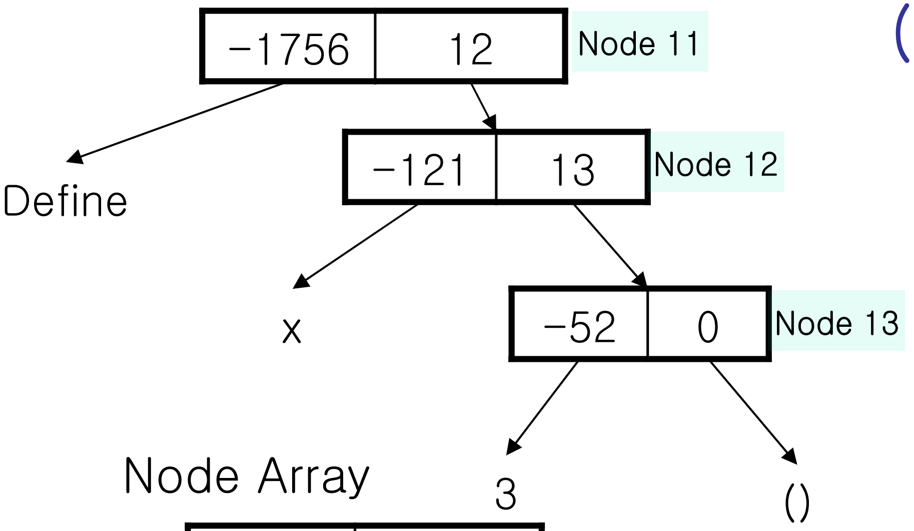
Scheme Implementation

Parse Tree & Node Array

- Build the parse tree with the node array when a command input comes
  - Read the input token by token
  - Make an new node and attach it to the tree
  - When we make a new node, allot it from the array
- Each node stores the indices of its left child and right child

Parse Tree & Node Array Example (1)

(define x 3)



Node Array

Array Index	Left	Right
…		
11	−1756	12
12	−121	13
13	−52	0

Symbol Hash Table

Hash Value	Symbol	Link of Value
…		
−52	3	NULL
…		
−121	x	NULL
…		
−1756	define	NULL
…		

Parse Tree & Node Array Example (2)

- **Input: (define x 3)**
  - Read the token '(' and allot the new root node (node 11) from the array.
  - Read the token ‘define’. Store the hash table index (= -1756) of 'define' as the left child index of node 11 and the node id (= 12) of the newly allotted node as the right child index of node 11.

Parse Tree & Node Array Example (3)

- Read the token 'x' and store the hash table index (= -121) of ‘x’ as the left child index of node 12. Then allot the new node and store the id (= 13) of it as the right child index of node 12.
- Read the next token ‘3’ and store the hash table index (= -52) of ‘3’ as the left child index of node 13. Since next token is ')', store null index (= 0) as the right child index of node 13 without allotting a new node. .

Symbol Table

- In Scheme, Symbol Table stores all “meaningful words”.
  - Built-in words, numbers, function names, symbols, etc.
  - Hash table should hold words and links for their contents.
- For fast search and retrieve, we’ll use hash table as a symbol table.
  - Use negative numbers for hash entries as its indices.
  - In the other hand, use positive numbers for entries of the node array

Symbol Table Example

Hash Value	Symbol	Link of Value
−1	“+”	NULL
−2	“car”	NULL
…		
−52	“3”	NULL
…		
−121	“x”	−52
…		
−3285	“list”	20
…		
−3501	“func”	25

- Assume we have a hash function “f” from each symbol to hash value.
- We assign some “special” region for “built−in words”.
  - f(“+”) = −1
- Hash function example
  - f(“3”) = −52, “3” is a number.
  - f(“x”) = −121, “x” is a variable.
  - f(“list”) = −3285, “list” is a list. List have a link for list data.
  - f(“func”) = −3501, “func” is a function. Function also have a link for function contents.

Collision Resolution (1)

- Two different words can have same hash value.
    - Hash table can store one element for one hash value. So, “collision problem” may happen.
- Here, we use Open Addressing Policy for hash table.
    - Resolve collision problem by using the first empty element from hash value.
    - If f(“3”)=–231 and f(“list”)=–231, the latter entry uses –232 as hash value.

Collision Resolution (2)

