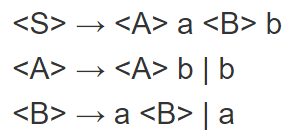
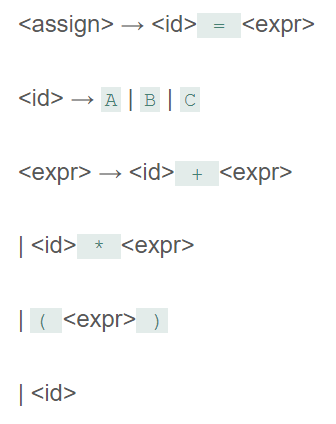
1. (10) Given the grammar below, identify which sentences are in the language (which are valid sentence).
   1. baab -> valid
   2. bbbab -> invalid (must have a sequence of two a’s)
   3. bbaaaaaa -> invalid (must end with b)
   4. bbaab -> valid



1. (10) Identify all of the tokens (categories of lexemes) in the grammar below, and which lexemes they categorize. Put them in a table.

|  |  |
| --- | --- |
| **Tokens** | **Lexemes** |
| Variable | A, B, C |
| Operator | =, +, \* |
| Paren | ), ( |



1. (10) Given the grammar from question 2, show a left-most derivation and draw the parse tree for the following statement.
   1. B = B + (C + (A \* A))

<assign> -> <id> = <expr>

-> B = <expr>

A close up of a map

Description automatically generated -> B = <id> + <expr>

-> B = B + <expr>

-> B = B + (<expr>)

-> B = B + (<id> + <expr>)

-> B = B + (C + <expr>)

-> B = B + (C + (<expr>))

-> B = B + (C + (<id> \* <expr>))

-> B = B + (C + (A \* <expr>))

-> B = B + (C + (A \* <id>))

-> B = B + (C + (A \* A))

1. (10) Remove all of the recursion from the following grammar:

S -> Aa | Bb

A -> Aa | AbC | C

B -> S | bb

C -> c

Solution:

S -> Aa | Bb

A -> CA’

A’ -> aA’ | bCA’ | ε

B -> AaB’ | bbB’

B’ -> bB’ | ε

C -> c

1. (10) Use left factoring to resolve the pairwise disjointness problems in the following grammar:

