Scheme Recursion Processing

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Main()
begin
1. while (true)
2.     Command := GetCommand()
3.     InitializeTokenizer(command)
4.     root := Read()
5.     result := Eval(root)
6.     PrintResult(result, true)
end
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Preprocessing(newcommand)
begin
1. // newcommand is an empty string when this procedure is first called
2. while (token := GetNextToken()) is not empty
3.     if token is "define"
4.         newcommand := Concatenate(newcommand, "define")
5.         token := GetNextToken()
6.         if token is "("
               // (define (square x) ( * x x )) ==>
               // (define square (lambda (x) ( * x x )))
7.             token := GetNextToken()
8.            newcommand := Concatenate(newcommand, token,
9.                                   "(lambda(", Preprocessing(newcommand), ")" )
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Preprocessing()
…
10.  elseif token is “’”
         // ’(a b c) ==> (quote (a b c))
11.      newcommand := Concatenate(newcommand, “(quote”)
12.      number_of_left_paren := 0
13.      do
14.         token := GetNextToken()
15.         newcommand := Concatenate(newcommand, token)
16.         if token is “(”
17.            number_of_left_paren := number_of_left_paren+1
18.         elseif token is “)”
19.            number_of_left_paren := number_of_left_paren−1
20.      while (number_of_left_paren>0)
21.      newcommand := Concatenate(newcommand, “)” )
22.  else newcommand := Concatenate(newcommand, token)
23. return newcommand
end
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
begin
1. tokenindex := GetHashValue(Memory[root].lchild)
2. if (token index = PLUS)
3.     return GetHashValue(GetVal(Eval(Memory[Memory[root].rchild].lchild))
4.                  + GetVal(Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)))
…
11. elseif (token index = isEQ) // eq?
12.     return Eval(Memory[Memory[root].rchild].lchild
13.                  = Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
14. elseif (token index = isEQUAL) // equal?
15.     return CheckStructure(Eval(Memory[Memory[root].rchild].lchild),
16.                          Eval(Memory[Memory[Memory[root].rchild].rchild].lchild))
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

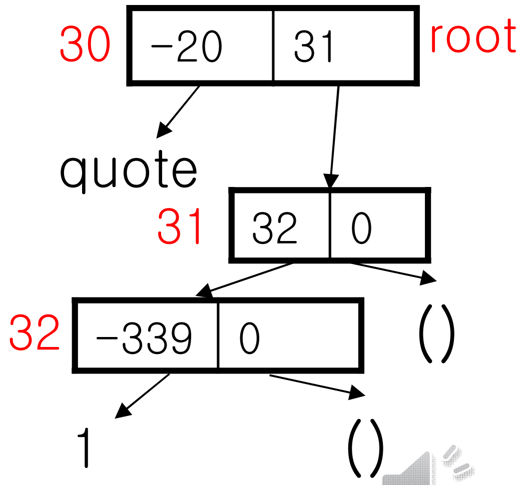
```
Procedure Eval(root)
…
17. elseif (token index = isNUMBER)
18.   if IsNumber(Eval(Memory[Memory[root].rchild].lchild)) is true
19.      return GetHashValue("#t")
20.   else return GetHashValue("#f")

21. elseif (token index = isSYMBOL)
22.   if result := EVAL(Memory[Memory[root].rchild].lchild) is true and IsNumber(result) is false
23.      return GetHashValue("#t")
24.   else return GetHashValue("#f")

25. elseif (token index = isNULL)
26.   if Memory[root].rchild is NIL or Eval(Memory[root].rchild) is NIL
27.      return GetHashValue("#t")
28.   else return GetHashValue("#f")
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
29. elseif (token index = CONS)
30.   newmemory := Alloc()
31.   Memory[newmemory].lchild := Eval(Memory[Memory[root].rchild].lchild)
32.   Memory[newmemory].rchild := Eval(Memory[Memory[Memory].root].rchild].rchild].lchild)
33.   return newmemory

34. elseif (token index = COND)
35.   while Memory[Memory[root].rchild].rchild is not NIL
36.      root := Memory[root].rchild
37.      if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
38.        return EVAL(Memory[Memory[root].lchild].rchild)
39.  if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
40.     Error()
41.  return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

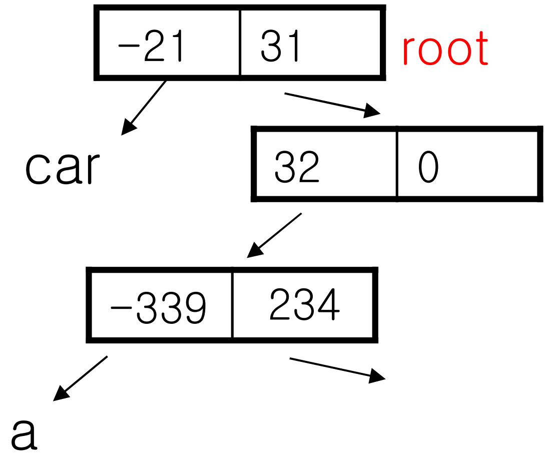
SCHEME INTERPRETER PSEUDO CODE (2nd Version)

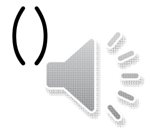
```
Procedure Eval(root)
…
42. elseif (token index = CAR)
43.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

44. elseif (token index = CDR)
45.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

46. elseif (token index = DEFINE)
47.   if function define
48.     hashTable[Memory[Memory[root].rchild].lchild].pointer :=
49.                 Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
50.   else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
51.                 EVAL(Memory[Memory[root].rchild].rchild)

52.   elseif (token index = QUOTE)
53.     return Memory[Memory[root].rchild].lchild
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