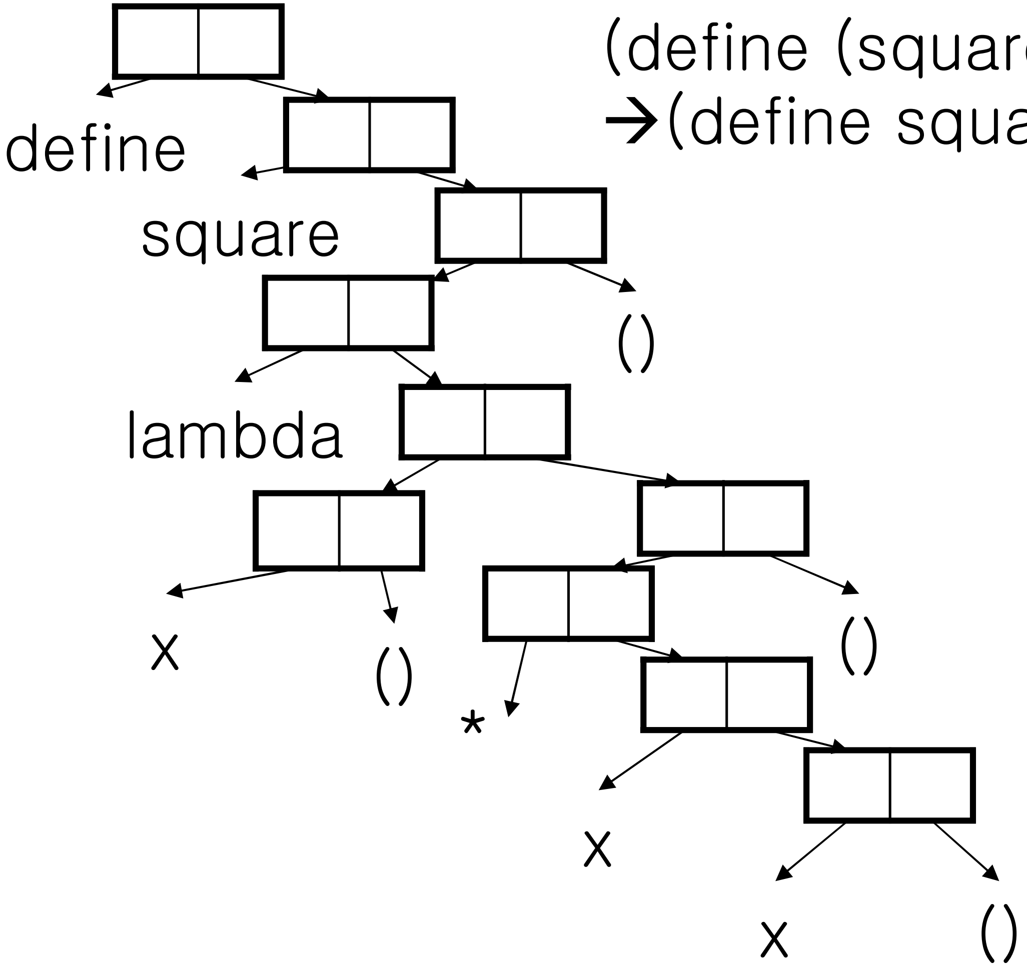
- Open Addressing has some weak points.
  - May make clusters, so search will be inefficient.
  - Very difficult to delete clustered entry.
- Here, We assume no delete operation and we have enough memory, so just use Open Addressing Policy.

Hash Value	Symbol	Link of Value
…	…	…
−231	“3”	NULL
−232	“list”	53
…	…	…

Example

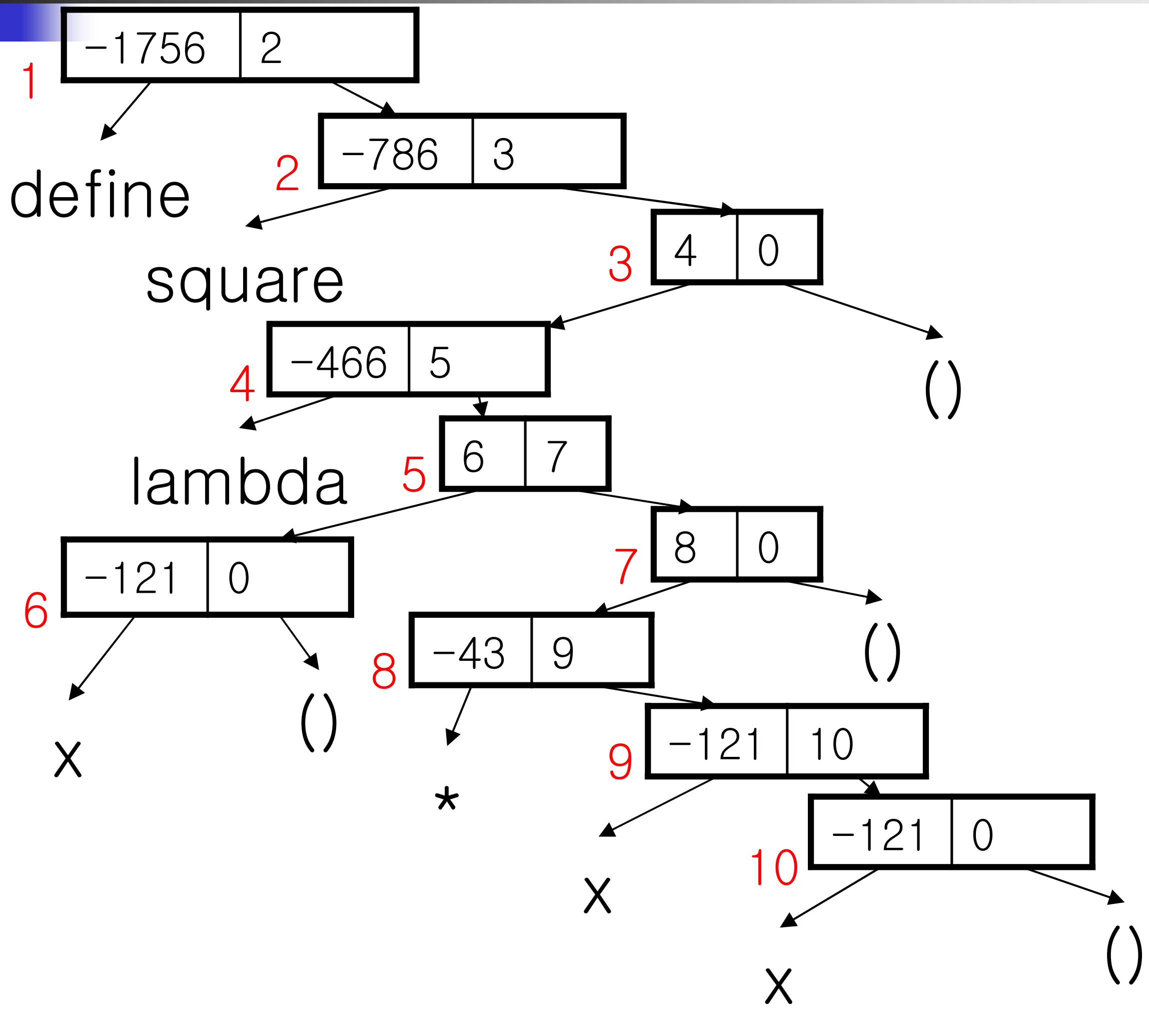
- Read & evaluate the following commands
  - (define (square x) (* x x))
  - (define x 3)
  - (square 6)