A -> aBc | ac | a

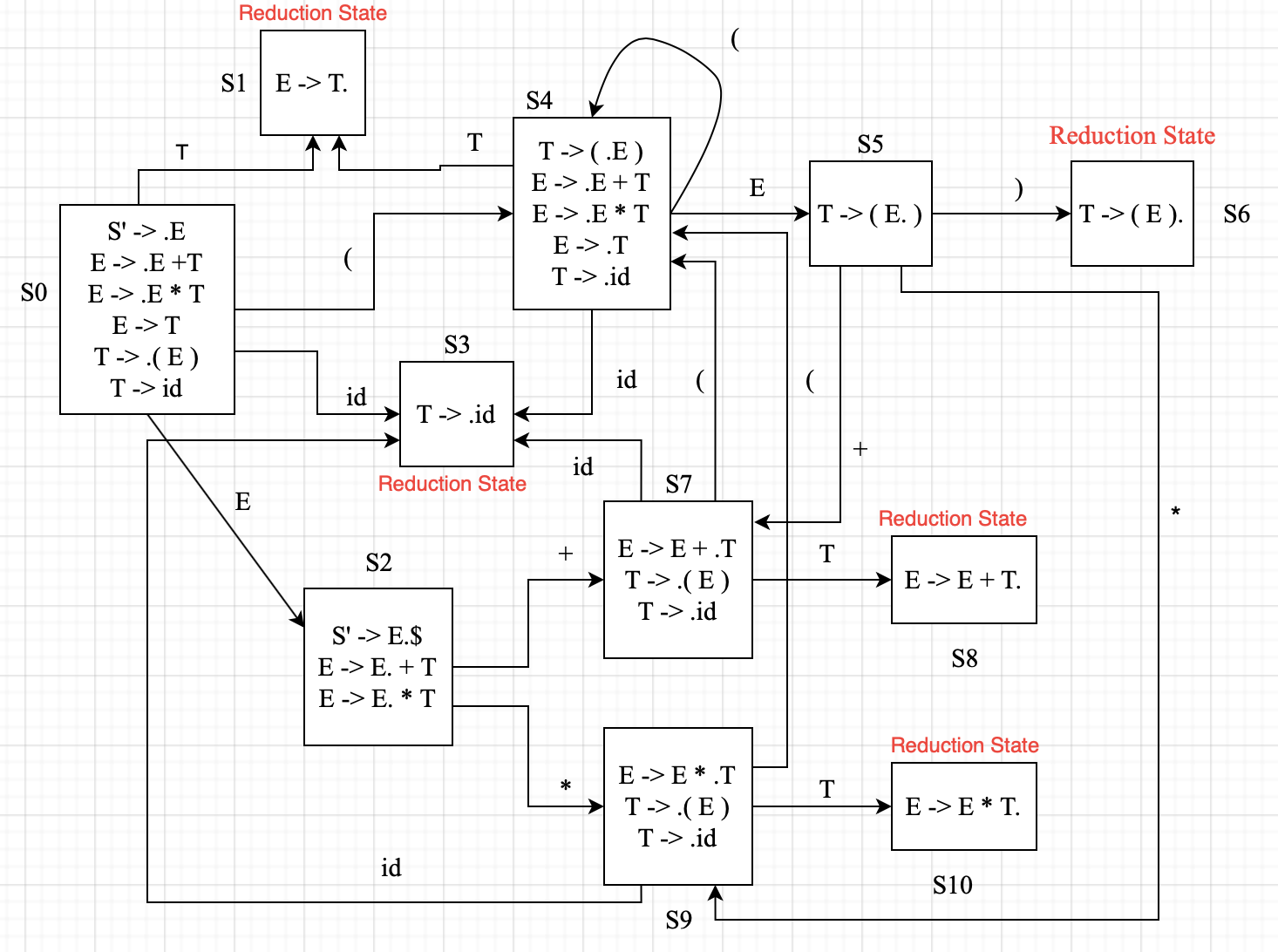
B -> b | aB

Solution:

A -> aA’

A’ -> Bc | c | ε

B -> b | aB

1.  (20 pts) Create an LR(0) parse table for the following grammar. Show all steps (creating closures, the DFA, the transition table, and finally the parse table):

E -> E + T | E \* T | T

T -> ( E ) | id

Rewritten:

r0: S’ -> E$

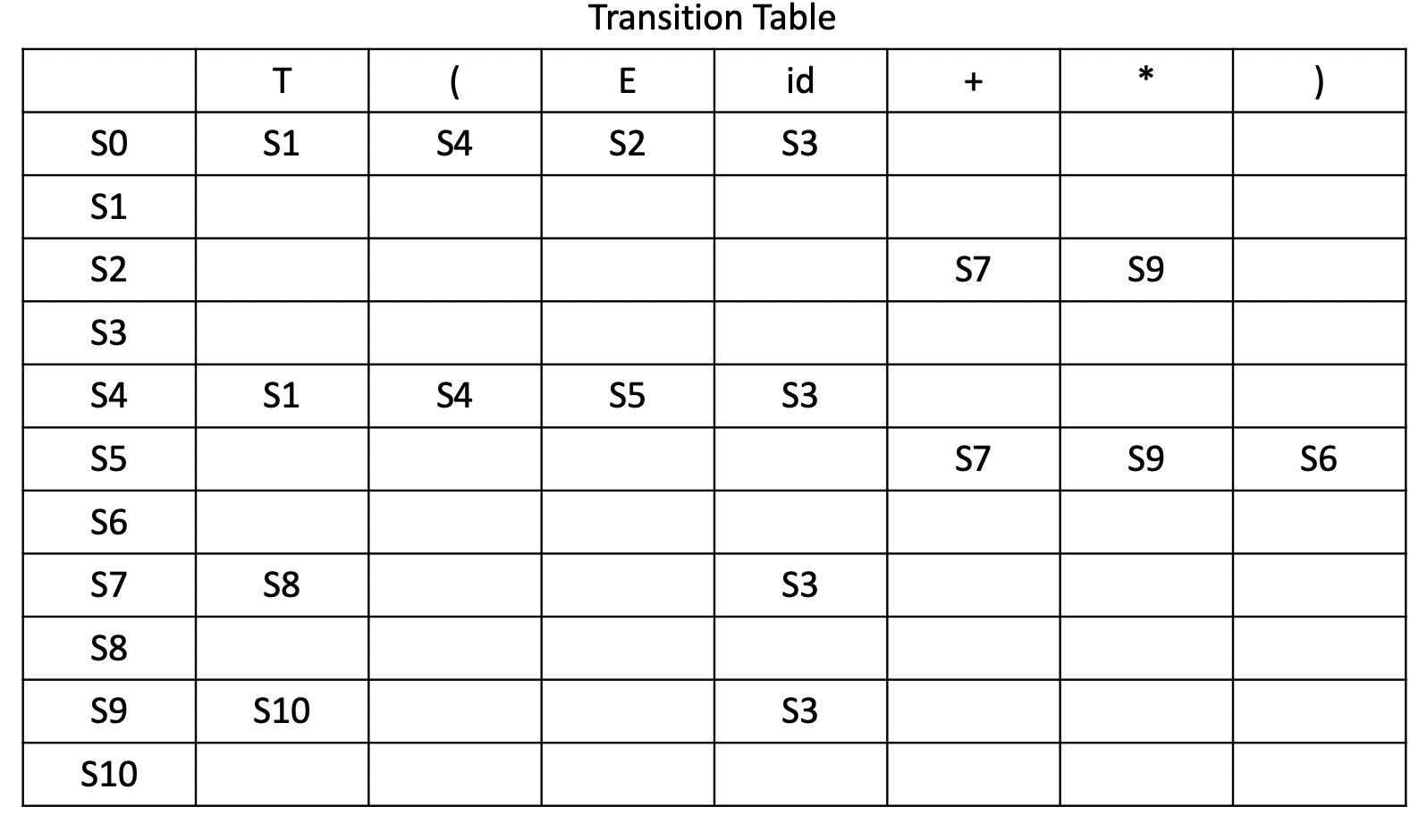
r1: E -> E + T

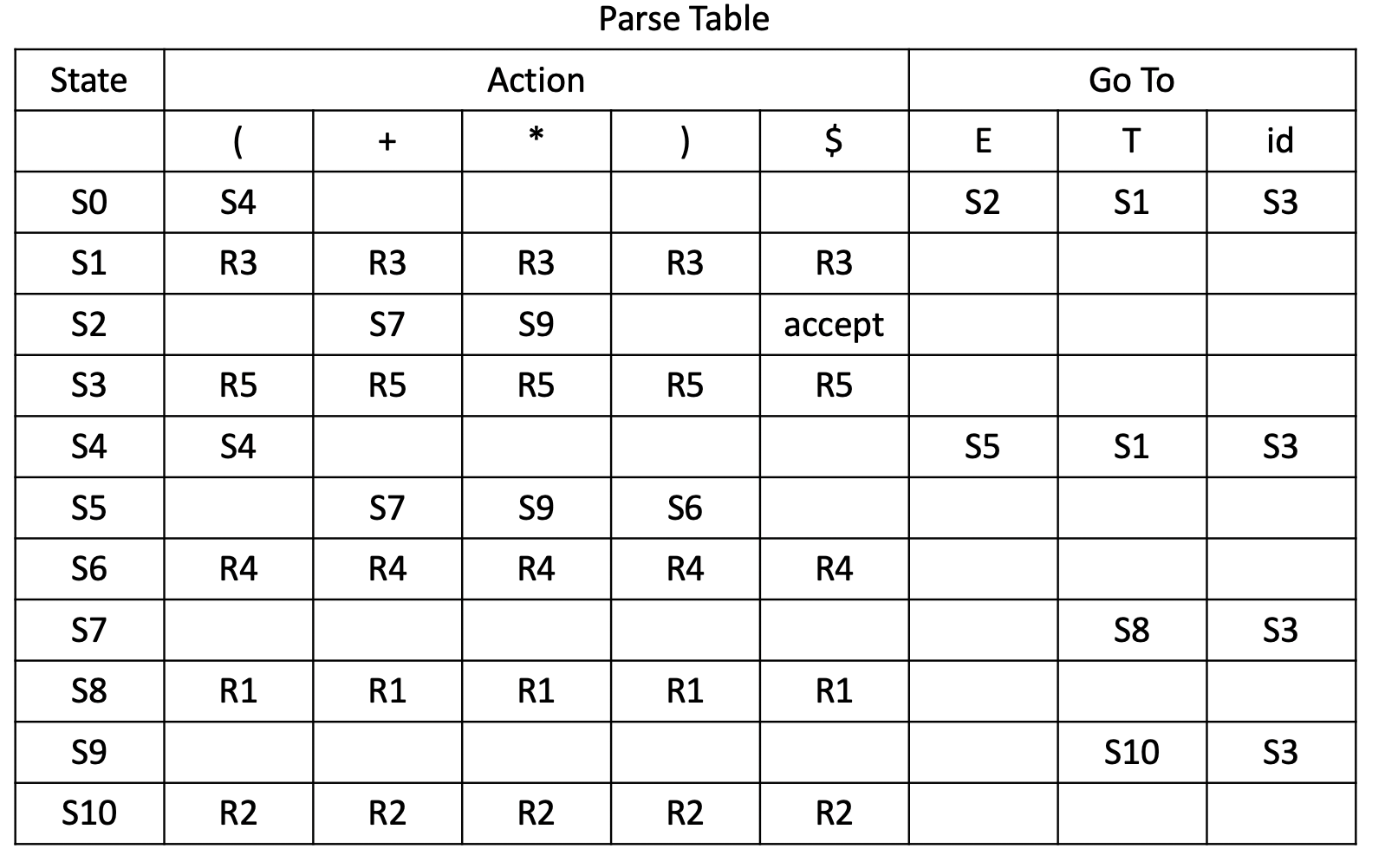
r2: E -> E \* T

r3: E -> T

r4: T -> ( E )

r5: T -> id





1. (20 pts) Show a complete bottom-up parse, including the parse stack contents, input string, and action for the string below using the parse table you created in step 6. Think about how I went through this in class.

(id + id) \* id

Step 1.

Input: .(id + id) \* id

Stack: 0

Output:

Step 2.

Input: (.id + id) \* id

Stack: 0 ( 4

Output:

Step 3:

Input: ( id. + id) \* id

Stack: 0 ( 4 id 3

Output:

Step 4:

Input: ( id. + id) \* id

Stack: 0 ( 4 T 1

Output: R5

Step 5:

Input: ( id. + id) \* id

Stack: 0 ( 4 E

Output: R5, R3

Step 6:

Input: ( id. + id) \* id

Stack: 0 ( 4 E 5

Output: R5, R3

Step 7:

Input: ( id +. id) \* id

Stack : 0 ( 4 E 5 + 7

Output: R5, R3

Step 8:

Input: ( id + id.) \* id

Stack : 0 ( 4 E 5 + 7 id 3

Output: R5, R3

Step 9:

Input: ( id + id.) \* id

Stack : 0 ( 4 E 5 + 7 T

Output: R5, R3, R5

Step 10:

Input: ( id + id.) \* id

Stack : 0 ( 4 E

Output: R5, R3, R5, R1

Step 11:

Input : ( id + id.) \* id

Stack : 0 ( 4 E 5

Output: R5, R3, R5, R1

Step 12:

Input: ( id + id). \* id

Stack : 0 ( 4 E 5 ) 6

Output: R5, R3, R5, R1

Step 13:

Input: ( id + id). \* id

Stack : 0 T 1

Output: R5, R3, R5, R1, R4

Step 14:

Input : ( id + id). \* id

Stack: 0 E 2

Output: R5, R3, R5, R1, R4, R3

Step 15:

Input: ( id + id) \*. Id

Stack: 0 E 2 \* 9

Output: R5, R3, R5, R1, R4, R3

Step 16:

Input: ( id + id) \* id.

Stack: 0 E 2 \* 9 id 3

Output : R5, R3, R5, R1, R4, R3

Step 17:

Input: ( id + id) \* id.

Stack: 0 E 2 \* 9 T

Output : R5, R3, R5, R1, R4, R3, R5

Step 18:

Input: ( id + id) \* id.

Stack: 0 E 2 \* 9 T 10

Output : R5, R3, R5, R1, R4, R3, R5

Step 19:

Input: ( id + id) \* id.

Stack: 0 E

Output : R5, R3, R5, R1, R4, R3, R5, R2

1. (10 pts) Show a rightmost derivation for the string above and show how the bottom-up parse you completed in step 7 correctly finds all of the handles for the input string above.

r0: S’ -> E$

r1: E -> E + T

r2: E -> E \* T

r3: E -> T

r4: T -> ( E )

r5: T -> id

(id + id) \* id Output: R5, R3, R5, R1, R4, R3, R5, R2

1. E

2. E \* T (2)

3. E \* id (5)

4. T \* id (3)

5. ( E ) \* id (4)

6. ( E + T) \* id (1)

7. ( E + id) \* id (5)

8. ( T + id) \* id (3)

9. ( id + id) \* id (5)