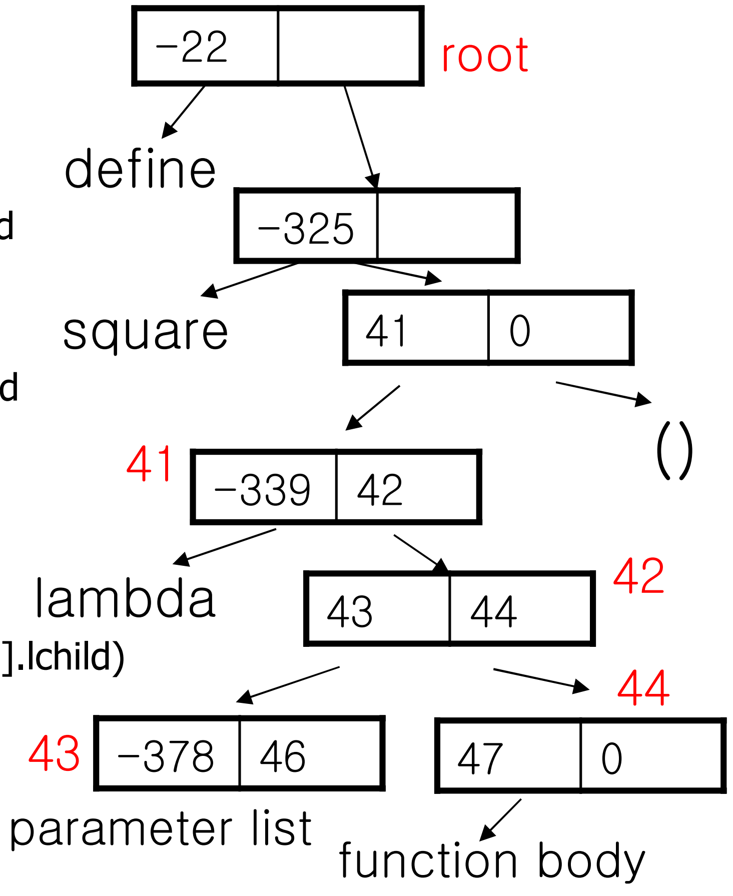
```
Procedure Eval(root)
…
42. elseif (token index = CAR)
43.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

44. elseif (token index = CDR)
45.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

46. elseif (token index = DEFINE)
47.   if function define
48.     hashTable[Memory[Memory[root].rchild].lchild].pointer :=
49.                 Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
50.   else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
51.                 EVAL(Memory[Memory[root].rchild].rchild)

52.   elseif (token index = QUOTE)
53.     return Memory[Memory[root].rchild].lchild
```





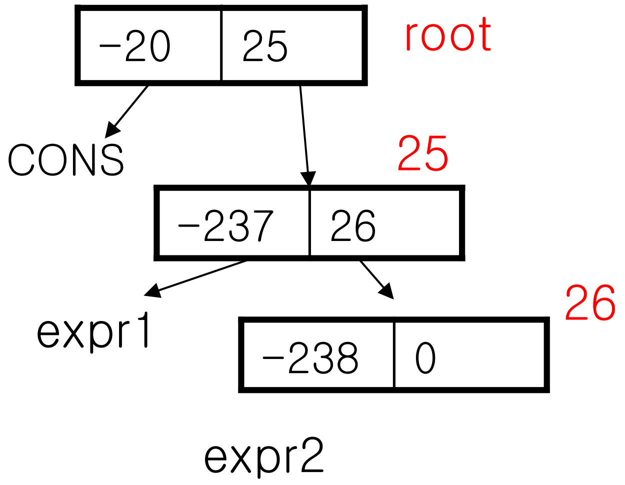
SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
42. elseif (token index = CAR)
43.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

44. elseif (token index = CDR)
45.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

46. elseif (token index = DEFINE)
47.   if function define
48.     hashTable[Memory[Memory[root].rchild].lchild].pointer :=
49.                  Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
50.   else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
51.                  EVAL(Memory[Memory[root].rchild].rchild)

52.   elseif (token index = QUOTE)
53.     return Memory[Memory[root].rchild].lchild
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
29. elseif (token index = CONS)
30.   newmemory := Alloc()
31.   Memory[newmemory].lchild := Eval(Memory[Memory[root].rchild].lchild)
32.   Memory[newmemory].rchild := Eval(Memory[Memory[Memory].root].rchild].rchild].lchild)
33.   return newmemory
```

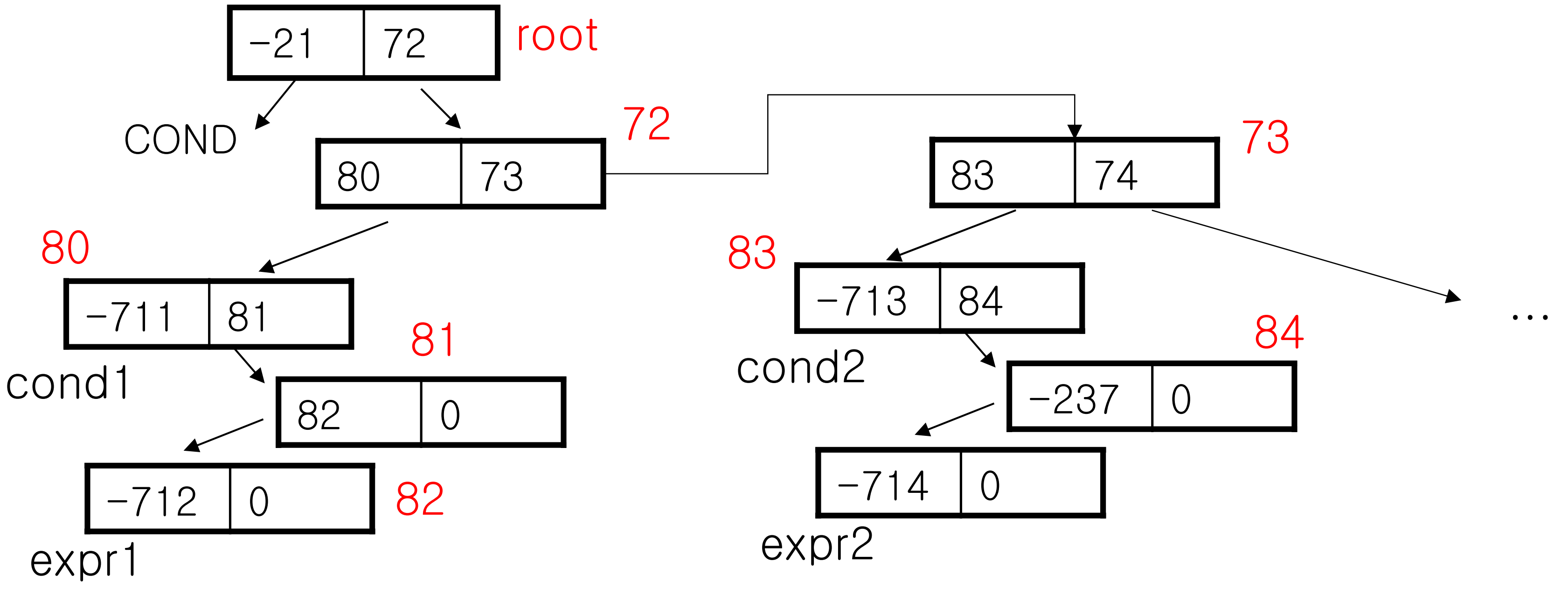


(CONS expr1 expr2)

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
34. elseif (token index = COND)
35.   while Memory[Memory[root].rchild].rchild is not NIL
36.     root := Memory[root].rchild
37.     if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
38.       return EVAL(Memory[Memory[root].lchild].rchild)
39.  if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
40.     Error()
41.  return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

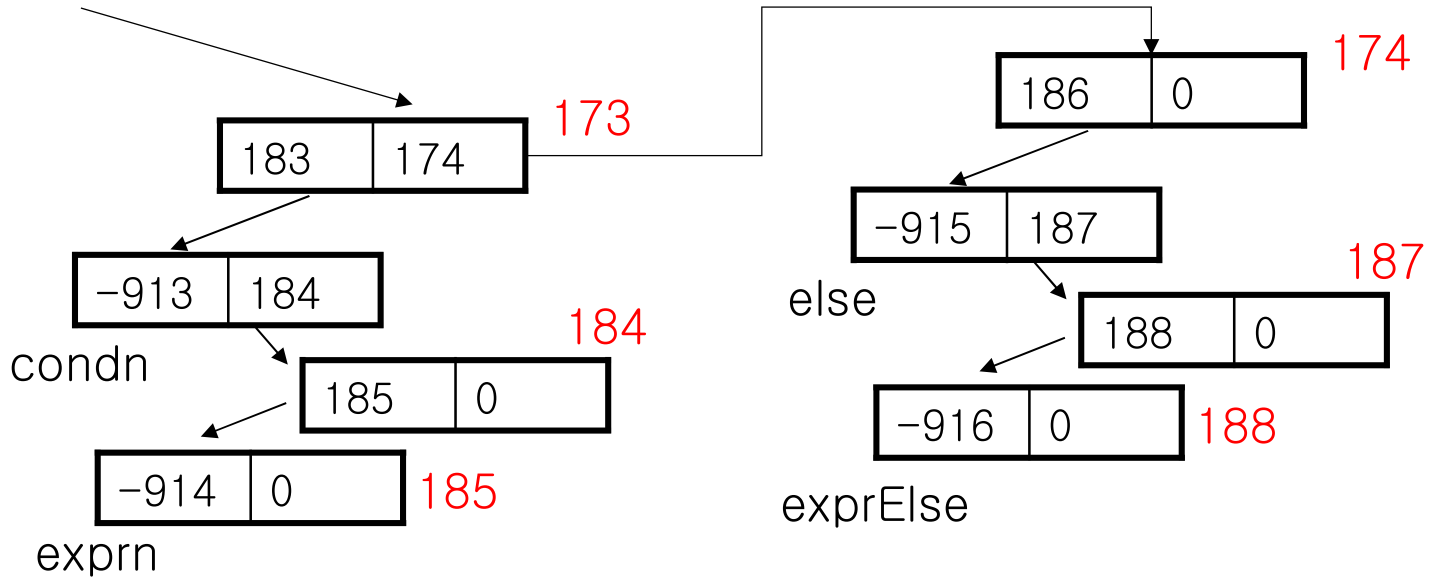
```
(COND ((cond1) (expr1))
      ((cond2) (expr2))
      ...
      ((condn) (exprn))
      ((else) (exprElse)))
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

