Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.
Signature: Moro Bamber
Date: 4/3/2024

Buffalo

Prepared by

Moro Bamber

Content

1. Introduction

In this project we set out to create a mini compiler that recognizes basic math, trigonometry, flow control, and more. The tools we used for this were Flex and Bison in combination with C code. Flex is a 'Fast LEXical Analyzer' and Bison is a parser of grammar. When used together we can write statements that Flex will generate tokens for, and Bison will use those tokens according to rules we define to execute statements evaluated based on C code. The code Buffalo is based on can be found in chapter 3 of the book found in references [1].

The name of the mini compiler is named Buffalo. The name is derived from the parser being called Bison. It is able to do arithmetic, trigonometry, let users define variables and functions, and has flow control (if, else, while, etc.).

In this report we will provide information on how to run our compiler and write statements it recognizes. There will also be a demonstration of the code in Buffalo and a brief explanation on how it works.

2. Environment Configuration

To clone the Buffalo repository from GitHub:

https://github.com/UnderYourSpell/buffalo.git

2.1 Linux

While it is possible to run flex and bison on other platforms. We highly recommend using a Linux/GNU based operating system for the use of Flex and Bison as well as compilation and editing of Buffalo.

To install Flex and Bison on Linux using apt:

sudo apt install flex

sudo apt install bison

For Arch Linux use pacman:

sudo pacman -Sy flex

sudo pacman -Sy bison

2.2 Compile and Run

Running Buffalo is very simple. Go to the directory the source files are in and run the command:

```
[morob@archlinux buffalo]$ make
```

Now we have a buffalo executable in the directory.

```
buffalo          buf_lexical_analyzer.l  buf_parser.y     README.md
buf_header.h     buf_parser.tab.c        buf_routines.c   syntax.txt
buf.lex.c        buf_parser.tab.h        makefile
```

Now we can run our compiler.

```
[morob@archlinux buffalo]$ ./buffalo
>>
```

3. Functionality and Syntax

3.1.1 Basic Operations

Operations	Syntax
Addition	x + y
Subtraction	x – y
Multiplication	x * y
Division	x / y

3.1.2 Advanced Operations

Operation	Syntax
Modulo	x % y
Power	x ^ y
Factorial	x!
Absolute Value	|x
Parentheses	(x)
Negate Value	-x

In order of operations, the later in the bison code we define an operation the more precedence it has. Negating a value has the most precedence, parentheses have next highest precedence, addition has the least precedence.

3.1.3 Built in Functions

The code for buffalo allows for definition of C math functions defined in math.h. At compilation we link the math library with the flag -lm. Other C functions may also be called if defined as a token and have a return function associated with it.

3.2.1 Arithmetic

Operation	Syntax
Square Root	sqrt(x)
Exponential	exp(x)
Log_2	log(x)

3.2.2: Trigonometry

Operation	Syntax
Sine	sin(x)
Arcsine	asin(x)
Cosine	cos(x)
Arccosine	acos(x)
Tangent	tan(x)
Arctangent	atan(x)

3.2.3 Other

Dual Argument Math Functions

To demonstrate the calculators' capabilities, we added two math functions that take two arguments instead of one.

1. Power – pow(x,y). Example: pow(3,2) = 9
2. Arctan Squared - atan2(x,y)

Print

print(value)

The option to print a number or variable is possible with the syntax: print(x), with the output being: = x.

Random

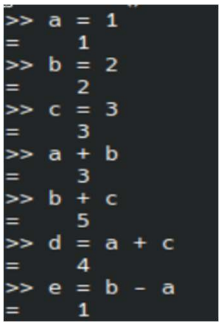
random(max value)

One can create a random number with a set max with random(x) where x is the maximum random number.

3.3 User Variables

A user can define variables according to specifications. A variable cannot be any of the keywords (seen later) nor any built-in function. It must also not start with a number, other than that it can be anything.

For example, we can define a, b, c, d, and e and perform arithmetic operations on them.



3.4 Flow Control

The first thing must understand about flow control in Buffalo is that it allows users to not only execute conditional statements and iteration statements, but also statement lists.

Statement Lists

A statement list is exactly what it sounds like, a list of statements. These statements are executed in the order they are written. A statement list can be written as follows:

<conditional or iteration statement> <expression> ; <expression>; <expression>;

It is simply an expression of some kind followed by a semicolon ';'. When reading on, when you see <list> know that it means a statement list.