Convert non lambda form to lambda form



(define (square x) (* x x))
→(define square (lambda (x) (* x x)))

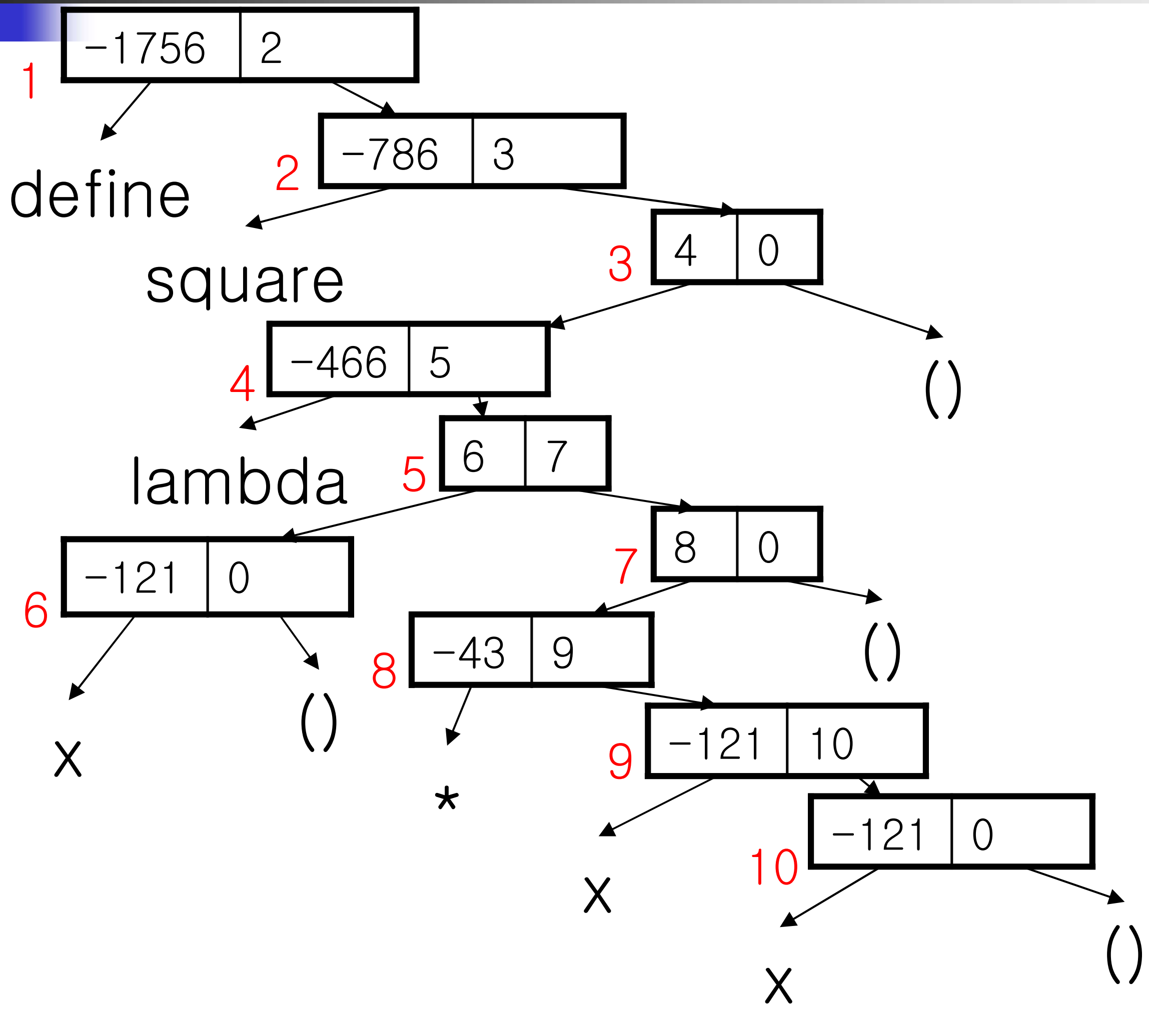
Parse Tree of (define square (lambda (x) (* x x)))



Node Array

Node ID	Left	Right
1	−1756	2
2	−786	3
3	4	0
4	−466	5
5	6	7
6	−121	0
7	8	0
8	−43	9
9	−121	10
10	−121	0