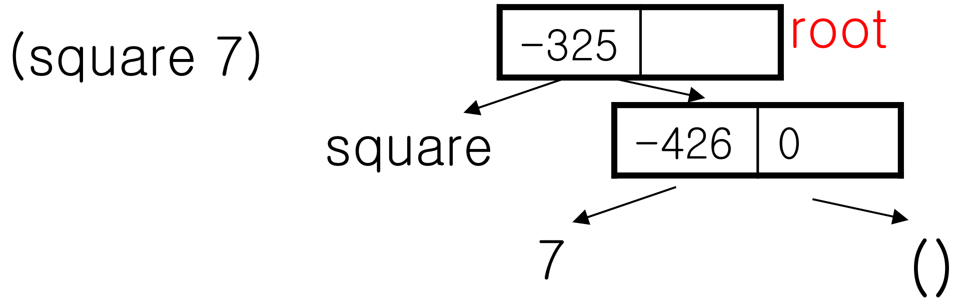
```
34. elseif (token index = COND)
35.   while Memory[Memory[root].rchild].rchild is not NIL
36.     root := Memory[root].rchild
37.     if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
38.       return EVAL(Memory[Memory[root].lchild].rchild)
39.  if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
40.     Error()
41.  return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

```
(COND ((cond1) (expr1))
      ((cond2) (expr2))
      ...
      ((condn) (exprn))
      ((else) (exprElse)))
```

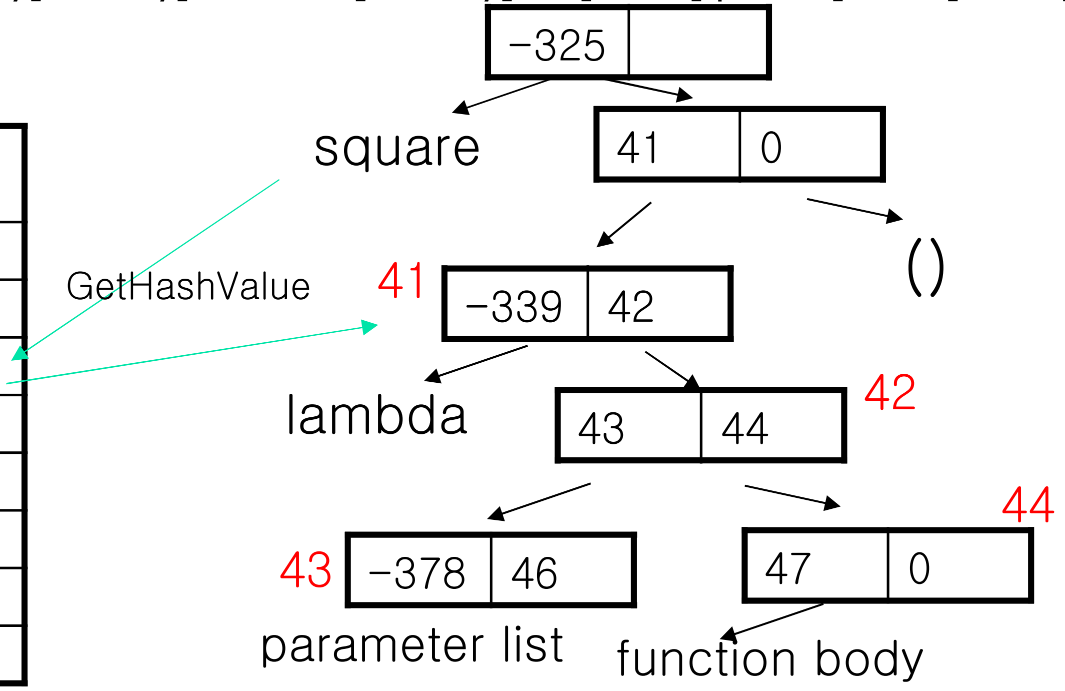


SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
54.  elseif token index is user defined function
55.      push current values to stack
56.      set parameter by function argument
57.      result := Eval(Memory[Memory[Memory[hashTable[Memory[root].lchild].pointer].rchild].rchild].lchild)
58.      pop the values from stack
59.  return result
```



