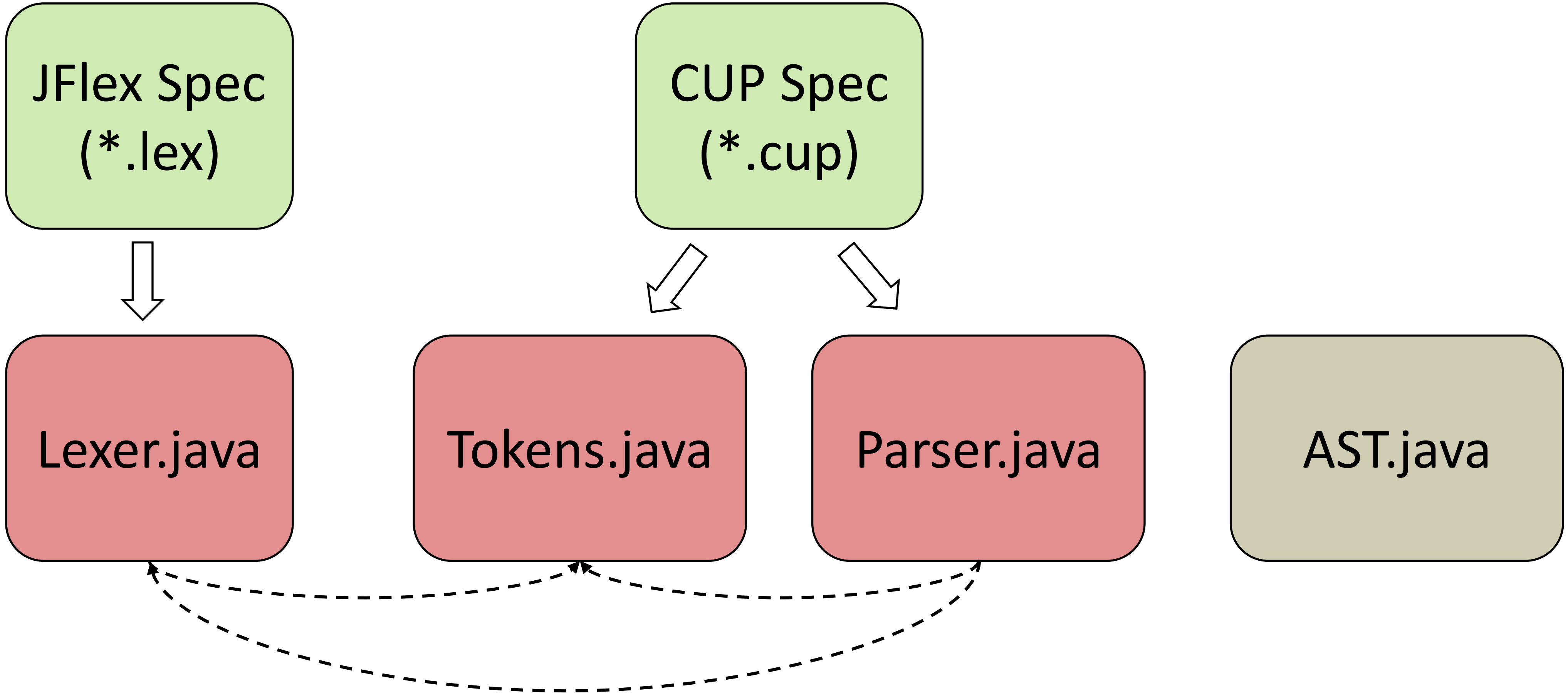
Stack: s_0 E s_{10} \$ s_{11}


ID
{
s0:
S → •E $
E → •ID = INT
E → •{ L }
{
s4:
E → { •L }
L → •L ; E
L → •E
E → •ID = INT
E → •{ L }
L
s6:
E → { L• }
L → L• ; E
;
s8:
L → L ; •E
E → •ID = INT
E → •{ L }
ID
ID
E
{
}
E
s1: E → ID •= INT
s5: L → E•
s7: E → { L }•
s9: L → L ; E•
=
E
s2: E → ID = •INT
s10: S → E• $
$
s11: S → E $•
INT
s3: E → ID = INT•
Input: { x = 7 ; y = 3 }$
Stack: s0 S