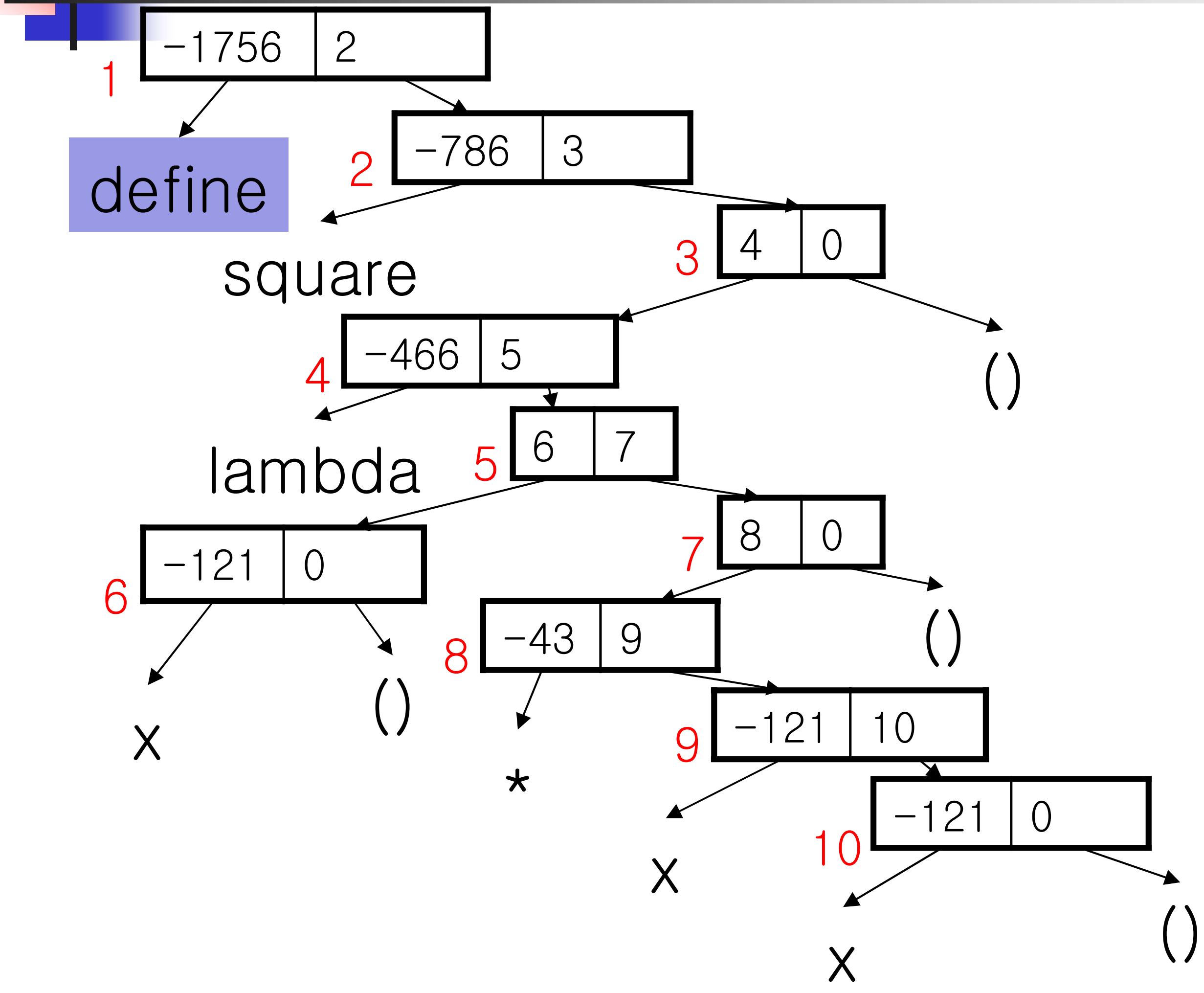
Parse Tree of (define square (lambda (x) (* x x)))



Hash Table

Hash Value	Symbol	Link of Value
…		
−43	*	
…		
−121	X	
…		
−466	Lambda	
…		
−786	square	
…		
−1756	Define	
…		