Hash Value	Symbol	Link of Value
…		
…		
−325	square	41
…		
−426	7	0
…		
…		

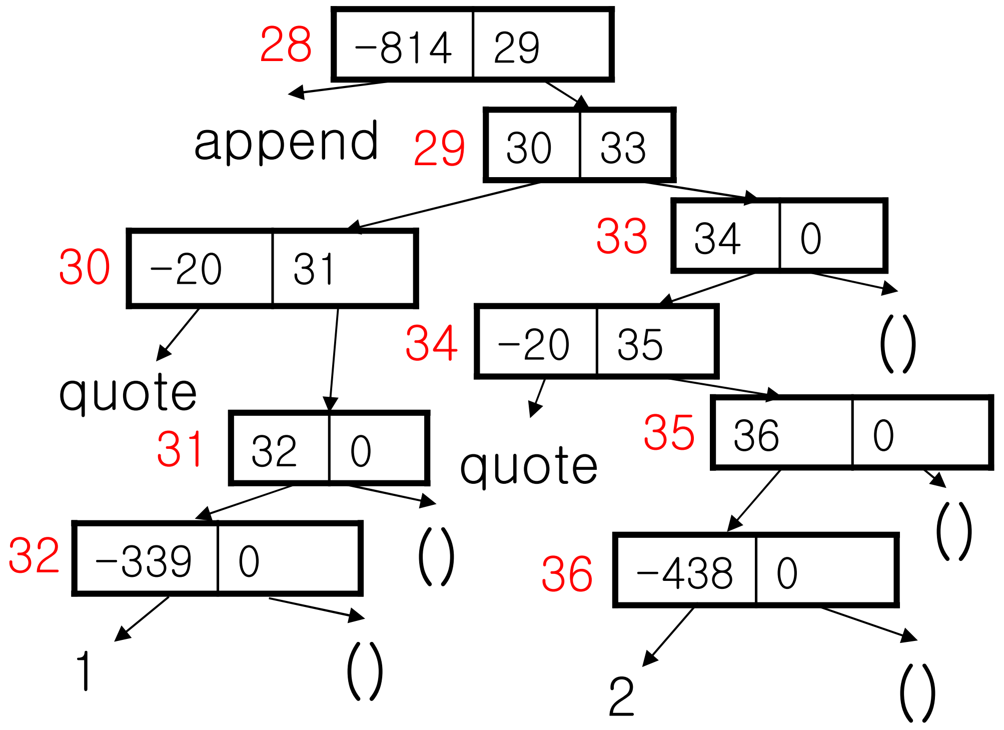


Append a list to another list

```
(define (append L R)
  (cond ((null? L) R)
    (else (cons (car L) (append (cdr L) R)))))
```

- (append '(1) '(2)) : (1 2)

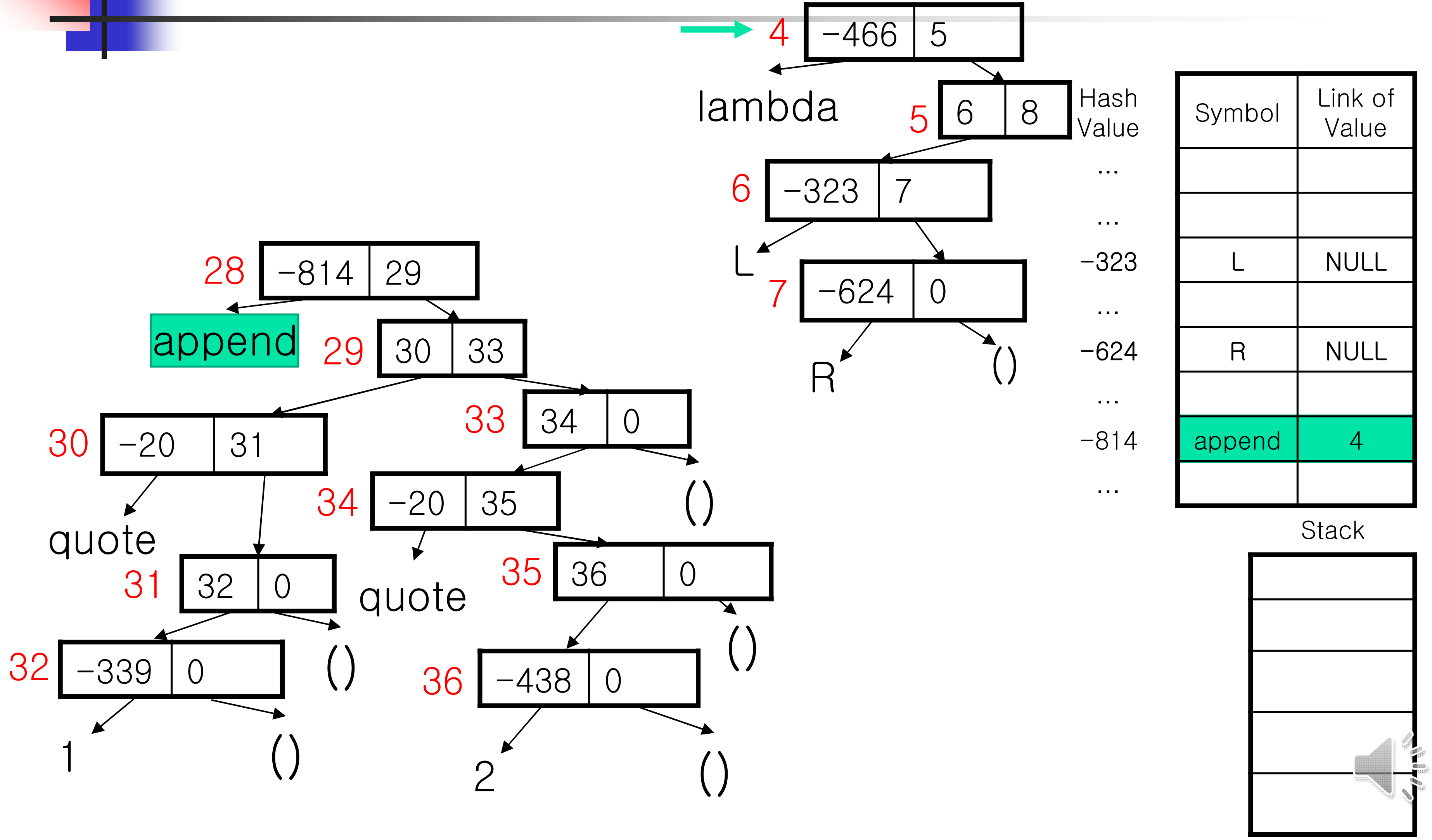
Evaluation of (append ‘(1) '(2))