3.4.1: Conditional Statements

Conditional Statements are supported by Bison.

To execute conditional statements, conditional operators were defined.

Statement	Syntax
Greater Than	>
Less Than	<
Not Equals	<> or !=
Equals	==
Greater or Equal	>=
Less or Equal	<=

If/Else

Buffalo supports If/Else Statements

Syntax:

if <cond> then <list>;

OR

if <cond> then <true list>; else <false list>;

Note the semicolon use after the true and false statements, these are necessary.

3.4.2: Iteration Statements

Buffalo has a while loop implementation as well as a custom iteration statement called 'loop'.

While

Like in other languages, a while statement will execute if the expression it evaluates each loop is true. It can be written as follows:

while <cond> do <list>;

Loop

Like a for loop, the 'loop' flow control statement allows a user to simply define the number of iterations to execute. It can be written as follows:

loop <num loops> do <list>;

3.5 User Defined Functions

Buffalo also supports defining a function. It is quite easy to do this, in Buffalo, we call it 'yak-ing' a function. We can define a function foo with arguments a and b as follows:

yak foo(a,b) = (a*b) – 5;

Flow control is also possible in user defined functions, take foo1 for example, which iterates n times, incrementing a variable c:

yak foo1(n,c) = loop n do c = c+1;;

Notice we must use ';;' at the end of the definition to indicate the end of both the loop statement and the function definition.

We will see more use of this in the Demonstration of Functionality section.

3.6 Strings

Buffalo supports the entry of a string. To enter a string, one must use "". For example:

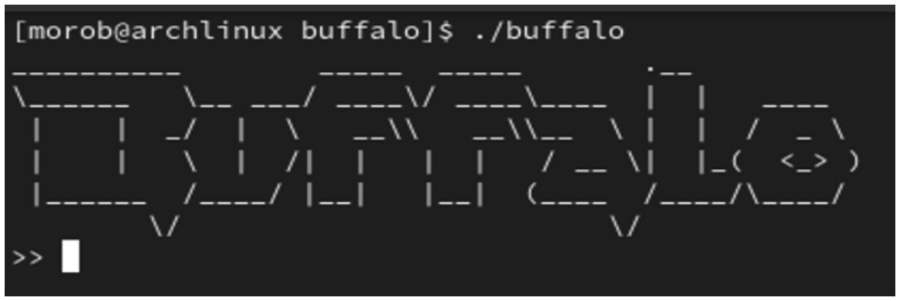
>> "hello"

String: "hello"

It recognizes that this token is not a part of the calculator syntax and acts accordingly. We did not implement any more features with strings but merely used this as a showcase in functionality.

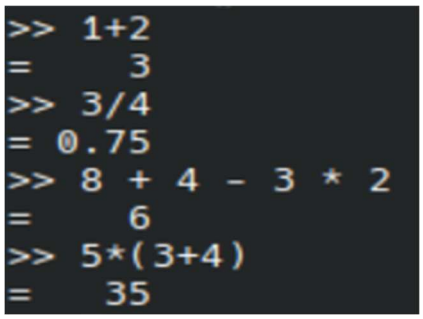
4. Demonstration of Functionality

In this section we will take you through writing statements in Buffalo and showcase its final functionality.

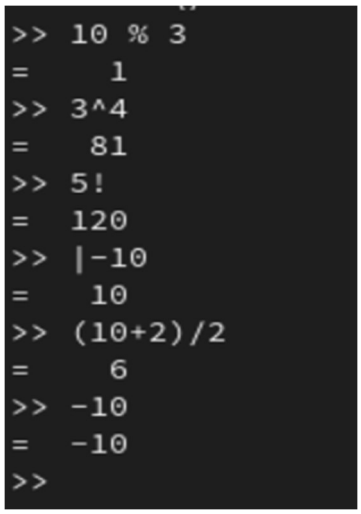


When we run the ./buffalo file this is what we are greeted with. We can now start executing expressions.