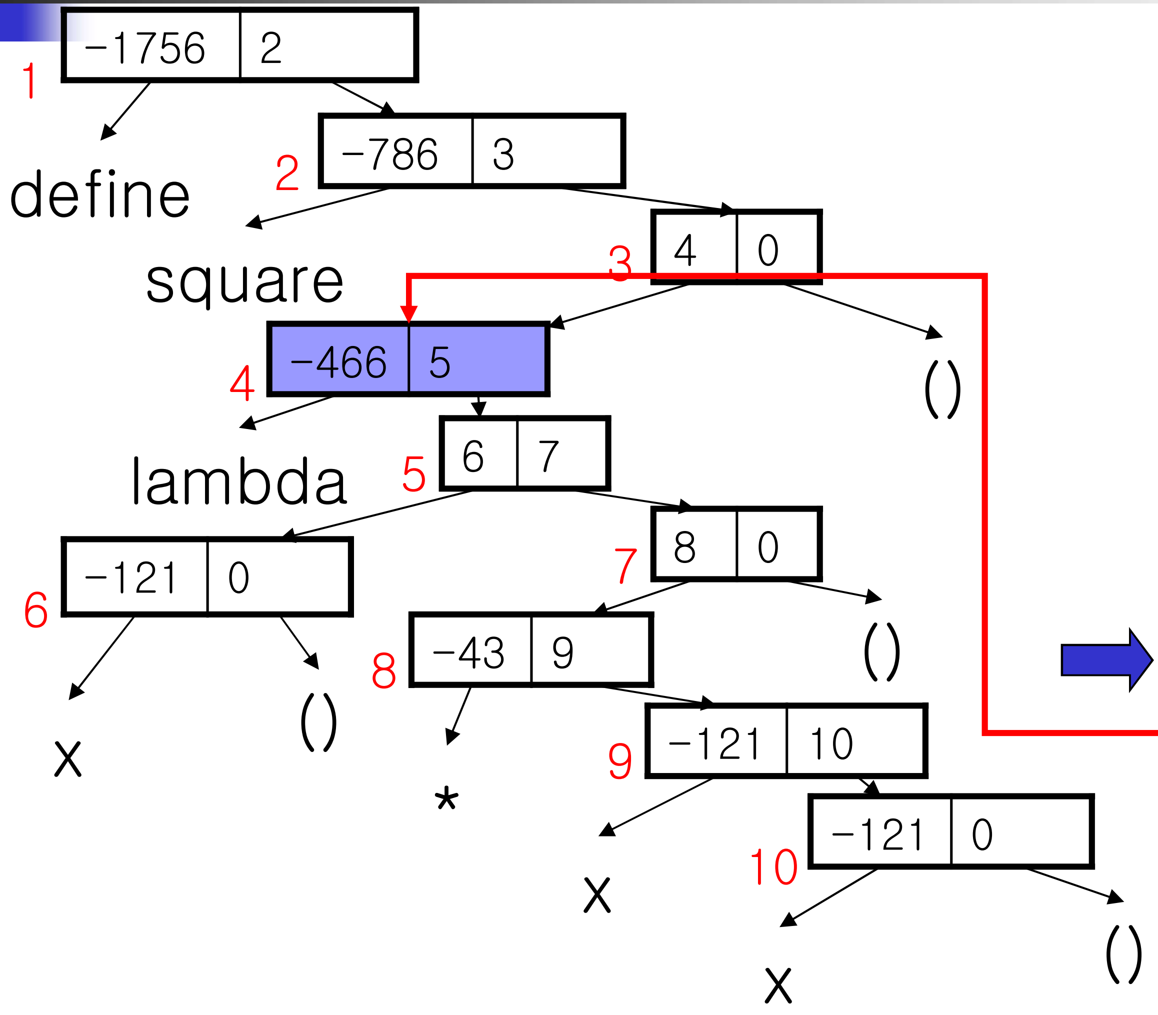
Evaluation of (define square (lambda (x) (* x x)))



Hash Table

Hash Value	Symbol	Link of Value
…		
-43	*	
…		
-121	X	
…		
-466	Lambda	
…		
-786	square	
…		
-1756	Define	
…		

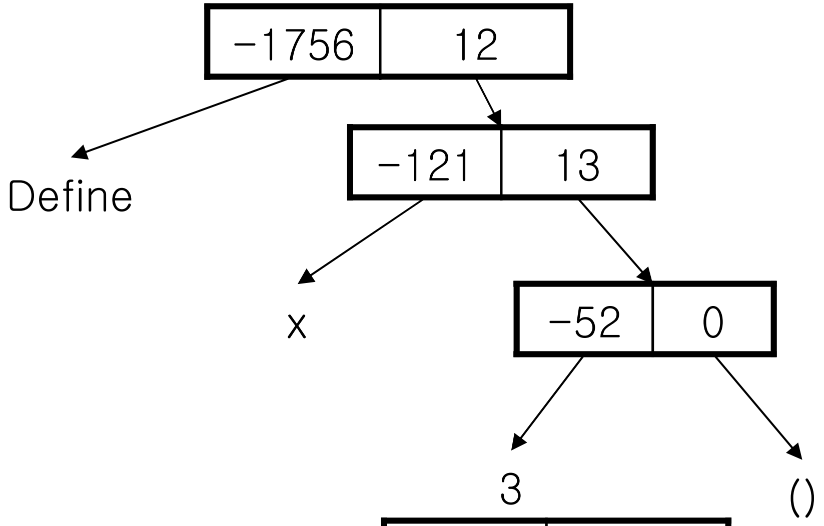
Evaluation of (define square (lambda (x) (* x x)))



Hash Table

Hash Value	Symbol	Link of Value
…		
−43	*	
…		
−121	X	
…		
−466	Lambda	
…		
−786	square	4
−1756	Define	
…		

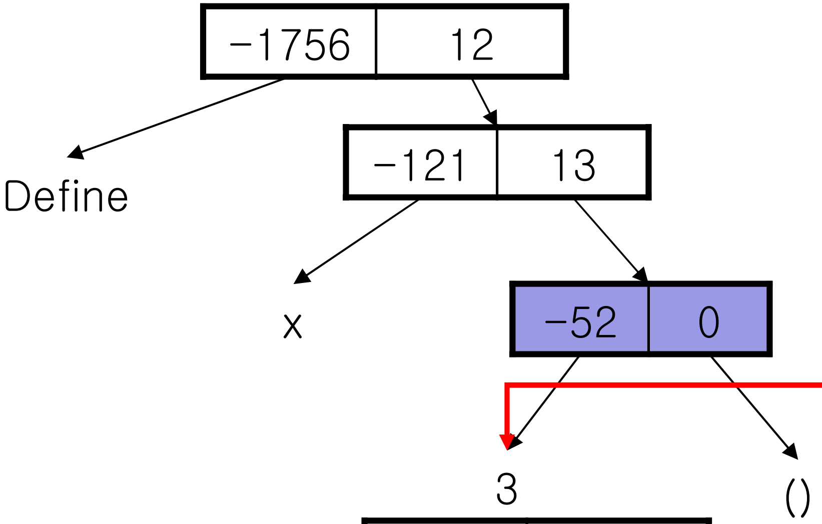
Parse Tree of (define x 3)