Hash Value	Symbol	Link of Value
…		
…		
−323	L	NULL
…		
−624	R	NULL
…		
−814	append	4
…		

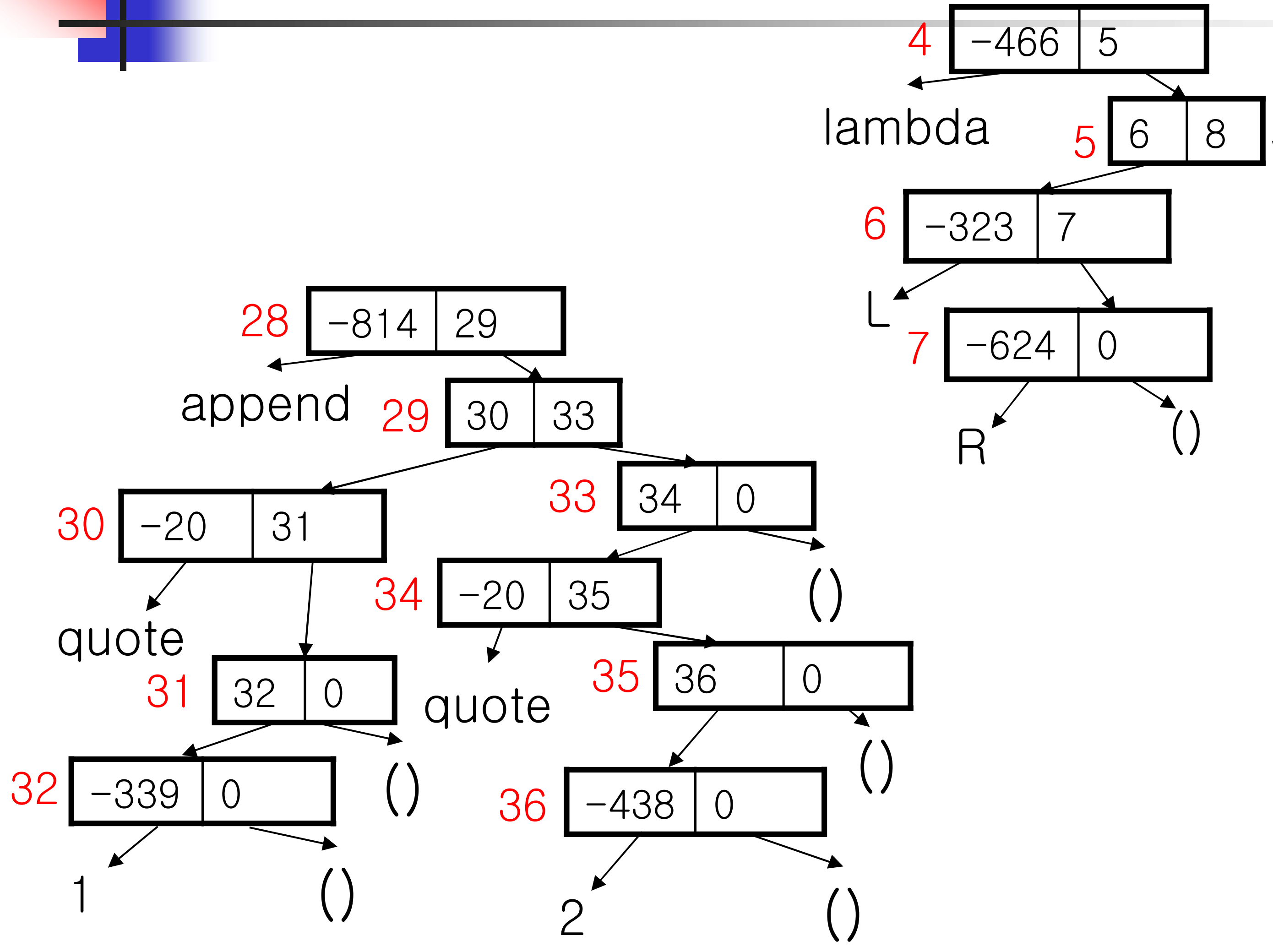
Stack

Evaluation of (append '(1) '(2))



Hash Value	Symbol	Link of Value
…		
…		
−323	L	NULL
…		
−624	R	NULL
…		
−814	append	4
…		

Evaluation of (append '(1) '(2))