Arithmetic



Further Arithmetic Operations



Arithmetic Functions

```
>> sqrt(9)
=    3
>> exp(0)
=    1
>> log(2)
= 0.6931
>>
```

Trigonometry

```
>> sin(1)
= 0.8415
>> cos(0)
=    1
>> tan(1)
= 1.557
>> asin(1)
= 1.571
>> acos(0)
= 1.571
>> atan(0)
=    0
```

Dual Argument Functions

```
>> pow(3,4)
=   81
>> atan2(10,2)
= 1.373
```

Print

```
>> print(10)
=   10
```

Random

```
>> random(10)
=    3
>> random(40)
=    6
>> random(5)
=    2
```

User Variables (as shown earlier)

```
>> a = 1
=    1
>> b = 2
=    2
>> c = 3
=    3
>> a + b
=    3
>> b + c
=    5
>> d = a + c
=    4
>> e = b - a
=    1
```

Flow Control

Conditional 1

```
>> a = 1
=    1
>> b = 1
=    1
>> if a==b then print(a+b); else print(0);
=    2
```

Conditional 2

```
>> if a > b then print(a); else print(200);
=  200
```

While 1

```
>> var1 = 10
=   10
>> var2 = 0
=    0
>> while var1 > var2 do var2 = var2+1; print(var2);
=    1
=    2
=    3
=    4
=    5
=    6
=    7
=    8
=    9
=   10
```

While 2

```
>> x = 64
=   64
>> y = 1
=    1
>> z = 2
=    2
>> while z < x do z = z^y; print(z); y = y+1;
=    2
=    4
=   64
=    4
```

Loop 1

```
>> loop x do print(random(50));
=   35
=   36
=   42
=   49
=   21
=   12
=   27
=   40
=    9
=   13
=   26
=   40
=   26
```

Loop 2

```
>> loop 10 do if random(2) == 0 then print(0); else print(1);;
=    1
=    1
=    1
=    1
=    0
=    0
=    1
=    1
=    0
=    1
```

;; at the end of the statement to close the loop and if controls.

User Defined Functions (UDF)

UDF 1

```
>> yak printNTimes(n,x) = loop n do print(x);;
Defined printNTimes
-> 0
=    0
>> loop 3 do x = random(10); printNTimes(2,x);
=    3
=    3
=    6
=    6
=    7
=    7
```

UDF 2

```
>> yak printRandom(null)=0; if random(2)==0 then print(0); else print(1);;
Defined printRandom
-> printRandom(null)
=    1
= 0.8415
>> loop 10 do printRandom(null);
=    0
=    1
=    1
=    1
=    1
=    0
=    0
=    1
=    1
=    0
=    0
```

Here we can use the same code written for the loop 2 example and turn it into a function instead. When we define our argument, we must declare that it is 0 as the return value before using a flow control statement. If we do not have a flow control statement, then we do not need to define our arguments first,

UDF 3

```
>> yak printRandom(null)=0; if random(2)==0 then print(0); else print(1);;
Defined printRandom
-> 0
=    0
>> yak randLoop(n,null)=0; loop n do printRandom(null);;
Defined randLoop
-> 0
=    0
>> randLoop(5,0)
=    1
=    0
=    1
=    1
=    1
```

In this example we define another function that does what UDF 2 did without having to call the loop flow control.

UDF 4

```
>> yak sq(n)=e=1; while |((t=n/e)-e)>0.001 do e = avg(e,t);;
Defined sq
-> yak avg(a,b) = (a+b)/2;
Defined avg
-> 0
=    0
>> sq(10)
= 3.162
>> sqrt(10)
= 3.162
>> sq(10) - sqrt(10)
= 0.000178
```

In this example we are creating a square root function that is accurate to better than 0.001 cutoff, hence the discrepancy between our sq() and math.h's sqrt(). In this sq() function we also use a function avg(e,t) which hasn't been defined yet. We must define avg in the next line for the sq() function to work.

Strings

```
>> "Hello World"
String: "Hello World"
```

This is the extent of string functionality in Buffalo.

5. Code Explanation

To create Buffalo using Flex and Bison, 5 files are required.

1. Flex file for discerning tokens – buf_lexical_analyzer.l
2. Bison file for parsing - buf_parser.y
3. Header file for helper routines – buf_header.h
4. File for definition of routines – buf_routines.c
5. Makefile – compiles all the files together to make an executable

A makefile is also required to build buffalo into an executable.

5.1 buf_lexical_analyzer.l

Flex files are separated into three sections separated by %%, definitions, rules, and subroutines. In Buffalo, we are only using the first two sections.