Node ID	Left	Right
…		
11	−1756	12
12	−121	13
13	−52	0

Hash Value	Symbol	Link of Value
…		
−43	*	NULL
…		
−52	3	NULL
…		
−121	X	NULL
..		
−466	Lambda	NULL
…		
−786	square	4
…		
−1756	Define	NULL
…		

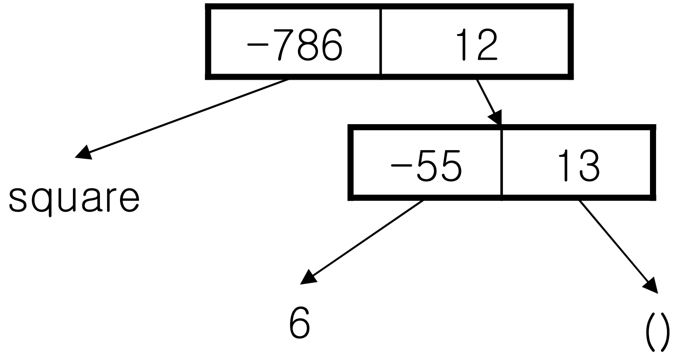
Evaluation of (define x 3)



Node ID	Left	Right
…		
11	−1756	12
12	−121	13
13	−52	0

Hash Value	Symbol	Link of Value
…		
−52	3	NULL
…		
−55	5	NULL
…		
−121	X	−52
−466	Lambda	NULL
…		
−786	square	4
…		
−1756	Define	NULL
…		

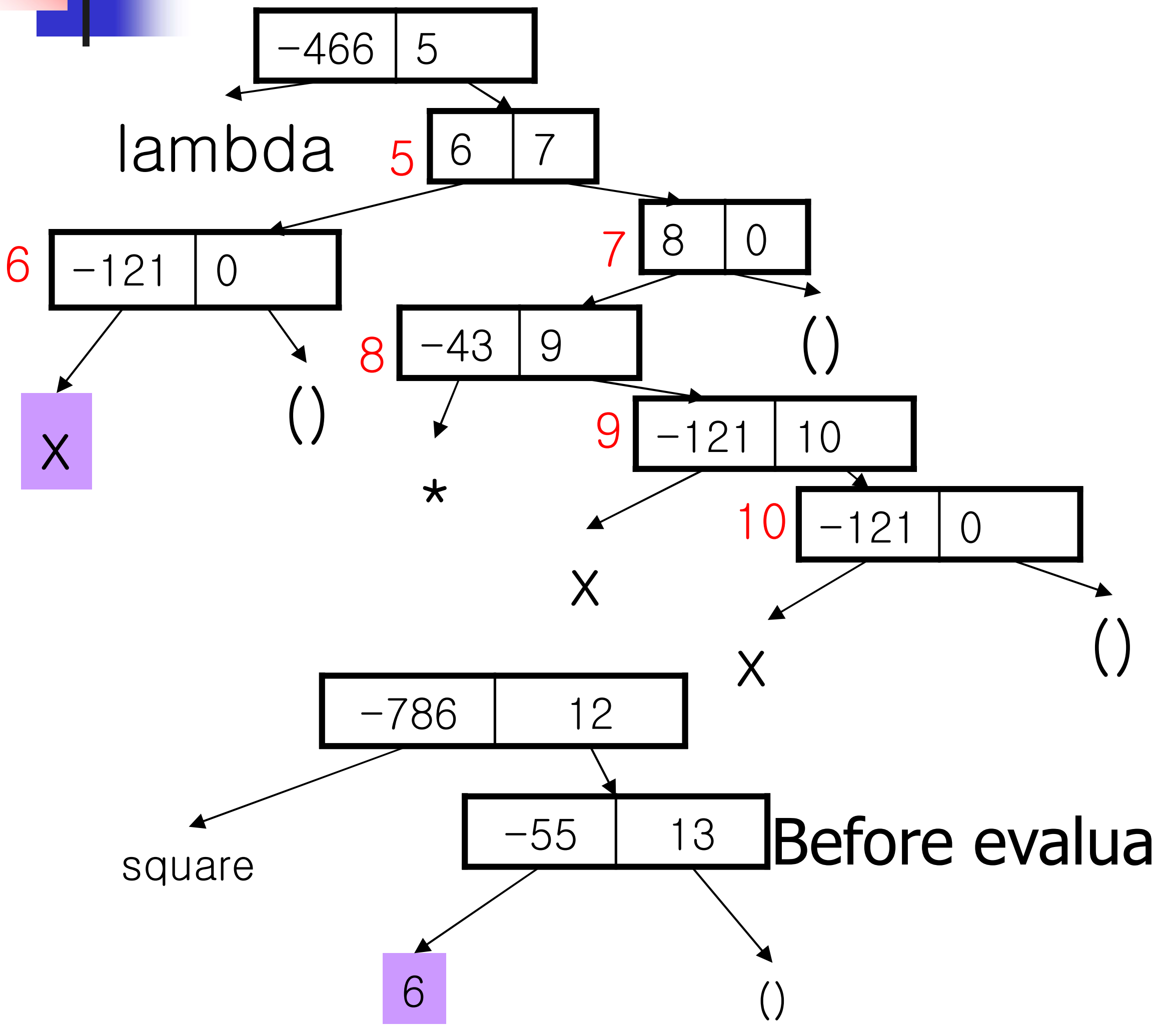
Parse Tree of (square 6)