Parsing with CUP

- Given a user-specified grammar, generates an LALR parser
- Works with JFlex, which provides the parsed tokens
- Other tools:
  - Bison (for C)

CUP/JFlex Workflow



CUP Format

parser setup

```
parser code {:
…
:}
```

lexer setup

```
scan with {:
…
:}
```

grammar

```
terminal …
non terminal …
start with …
<derivation rules…>
```

CUP Spec: Parser Setup

```
parser code {:
    public Lexer lexer;
    public Parser(Lexer lexer) {
        super(lexer);
        this.lexer = lexer;
    }
    public void report_error(String message, Object info) {
        System.exit(0);
    }
:}
```

If the parser detects a syntax error, it calls this

CUP Spec: Lexer Setup

```
scan with {:
        Symbol s;
        s = lexer.next_token();
        // print token…
        return s;
:};
```

CUP Spec: Terminals

terminal T1;

terminal T2;

terminal T3;

terminal T4;

…

The tokens enum
is generated
according to the
declared terminals

CUP Spec: Non-Terminals

non terminal AST_NODE_1 E1;

non terminal AST_NODE_2 E2;

non terminal AST_NODE_3 E3;

…

object class name

CUP Spec: Operator Precedence

```
precedence right OP1;
precedence right OP2;
precedence left OP3;
precedence left OP4;
…
```

- These are token names

CUP Spec: Operator Precedence

precedence right OP1;

precedence right OP2;

precedence left OP3;

precedence left OP4;

…

associativity

Increasing precedence

- More on precedence and associativity next week

CUP Spec: Grammar

```
start with exp;
exp ::= INT                        {:  /* action */ :}
        | var
        | exp PLUS exp
        | exp MINUS exp
        ;
var ::= ID
        | var DOT ID
        ;
```

$E \rightarrow INT$
$E \rightarrow V$
$E \rightarrow E + E$
$E \rightarrow E - E$
$V \rightarrow ID$
$V \rightarrow V.ID$

CUP Spec: AST Nodes

- We need to **decide** which node types we have in our AST
- We need to **define** the classes for these AST nodes

CUP Example: Terminals

```
terminal Integer INT;
terminal String ID;
terminal PLUS;
terminal MINUS;
terminal DOT;
```

$E \rightarrow INT$
$E \rightarrow V$
$E \rightarrow E + E$
$E \rightarrow E - E$
$V \rightarrow ID$
$V \rightarrow V.ID$

CUP Example: Non-Terminals

non terminal AstExp exp;

non terminal AstVar var;

$E \rightarrow INT$
$E \rightarrow V$
$E \rightarrow E + E$
$E \rightarrow E - E$
$V \rightarrow ID$
$V \rightarrow V.ID$

CUP Spec: Grammar

$E \rightarrow INT$
$E \rightarrow V$
$E \rightarrow E + E$
$E \rightarrow E - E$
$V \rightarrow ID$
$V \rightarrow V.ID$

```
start with exp;
exp ::= INT:i {: RESULT = new AstExpInt(i); :}
        | var:v {: RESULT = new AstExpVar(v); :}
        | exp:e1 PLUS exp:e2 {: RESULT = new AstExpBinop(e1, e2, 0); :}
        | exp:e1 MINUS exp:e2 {: RESULT = new AstExpBinop(e1, e2, 1); :}
        ;
var ::= ID:name {: RESULT = new AstVarSimple(name); :}
        | var:v DOT ID:fieldName {: RESULT = new AstVarField(v, fieldName); :}
        ;
```

CUP Example: AST Nodes

For the non-terminal **var**:

public abstract class **AstVar** extends AstNode {

}

CUP Example: AST Nodes

For the rule **var ::= ID:**name:

```
public class AstVarSimple extends AstVar {
    public String name;
    public AstVarSimple(String name)    {
        this.name = name;
    }
}
```

