STRATEGY

I used shift reduce parser algorithm for parsing. For example:

Input: (+ 5b1 6b9)

Tokens: OP_OP OP_PLUS VALUEF VALUEF OP_CP

Iteration:

OP_OP

OP_PLUS
OP_OP OP_PLUS

VALUEF
OP_PLUS VALUEF
OP_OP OP_PLUS VALUEF

VALUEF
VALUEF VALUEF
OP_PLUS VALUEF VALUEF
OP_OP OP_PLUS VALUEF VALUEF

OP_CP
VALUEF OP_CP
VALUEF VALUEF OP_CP
OP_PLUS VALUEF VALUEF OP_CP
OP_OP OP_PLUS VALUEF VALUEF OP_CP

Every subsequence like this is sent into is_expression and the one I writed in bold is an expression so that part will be deleted and $EXP will be written instead of it.

Example from code:

Input: (* (/ 1b2 4b4) (- 2b1 1b1))

Tokens: OP_OP OP_MULT OP_OP OP_DIV VALUEF VALUEF OP_CP OP_OP OP_MINUS VALUEF VALUEF

OP_CP OP_CP

Steps:

```
serdar@serdar-VirtualBox:~/Desktop/HW3/lisp$ make input
clisp gpp_interpreter.lisp < input.gpp
RESULT:
(OP_OP OP_MULT OP_OP OP_DIV $EXP VALUEF OP_CP OP_OP OP_MINUS VALUEF VALUEF
 OP_CP OP_CP)

RESULT:
(OP_OP OP_MULT OP_OP OP_DIV $EXP $EXP OP_CP OP_OP OP_MINUS VALUEF VALUEF OP_CP
 OP_CP)

RESULT: (OP_OP OP_MULT $EXP OP_OP OP_MINUS VALUEF VALUEF OP_CP OP_CP)

RESULT: (OP_OP OP_MULT $EXP OP_OP OP_MINUS $EXP VALUEF OP_CP OP_CP)

RESULT: (OP_OP OP_MULT $EXP OP_OP OP_MINUS $EXP $EXP OP_CP OP_CP)

RESULT: (OP_OP OP_MULT $EXP $EXP OP_CP)

RESULT: ($EXP)

Result: 4b8
```

For **function definitions** when I see kw_def I saved all functions into a functions list for example:
Input:

(def sum x y (+ (+ x y) 1b1))
(def sub x y (- x y))

```
serdar@serdar-VirtualBox:~/Desktop/HW3/lisp$ make input
clisp gpp_interpreter.lisp < input.gpp
(OP_OP KW_DEF sum x y OP_OP OP_PLUS OP_OP OP_PLUS x y OP_CP 1b1 OP_CP OP_CP)
#function

(OP_OP KW_DEF sum x y OP_OP OP_PLUS OP_OP OP_PLUS x y OP_CP 1b1 OP_CP OP_CP OP_OP KW_DEF sub x y OP_OP OP_MINUS x y OP_CP OP_CP)
#function
```

This is the functions list for every function definition added.
The first identifier after kw_def is always a function name so I add the name into funcNames list.

Function call
Input: (sum 4b1 6b1)

If it is a function call, after the op_op there is an identifier, and it is the function name so I can search the name in funcNames list. If the name exists, I find the index of that function name in functions list.
For example, according to the input (sum 4b1 6b1) I find the index of 'sum' in functions list which is in our example. I save the next elements until next OP_OP into vars list which stores the variables.
Also in function call (sum 4b1 6b1) I store 4b1 and 6b1 into vals list.
In this situation indexes of values and variables are matched.
Example for index 0: in vars list: y
in vals list:6b1
Example for index 1: in vars list:x
in vals list:4b1
which shows y=6b1 and x=4b1

In the next part I iterate every element of functions list from:

start index: first OP_OP after function name index
end index: next kw_def or end of list

In this iteration when I see an element that is also stored in vars list I get the index of that element in vars list and push the element with same index in vals list into values list which is the list for storing values that will be used in add, sub, mul, div operations. If the element is a VALUEF like 1b1 I also push the value into values list.

After pushing the values, I change the identifier or 1b1 value as VALUEF to have a proper token list.
Next part is sending the tokens list and values into is_expression function and do the four operations.
Example for the input:

(def sum x y (+ (+ x y) 1b1))
(sum 4b1 6b1)
so this means y=6b1 and x=4b1
functions list: op_op kw_def sum x y **op_op** op_plus op_op op_plus x y op_cp 1b1 op_cp op_cp
starting from bolded op_op pushes 6b1 into values every time it sees a y in functions list
starting from bolded op_op pushes 4b1 into values every time it sees an x in functions list
when it sees a valuef it pushes it, 1b1 in our example

```
serdar@serdar-VirtualBox:~/Desktop/HW3/lisp$ make input
clisp gpp_interpreter.lisp < input.gpp
#function

#function

tokenlist: (OP_OP OP_PLUS OP_OP OP_PLUS VALUEF VALUEF OP_CP VALUEF OP_CP)

values: (4b1 6b1 1b1)

Result: 11b1
```