Node ID	Left	Right
…		
14	−786	12
15	−55	13

Hash Value	Symbol	Link of Value
…		
−52	3	NULL
…		
−55	6	NULL
…		
−121	X	−52
…		
−466	Lambda	NULL
…		
−786	square	4
…		
−1756	Define	NULL
…		

Evaluation of (square 6)



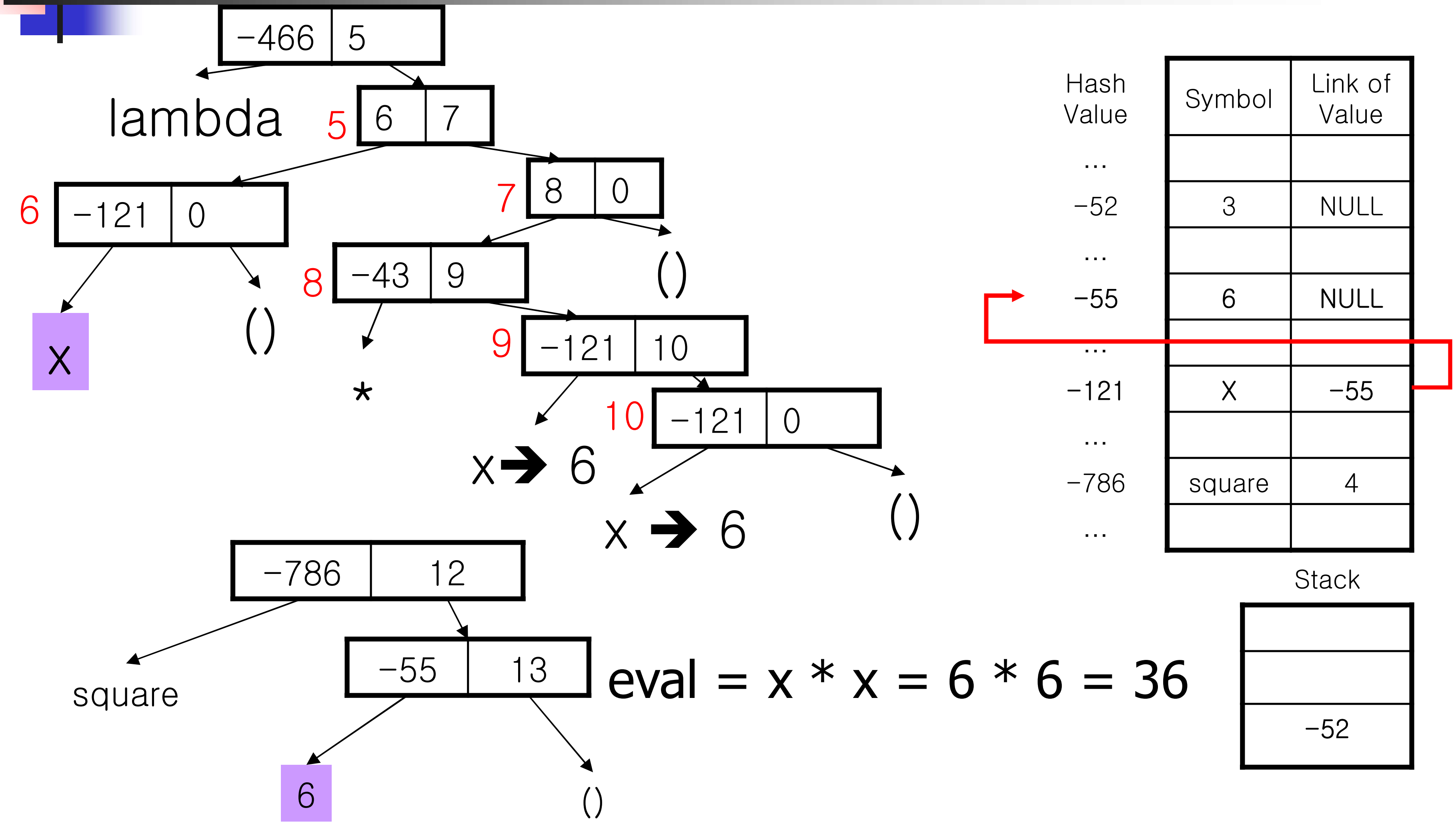
Before evaluation

Hash Value	Symbol	Link of Value
...		
−52	3	NULL
...		
−55	6	NULL
...		
−121	X	−55
...		
−786	square	4
...		

Stack

−52

Evaluation of (square 6)