CUP Example: AST Nodes

For the rule **var ::= var:v DOT ID:fieldName**:

```
public class AstVarField extends AstVar {
        public AstVar var;
        public String fieldName;
        public AstVarField(AstVar var, String fieldName) {
                this.var = var;
                this.fieldName = fieldName;
        }
}
```

CUP Example: AST Nodes

For the non-terminal **exp**:

```
public abstract class AstExp extends AstNode {

}
```

CUP Example: AST Nodes

For the rule **exp ::= int:i**:

```
public class AstExpInt extends AstExp {
	public int value;
	public AstExpInt(Integer value)  {
		this.value = value;
	}
}
```

CUP Example: AST Nodes

For the rule **exp ::= var:v**:

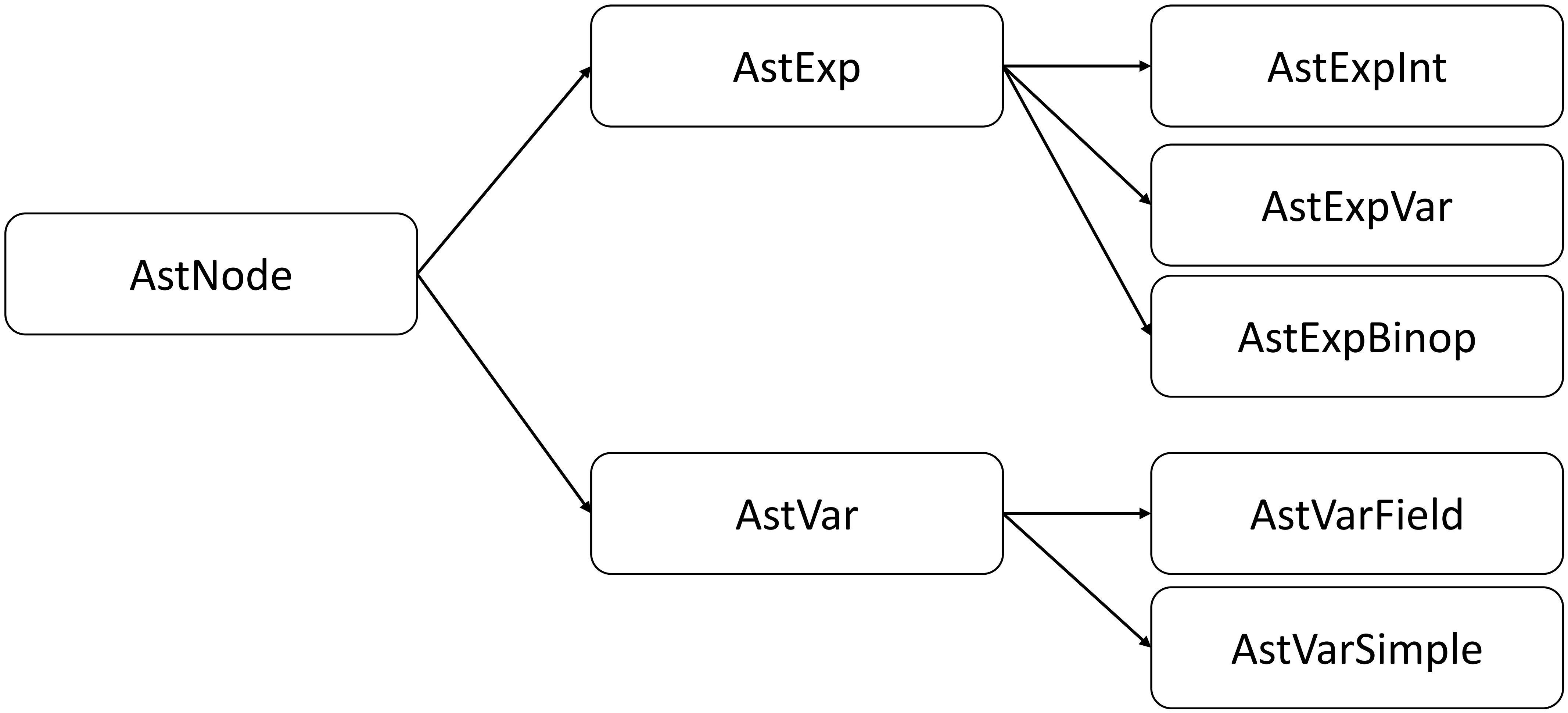
```
public class AstExpVar extends AstExp {
    public AstVar var;
    public AstExpVar(AstVar var) {
        this.var = var;
    }
}
```

