

Reasonable Grammar Rule Changes and Additions Documentation

- Variable Scoping: Variables are dynamically scoped which means that a variable is searched inside the variable stack starting from the last to the first. If a variable with the same name as a variable created before is created, then the last created one shadows the others.
- Expression evaluation: Expressions are evaluated from left to right.
- The function call production rules are refactored by adding OP_CP at the end of the rules since this language has LISP like syntax and there are close parentheses at the end of function calls in the example in the G++ Syntax Concrete pdf file.

```
$EXP -> OP_OP IDENTIFIER $EXP OP_CP |  
        OP_OP IDENTIFIER $EXP $EXP OP_CP |  
        OP_OP IDENTIFIER $EXP $EXP $EXP OP_CP
```

- In the examples in G++ Syntax Concrete pdf file, the fractions are given as “10f1” but it is said a fraction is in the format which numerator and denominator are separated with “b” letter like “10b1”, and this rule is valid for the previous homework. Therefore, in my G++ implementation the fractions are in “10b1” format.
- There were some missing rules in \$EXP and \$FUNCTION non-terminals production rules. In \$FUNCTION rules, there are functions defined with parameter numbers 0, 1 and 2. In \$EXP rules, there are function calls defined with parameter numbers 1, 2 and 3. Therefore, I combine two of them and provide function definitions and calls with parameter number 0, 1, 2, 3. Additional rules added are below:

```
$EXP -> OP_OP IDENTIFIER OP_CP  
$FUNCTION -> (def IDENTIFIER IDENTIFIER IDENTIFIER IDENTIFIER $EXP)
```

- All CFG rules defined in G++ Concrete Syntax pdf file are implemented and evaluated. Other than these, it seems there is no need to implement extra rules for other tokens since they are not mentioned in G++ Concrete Syntax pdf file.

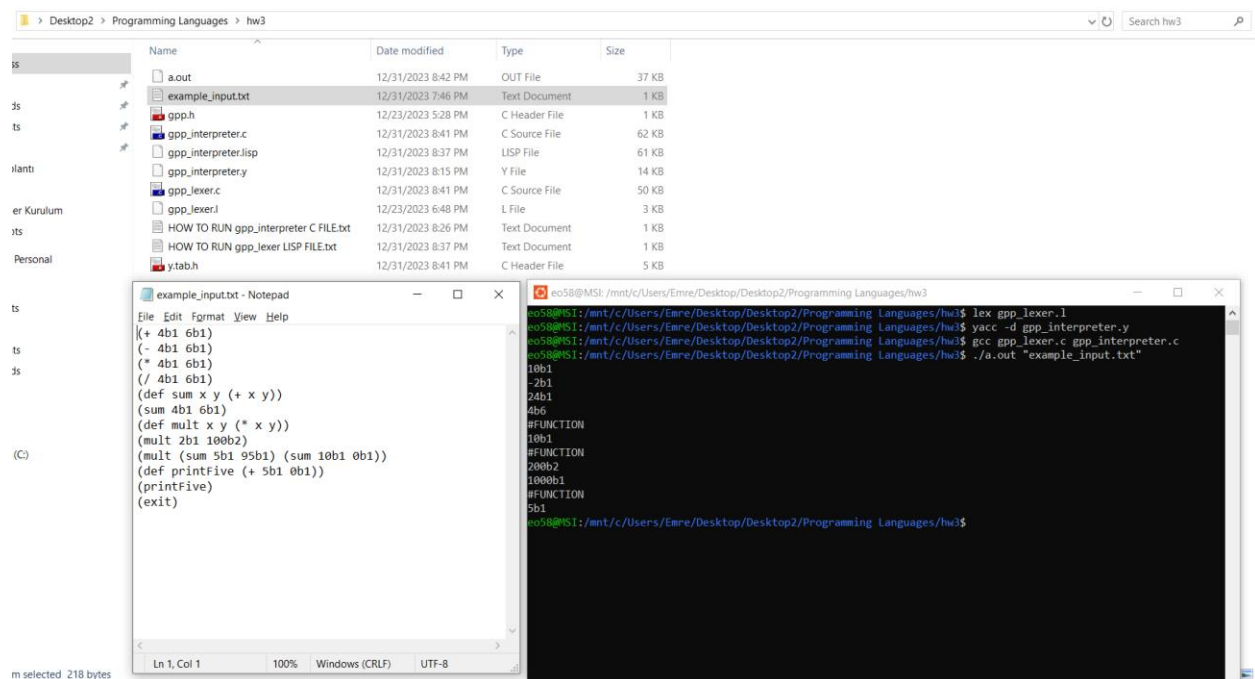
Abstract Syntax Rules for Evaluation:

Abstract syntax rules are added to evaluate the given instance of the language. These abstract syntax rules are obviously seen in the .yacc file so I do not repeat them here, but the main idea is keeping the expressions, functions, and variables in the struct format and building the abstract syntax tree while parsing the given input. When the start symbol is reached, then the

actual value evaluation process is started by using evaluate expression and evaluate function procedures.

- When VALUEF token is read, denominator == 0 check is made and if it is an error is printed.
- In division, denominator == 0 check is made and if it is an error is printed.
- In expression evaluation which has variable, an error is printed if the variable is not found in the variable stack.