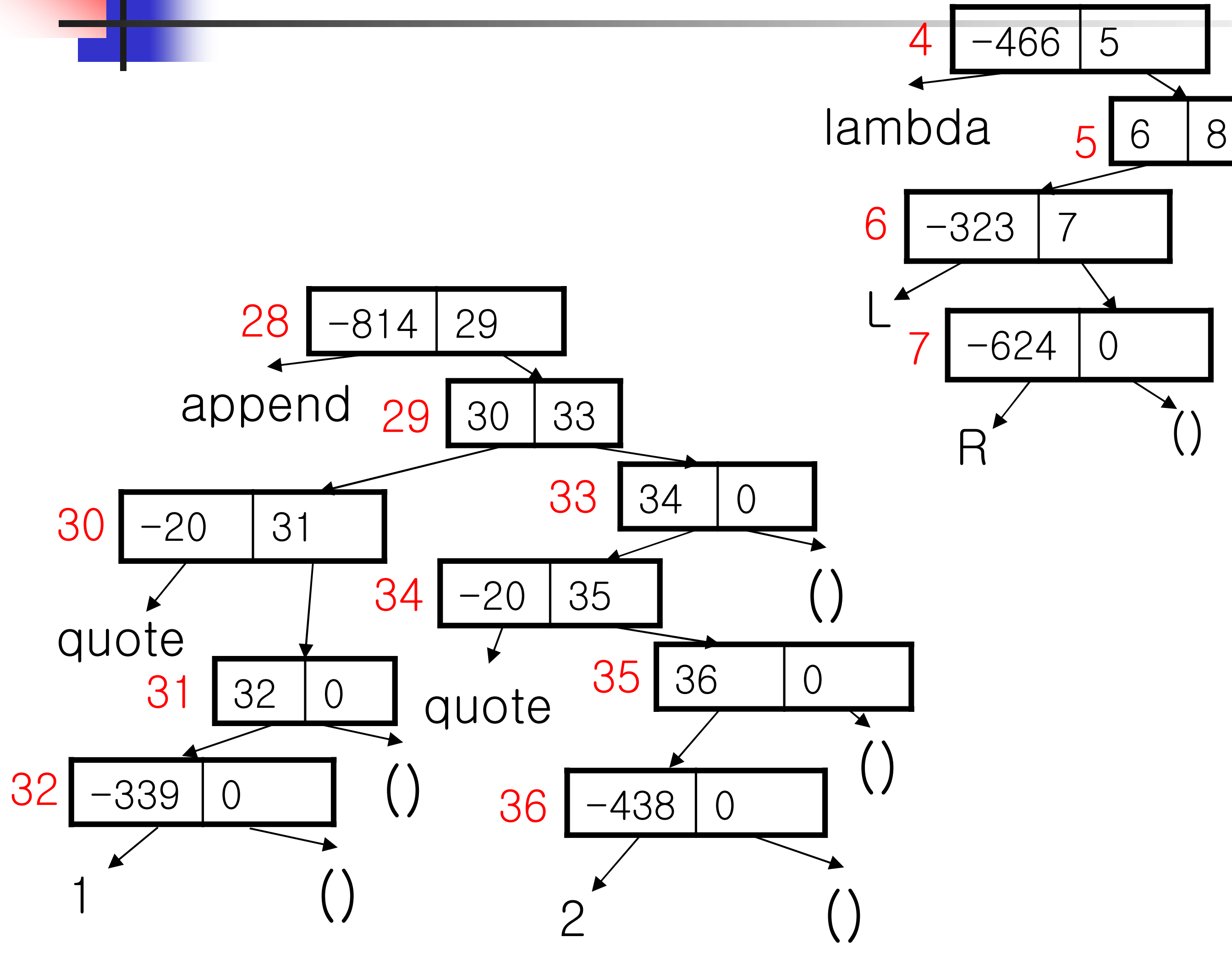


Hash Value	Symbol	Link of Value
…		
…		
-323	L	32
…		
-624	R	NULL
…		
-814	append	4
…		

Stack

L : NULL

Evaluation of (append '(1) '(2))



Hash Value	Symbol	Link of Value
…		
…		
−323	L	32
…		
−624	R	36
…		
−814	append	4
…		

Stack

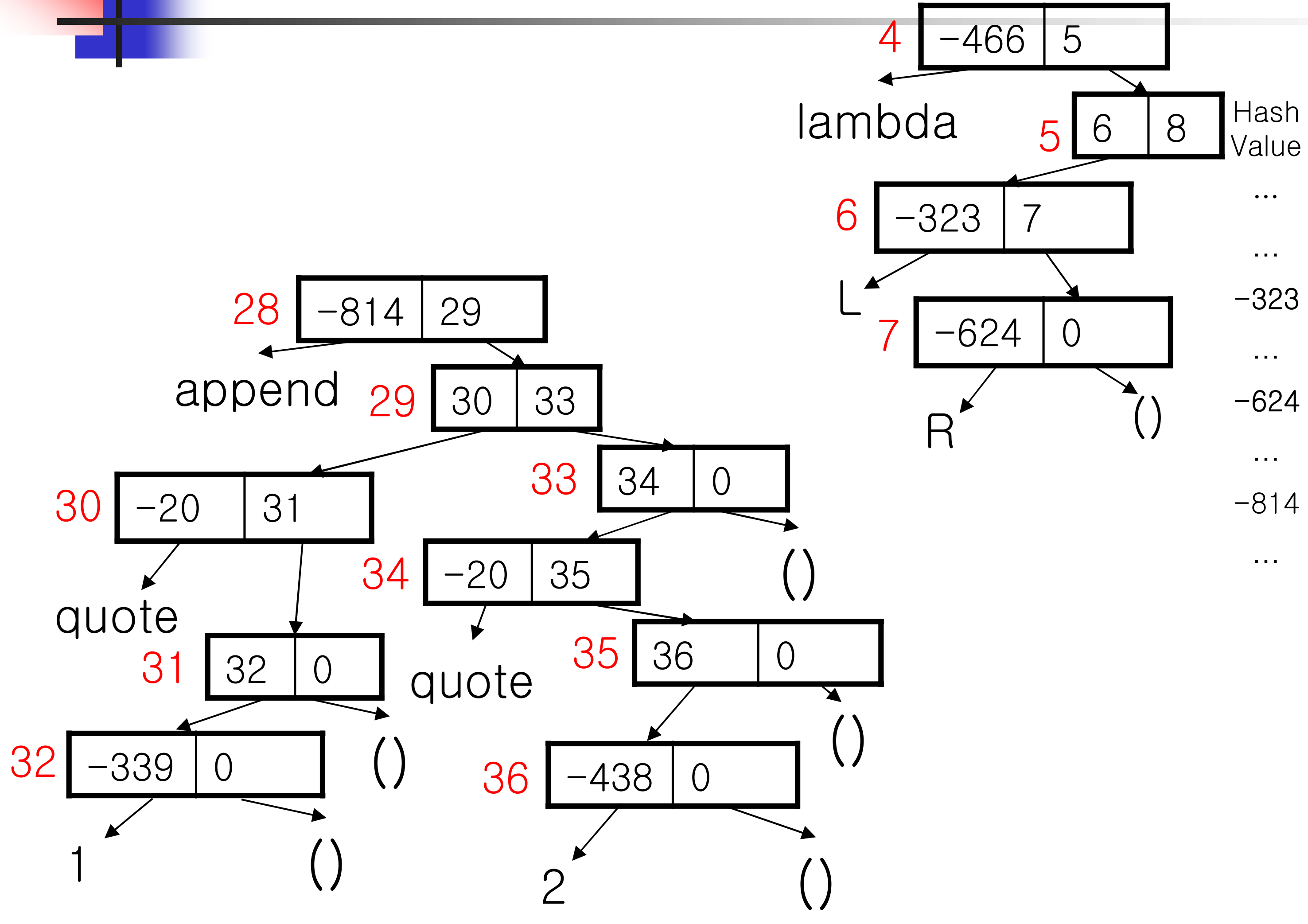
R : NULL
L : NULL

Append a list to another list

```
(define (append L R)
  (cond ((null? L) R)
    (else (cons (car L) (append (cdr L) R)))))
```