CUP Example: AST Nodes

For the rule **exp ::= exp:e1 <OP> exp:e2**:

```
public class AstExpBinop extends AstExp {
    int op;
    public AstExp left;
    public AstExp right;
    public AstExpBinop(AstExp left, AstExp right, int op) {
        this.left = left;
        this.right = right;
        this.op = op;
    }
}
```

Class Hierarchy (Inheritance)

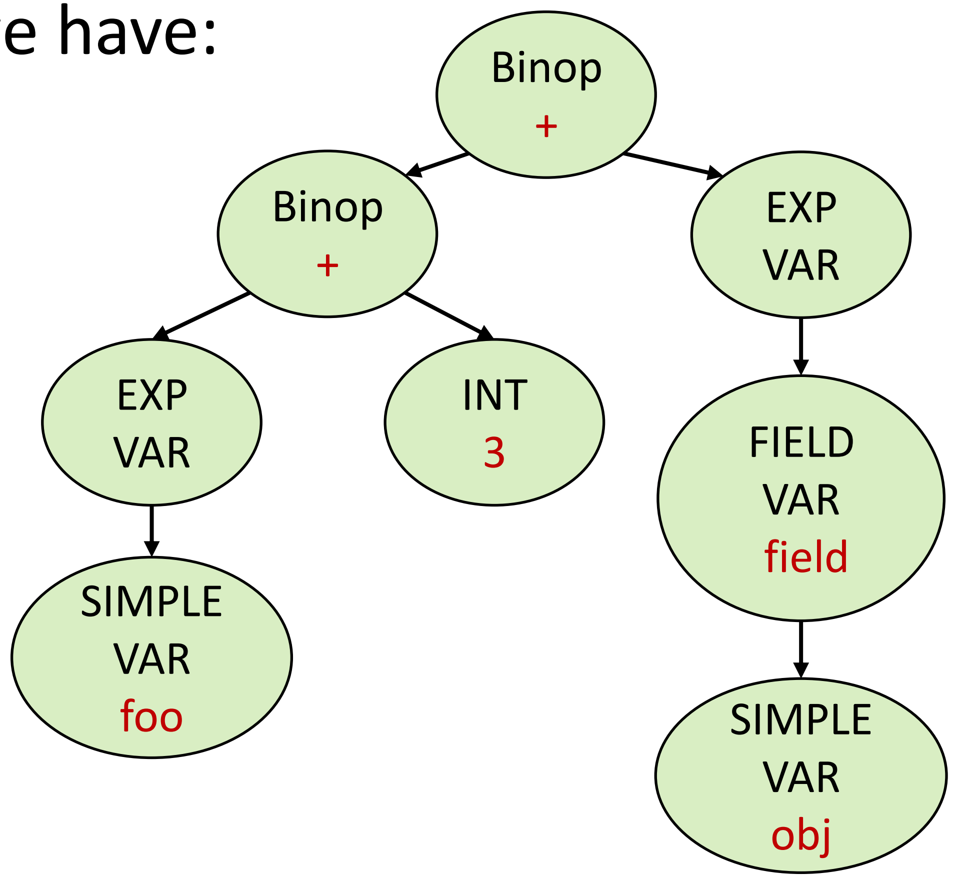


Accessing Line Numbers

```
expr ::= expr:e1 PLUS expr:e2
        {:
                int line = e1left;
                System.out.println("Rule matched at line: " + line);
        :}
```

CUP Example: Debugging

- We can generate an image of the AST (using the exercise template)
- For the input **foo + 3 + obj.field** we have:



BNF Notation

- Similar to CFG but adds regex-like operations
- Rules of the form:

  <nonterminal> ::= RHS

[x]	Optional zero or one occurrences of x
{ x }	Repetition Zero or more occurrences of x
(x | y)	choice One of either x or y