GTU Department of Computer Engineering

CSE 341 - Fall 2024

Homework 3 Report

ALPER TAVŞANOĞLU

210104004142

Language Overview

The Simple Interpreter Language is designed to process code line by line and supports basic arithmetic operations, function definitions, S-expression evaluation strategy, and basic scoping rules. It uses a simplified representation for literals and supports the following operations: addition (+), subtraction (-), multiplication (*), and division (/).

Literal Representation

In the Simple Interpreter Language, literals are expressed using a specific format. Expressions in the form of 23f1 or 12f1 are interpreted as 23/1 or 12/1, respectively. The "b" notation represents a division operation, and the numerator and denominator are separated by a forward slash (/).

S-expression Evaluation Strategy

The Simple Interpreter Language adopts an S-expression evaluation strategy. S-expressions are nested expressions enclosed within parentheses. The interpreter evaluates S-expressions by recursively evaluating the sub-expressions from left to right. The result of the evaluation is the final value of the S-expression.

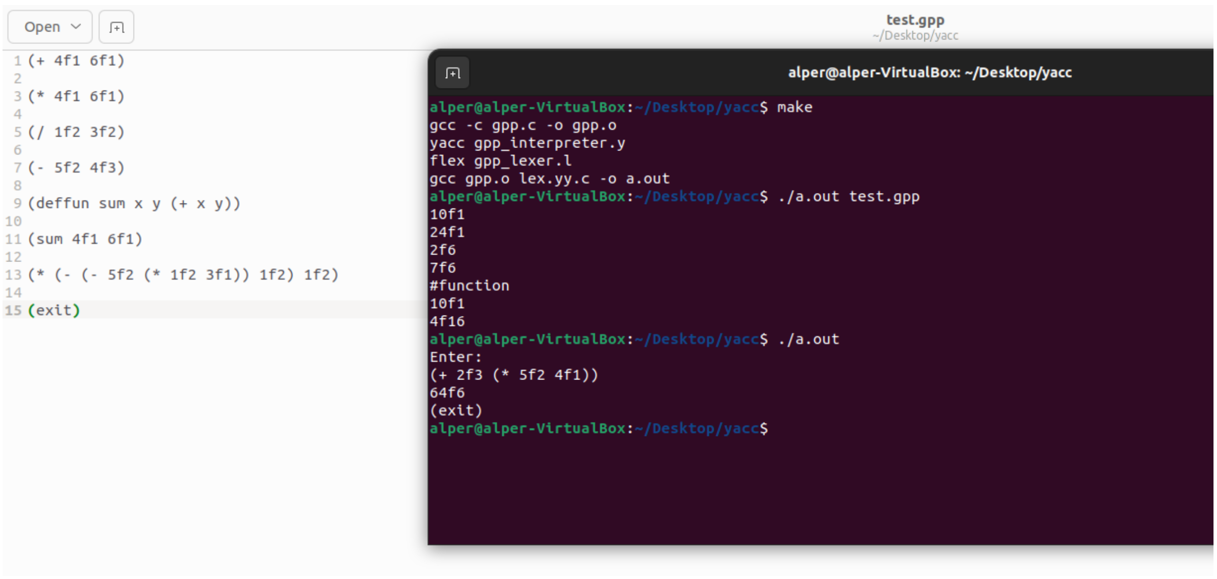
Conclusion

The Simple Interpreter Language follows basic scoping rules. Variables defined inside a function have local scope and are accessible only within that function.

The Simple Interpreter Language is a line-by-line interpreter language that supports literals, basic arithmetic operations, function definitions with optional parameters, S-expression evaluation strategy, and basic scoping rules. By leveraging these features, developers can write programs to perform calculations, define reusable code blocks, evaluate nested S expressions.