- (append '(1) '(2)) : (1 2)

Evaluation of (append '(1) '(2))

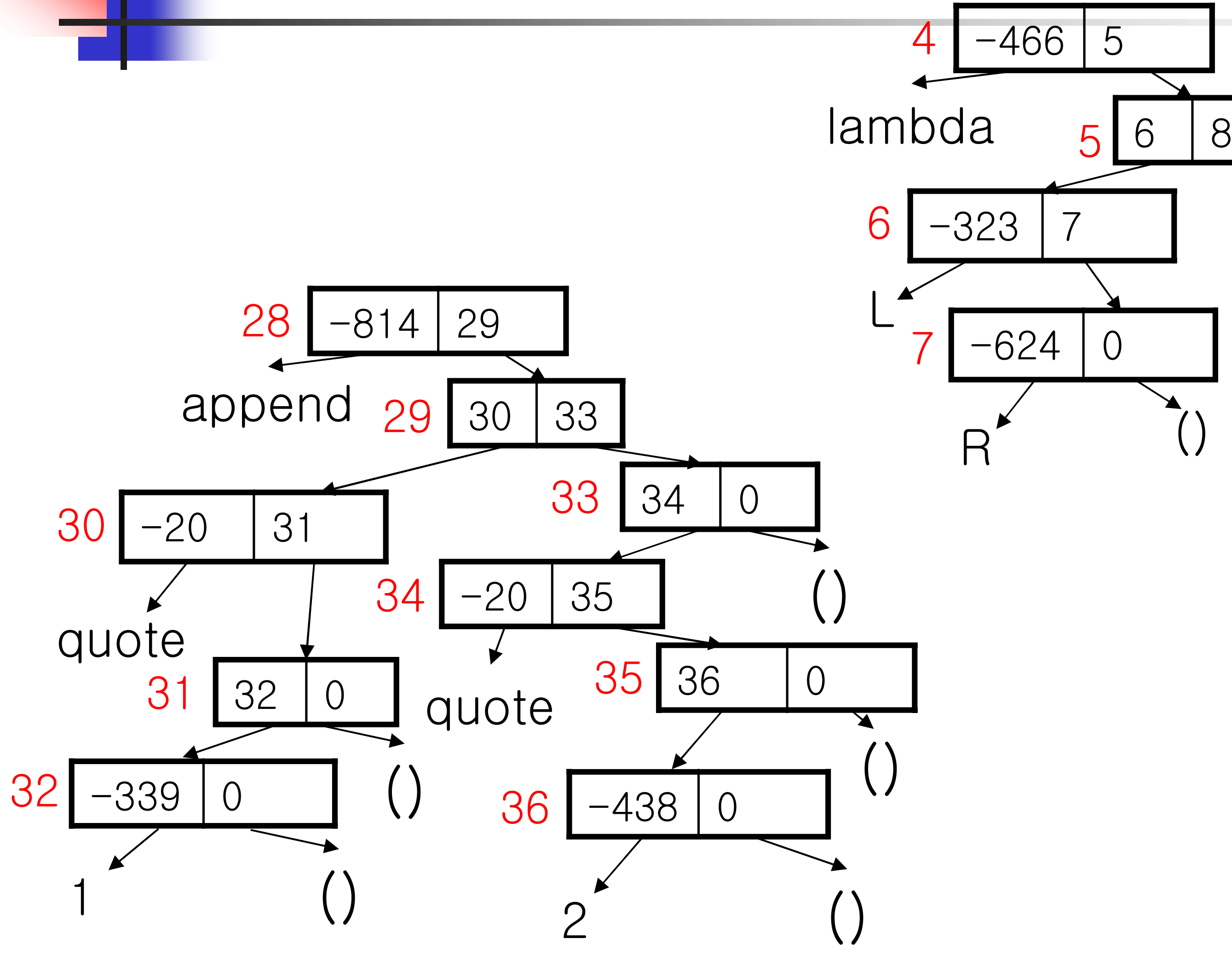


Hash Value	Symbol	Link of Value
...		
...		
-323	L	0
...		
-624	R	36
...		
-814	append	4
...		

Stack

L : 32
R : NULL
L : NULL

Evaluation of (append '(1) '(2))