Screenshots

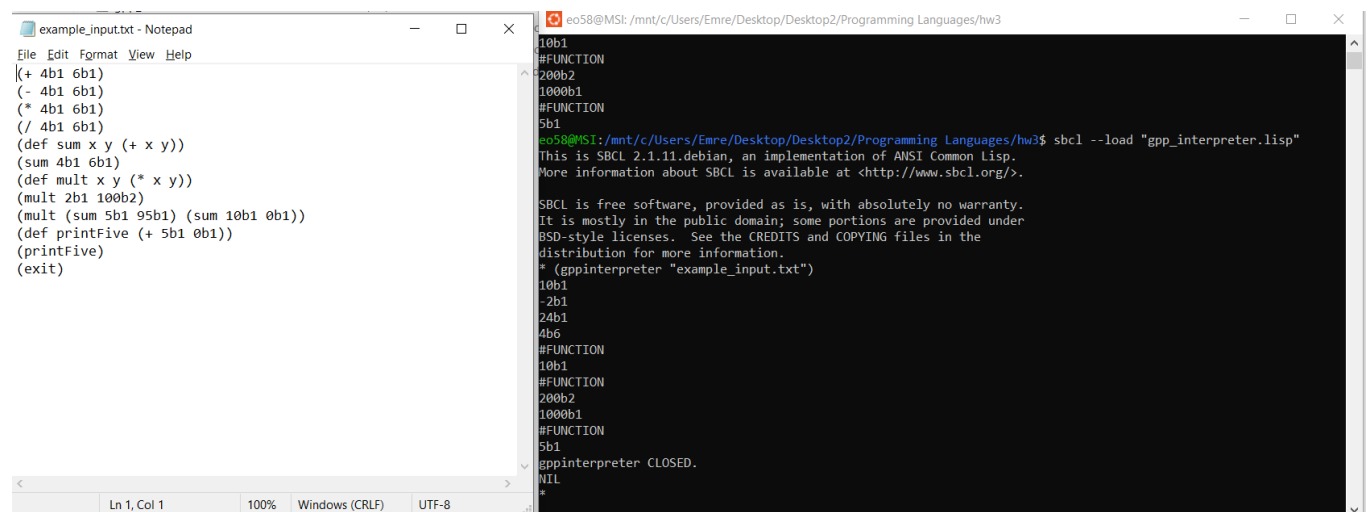


```
Desktop2 > Programming Languages > hw3
```

Name	Date modified	Type	Size
a.out	12/31/2023 8:42 PM	OUT File	37 KB
example_input.txt	12/31/2023 7:46 PM	Text Document	1 KB
gpp.h	12/23/2023 5:28 PM	C Header File	1 KB
gpp_interpreter.c	12/31/2023 8:41 PM	C Source File	62 KB
gpp_interpreter.lisp	12/31/2023 8:37 PM	LISP File	61 KB
gpp_interpreter.y	12/31/2023 8:15 PM	Y File	14 KB
gpp_lexer.c	12/31/2023 8:41 PM	C Source File	50 KB
gpp_lexer.l	12/23/2023 6:48 PM	L File	3 KB
HOW TO RUN gpp_interpreter C FILE.txt	12/31/2023 8:26 PM	Text Document	1 KB
HOW TO RUN gpp_lexer LISP FILE.txt	12/31/2023 8:37 PM	Text Document	1 KB
y.tab.h	12/31/2023 8:41 PM	C Header File	5 KB

```
example_input.txt - Notepad
File Edit Format View Help
(+ 4b1 6b1)
(- 4b1 6b1)
(* 4b1 6b1)
(/ 4b1 6b1)
(def sum x y (+ x y))
(sum 4b1 6b1)
(def mult x y (* x y))
(mult 2b1 100b2)
(mult (sum 5b1 95b1) (sum 10b1 0b1))
(def printFive (+ 5b1 0b1))
(printFive)
(exit)
```

```
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ lex gpp_lexer.l
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ yacc -d gpp_interpreter.y
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ gcc gpp_lexer.c gpp_interpreter.c
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ ./a.out "example_input.txt"
10b1
-2b1
24b1
4b6
#FUNCTION
10b1
#FUNCTION
200b2
1000b1
#FUNCTION
5b1
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$
```



```
example_input.txt - Notepad
File Edit Format View Help
(+ 4b1 6b1)
(- 4b1 6b1)
(* 4b1 6b1)
(/ 4b1 6b1)
(def sum x y (+ x y))
(sum 4b1 6b1)
(def mult x y (* x y))
(mult 2b1 100b2)
(mult (sum 5b1 95b1) (sum 10b1 0b1))
(def printFive (+ 5b1 0b1))
(printFive)
(exit)
```

```
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ sbcl --load "gpp_interpreter.lisp"
This is SBCL 2.1.11.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.

* (gppinterpreter "example_input.txt")
10b1
-2b1
24b1
4b6
#FUNCTION
10b1
#FUNCTION
200b2
1000b1
#FUNCTION
5b1
gppinterpreter CLOSED.
NIL
*
```

```
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ sbcl --load "gpp_interpreter.lisp"
This is SBCL 2.1.11.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
* (gppinterpreter)
>(+ 3b1 5b1)
8b1
>(/ 3b1 5b1)
3b5
>(- 3b1 5b1)
-2b1
>(* 3b1 5b1)
15b1
>(def addOne x (+ x 1b1))
#FUNCTION
>(addOne 3b1)
4b1
>(exit)
gppinterpreter CLOSED.
NIL
*
```

```
eo58@MSI:/mnt/c/Users/Emre/Desktop/Desktop2/Programming Languages/hw3$ ./a.out
(+ 3b1 5b1)
8b1
(/ 3b1 5b1)
3b5
(* 3b1 5b1)
15b1
(- 3b1 5b1)
-2b1
(def addOne x (+ x 1b1))
#FUNCTION
(addOne 4b1)
5b1
(exit)
```