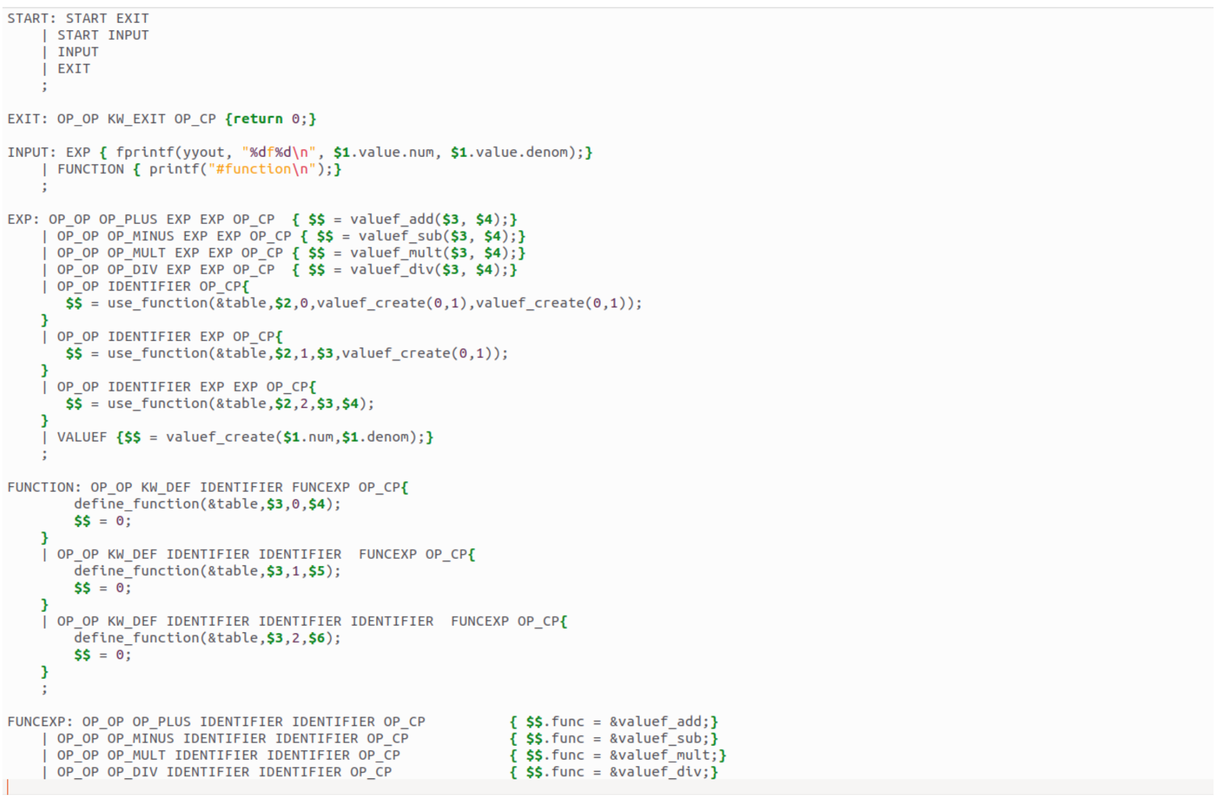
Examples

In part 1 i have gpp_interpreter.y and gpp_lexer.l with helper gpp.c and gpp.h files. Also makefile to compile them. After make command on terminal executable file can execute with **./a.out** command. And also, can read from test file and print to terminal with **./a.out test.gpp** command. Both usage example:



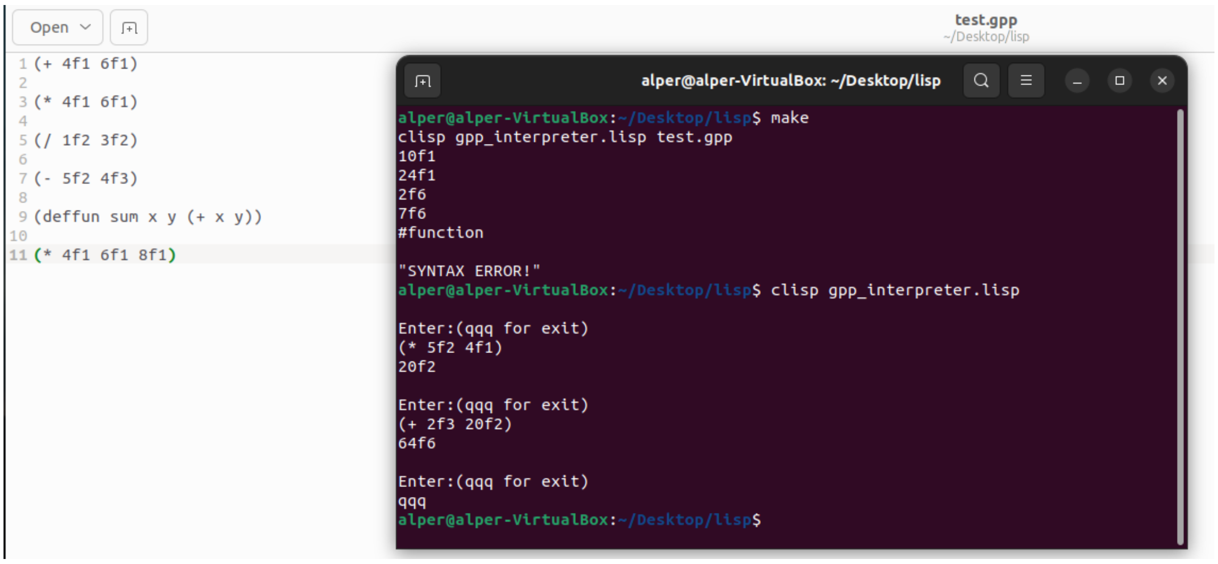
```
(+ 4f1 6f1)

(* 4f1 6f1)

(/ 1f2 3f2)

(- 5f2 4f3)

(deffun sum x y (+ x y))

(sum 4f1 6f1)

(* (- (- 5f2 (* 1f2 3f1)) 1f2) 1f2)

(exit)
```

```
alper@alper-VirtualBox:~/Desktop/yacc$ make
gcc -c gpp.c -o gpp.o
yacc gpp_interpreter.y
flex gpp_lexer.l
gcc gpp.o lex.yy.c -o a.out
alper@alper-VirtualBox:~/Desktop/yacc$ ./a.out test.gpp
10f1
24f1
2f6
7f6
#function
10f1
4f16
alper@alper-VirtualBox:~/Desktop/yacc$ ./a.out
Enter:
(+ 2f3 (* 5f2 4f1))
64f6
(exit)
alper@alper-VirtualBox:~/Desktop/yacc$
```



```
START: START EXIT
    | START INPUT
    | INPUT
    | EXIT
    ;

EXIT: OP_OP KW_EXIT OP_CP {return 0;}

INPUT: EXP { fprintf(yyout, "%df%d\n", $1.value.num, $1.value.denom);}
    | FUNCTION { printf("#function\n");}
    ;

EXP: OP_OP OP_PLUS EXP EXP OP_CP  { $$ = valuef_add($3, $4);}
    | OP_OP OP_MINUS EXP EXP OP_CP { $$ = valuef_sub($3, $4);}
    | OP_OP OP_MULT EXP EXP OP_CP { $$ = valuef_mult($3, $4);}
    | OP_OP OP_DIV EXP EXP OP_CP  { $$ = valuef_div($3, $4);}
    | OP_OP IDENTIFIER OP_CP{
      $$ = use_function(&table,$2,0,valuef_create(0,1),valuef_create(0,1));
    }
    | OP_OP IDENTIFIER EXP OP_CP{
      $$ = use_function(&table,$2,1,$3,valuef_create(0,1));
    }
    | OP_OP IDENTIFIER EXP EXP OP_CP{
      $$ = use_function(&table,$2,2,$3,$4);
    }
    | VALUEF {$$ = valuef_create($1.num,$1.denom);}
    ;

FUNCTION: OP_OP KW_DEF IDENTIFIER FUNCEXP OP_CP{
        define_function(&table,$3,0,$4);
        $$ = 0;
    }
    | OP_OP KW_DEF IDENTIFIER IDENTIFIER  FUNCEXP OP_CP{
        define_function(&table,$3,1,$5);
        $$ = 0;
    }
    | OP_OP KW_DEF IDENTIFIER IDENTIFIER IDENTIFIER  FUNCEXP OP_CP{
        define_function(&table,$3,2,$6);
        $$ = 0;
    }
    ;

FUNCEXP: OP_OP OP_PLUS IDENTIFIER IDENTIFIER OP_CP          { $$.func = &valuef_add;}
    | OP_OP OP_MINUS IDENTIFIER IDENTIFIER OP_CP            { $$.func = &valuef_sub;}
    | OP_OP OP_MULT IDENTIFIER IDENTIFIER OP_CP             { $$.func = &valuef_mult;}
    | OP_OP OP_DIV IDENTIFIER IDENTIFIER OP_CP              { $$.func = &valuef_div;}
```

In part 2 i have gpp_interpreter.lisp and makefile to compile. With make command code can run with test input or with **clisp gpp_interpreter.lisp** can run to wait user input.



```
(+ 4f1 6f1)

(* 4f1 6f1)

(/ 1f2 3f2)

(- 5f2 4f3)

(deffun sum x y (+ x y))

(* 4f1 6f1 8f1)
```

```
alper@alper-VirtualBox:~/Desktop/lisp$ make
clisp gpp_interpreter.lisp test.gpp
10f1
24f1
2f6
7f6
#function

"SYNTAX ERROR!"
alper@alper-VirtualBox:~/Desktop/lisp$ clisp gpp_interpreter.lisp

Enter:(qqq for exit)
(* 5f2 4f1)
20f2

Enter:(qqq for exit)
(+ 2f3 20f2)
64f6

Enter:(qqq for exit)
qqq
alper@alper-VirtualBox:~/Desktop/lisp$
```