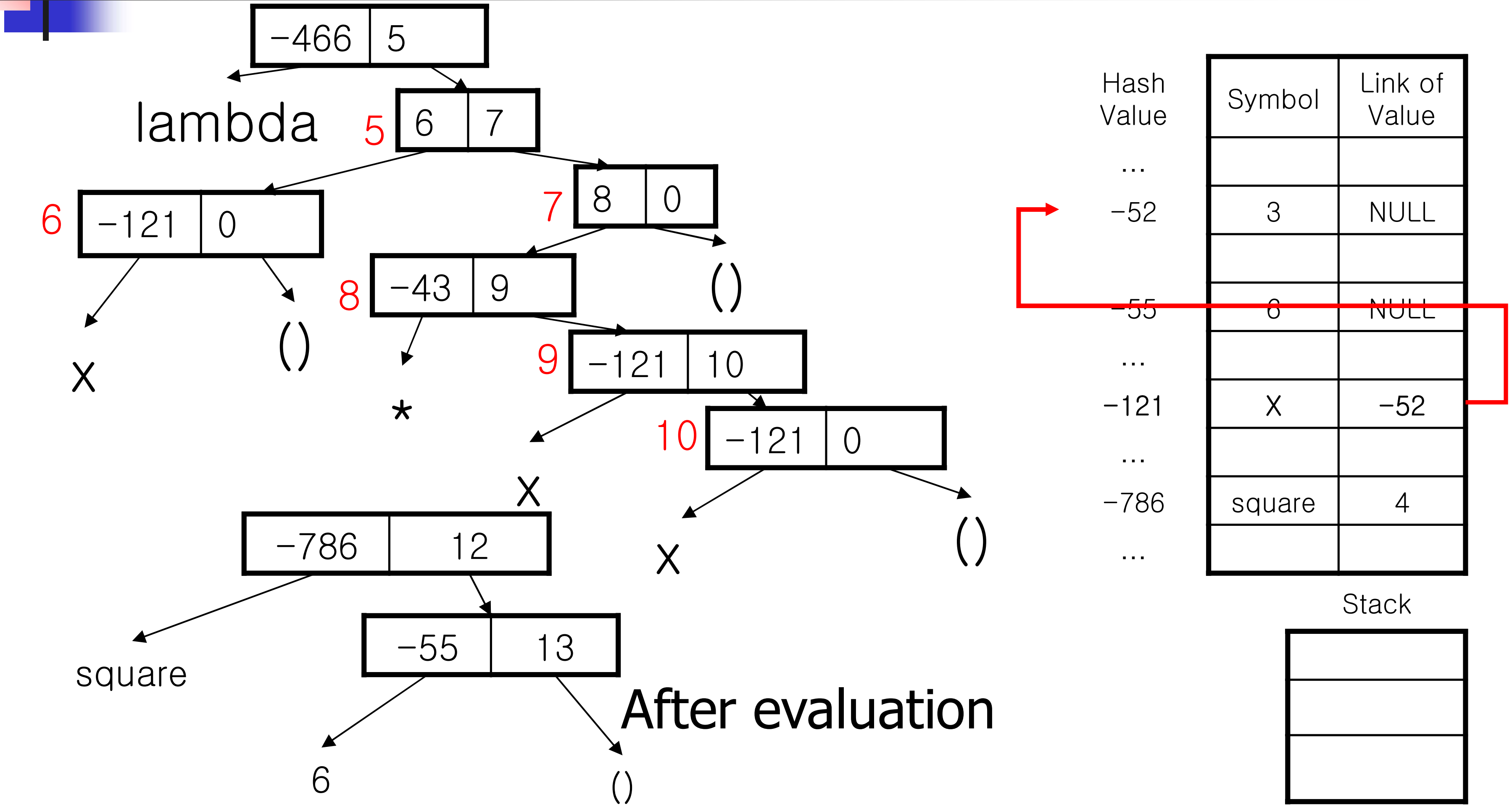
Hash Value	Symbol	Link of Value
…		
-52	3	NULL
…		
-55	6	NULL
…		
-121	x	-55
…		
-786	square	4
…		

Stack

-52

eval = x * x = 6 * 6 = 36

Evaluation of (square 6)



Hash Value	Symbol	Link of Value
…		
−52	3	NULL
−55	6	NULL
…		
−121	X	−52
…		
−786	square	4
…		

Stack

After evaluation

SCHEME INTERPRETER PSEUDO CODE (First Version)

```
Procedure Main()
begin
1. Initialize
2. while (true)
3.
4.
5.
6.     root := Read()
7.     result := Eval(root)
8.     if result is not NIL
9.         PrintResult(result)
end
```

SCHEME INTERPRETER PSEUDO CODE (First Version)

```
Procedure Read()
root := NIL
first := true
token hash value := GetHashValue(GetNextToken())
if token hash value is LEFT PAREN
    // iterate until ‘)’ appears
    while (token hash value := GetHashValue(GetNextToken()) is not RIGHT PAREN
        // the first entry of the list is read
        if first is true
            temp := Alloc()
            root := temp
            first := false
        // the remaining elements in the list should be put into the rchild
        else
            Memory[temp].rchild := Alloc()
            temp := Memory[temp].rchild
        // if the nested list appears, do recursion
        if token hash value = LEFT PAREN
            PushBack()
            Memory[temp].lchild := Read()
        else  Memory[temp].lchild := token hash value
        if first is false
            Memory[temp].rchild := NIL
    return root
else return token hash value
```

```
Procedure Read()
root := NIL
first := true
token hash value := GetHashValue(GetNextToken())
if token hash value is LEFT PAREN
    // iterate until ‘)’ appears
    while (token hash value := GetHashValue(GetNextToken())) is not RIGHT PAREN
        // the first entry of the list is read
        if first is true
            temp := Alloc()
            root := temp
            first := false
        // the remaining elements in the list should be put into the rchild
        else
            Memory[temp].rchild := Alloc()
            temp := Memory[temp].rchild
        // if the nested list appears, do recursion
        if token hash value = LEFT PAREN
            PushBack()
            Memory[temp].lchild := Read()
        else  Memory[temp].lchild := token hash value
        Memory[temp].rchild := NIL
    return root
else return token hash value
```

SCHEME INTERPRETER PSEUDO CODE (First Version)

```
PRINT(index, startList)
if root is NIL
    print ”()”
else if root < 0
    print hashTable[root].symbol
else if root > 0
    if startList is true
        print ”(”
    PRINT(Memory[root].lchild, true)
    if Memory[root].rchild is not NIL
        PRINT(Memory[root].rchild, false)
    else print ”)”
```