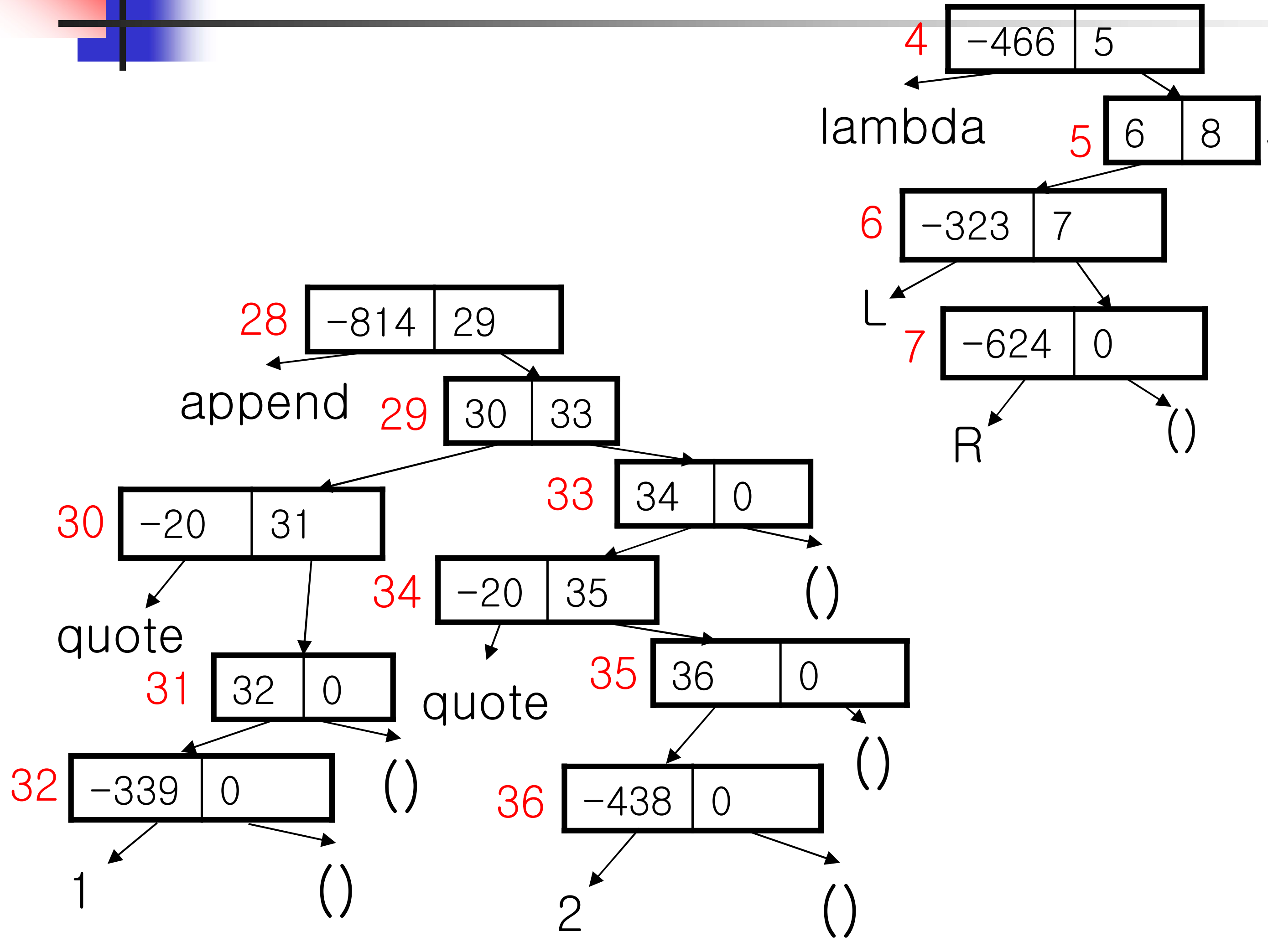


Hash Value	Symbol	Link of Value
…		
…		
−323	L	0
…		
−624	R	36
…		
−814	append	4
…		

Stack

R : 36
L : 32
R : NULL
L : NULL

Evaluation of (append '(1) '(2))

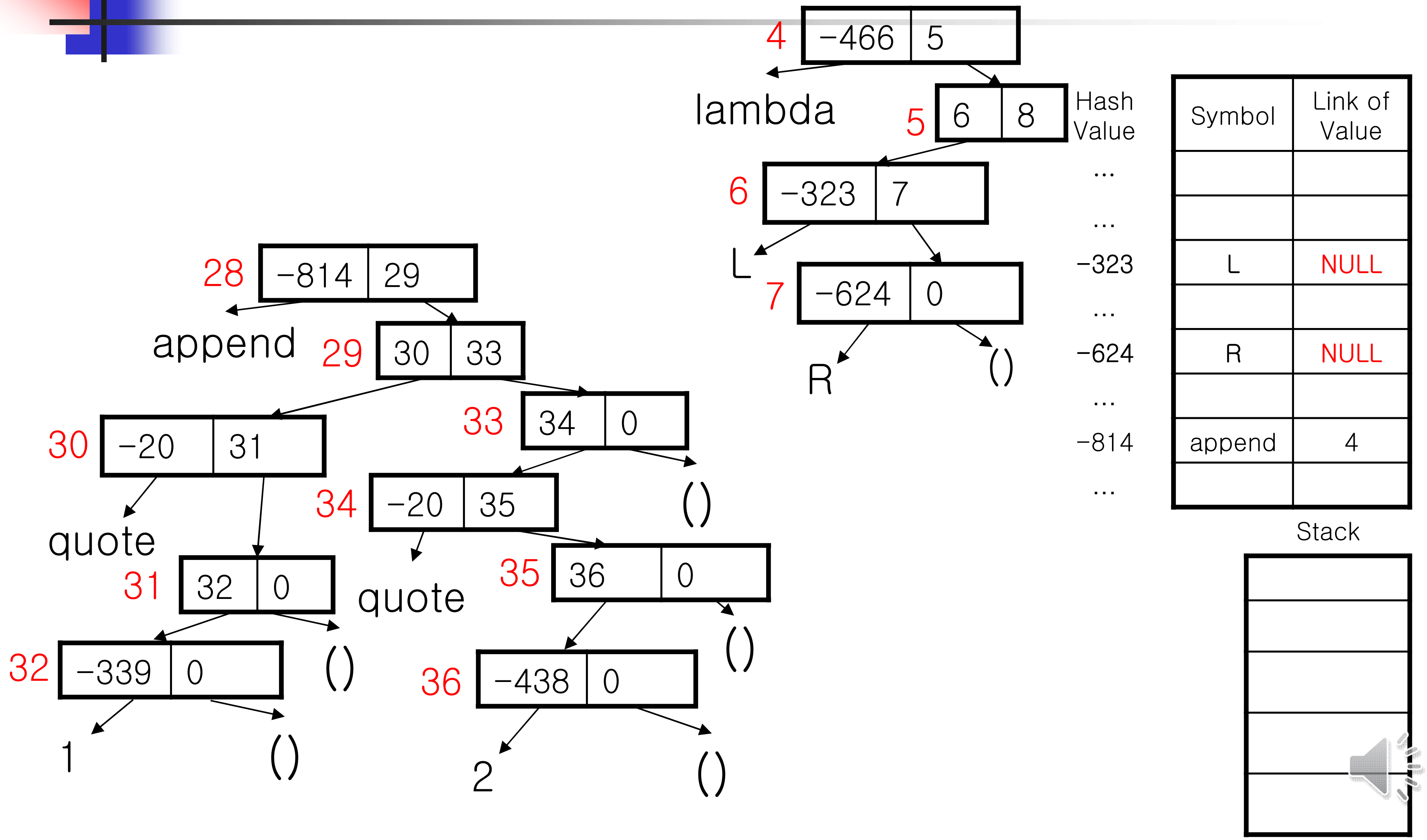


Hash Value	Symbol	Link of Value
…		
…		
−323	L	32
…		
−624	R	36
…		
−814	append	4
…		

Stack

R : NULL
L : NULL

Evaluation of (append '(1) '(2))



Hash Value	Symbol	Link of Value
…		
…		
−323	L	NULL
…		
−624	R	NULL
…		
−814	append	4
…		