Definitions

```
/* recognize tokens for the calculator */
%option noyywrap nodefault yylineno
%{
# include "buf_header.h"
# include "buf_parser.tab.h"
%}

/* float exponent */
EXP ([Ee][-+]?[0-9]+)
%%
```

Here in the definitions section we declare options, include files, and a way to make float exponent numbers.

yywrap() is a way for Flex to handle input. But we do not want it to do this, so we turn it off and use a different method for evaluating the end of an input.

nodefault tells flex not to define any default rules.

yylineno is a Flex variable used for tracking line numbers.

The files we include link the lexical analyzer to the parser and its routines.

Rules

The rules section of a Flex file allows one to use regular expressions to define tokens for the parser to interpret. The rules governing ambiguous tokens for Flex are simple, match the longest string every time the scanner matches input, and in the case of a tie, use the pattern that appears first. With these rules in mind, we define keywords and symbols before we allow Flex to interpret a variable name. This keeps the issue of ambiguous input at bay. For many of our tokens, we set a variable called yylval to a certain letter. This is to declare the type of token it is when we pass it to the parser. For every token we return something. For example, if we encounter "sqrt" we are telling the parser it is a FUNC token with the value of B_sqrt. Or if encounter a named variable, first we look for it in a symbol table, then return that it is a token of type NAME. The parser depends on Flex to let it know what kind of token it is when first encountered.

We will not show a few examples for different types of tokens and how we are reading them in in Flex. Note: Not all tokens for each class of token are shown, for brevity. If the example is included, all tokens of given class of token are defined the same.

Special Characters

```
";" |
"(" |
")" { return yytext[0]; }
```

Comparison Operators

```
">" { yylval.fn = 1; return CMP; }
"<" { yylval.fn = 2; return CMP; }
```

Built-In Functions

```
"sqrt" { yylval.fn = B_sqrt; return FUNC; }
"exp" { yylval.fn = B_exp; return FUNC; }
"log" { yylval.fn = B_log; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }
```

Flow Control

```
"if" { return IF; }
"then" { return THEN; }
"else" { return ELSE; }
```

User Variables, Constants, and Whitespace

```
[a-zA-Z][a-zA-Z0-9]* { yylval.s = lookup(yytext); return NAME; }

[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }

"//".*
[ \t] /* ignore whitespace */

\\\n { printf("c> "); } /* ignore line continuation */
```

End of Line and Unrecognized Token

```
\n { return EOL; }

. { yyerror("Mystery character %c\n", *yytext); }
```

5.2 buf_parser.y

The buf_parser.y file is the file where we define our bison parser. Like the flex program, bison files are split into the same 3 sections – Definitions, Rules, Subroutines. And like our Flex file, we are only using the first two sections. The subroutines are defined in the header and c supporting files. We include the header file, as well as standard c libraries in parser file.

In the parser file, we first declare the tokens and their type using %union. %union declares all the types of tokens to be used in the parser, and since different tokens may have

different types, we need to account for all these types. In the flex file, we saw that we were setting something called yyval to something. That value, whether it was s, d, or fn denoted the type of token. Here in the parser we match that type to an action. The type is declared un the %union.

```
%union {
    struct ast *a;
    double d;
    struct symbol *s; /*which symbol*/
    char* g;
    struct symlist *sl;
    int fn; /*which function*/
    int ft; /*which 2 arg function*/
}
```

We then tell the parser the different types of non-terminal symbols that exist as well as the start symbol 'calclist'.

```
%type <a> exp stmt list explist string_expr
%type <sl> symlist
%start calclist
```

Once we have declared the types of tokens that the parser recognizes, we define our grammar.

While calclist is defined last in the parser since it has the most precedence, we will show it first as it provides good insight into the operation of the calculator.

```
calclist: /* nothing */
    | calclist stmt EOL {
        printf("= %4.4g\n>> ", eval($2));
        treefree($2);
    }
    | calclist LET NAME '(' symlist ')' '=' list EOL {
            dodef($3, $5, $8);
            printf("Defined %s\n-> ", $3->name); }
    | calclist string_expr EOL{
        printf("String: %s", $2);
    }
    | calclist error EOL { yyerrok; printf(">> "); }
;
```

Here it shows that when we start a calclist, we have a statement between it and the end of a line. When this is the case, we evaluate '$2' or the second token in this sequence, which is the stmt or statement we wish to evaluate. The value of this statement is then printed for us, and we can then write another statement.

We will now present the code that constitutes the grammar of the calculator in order of precedence.

```
stmt: IF exp THEN list           { $$ = newflow('I', $2, $4, NULL); }
    | IF exp THEN list ELSE list { $$ = newflow('I', $2, $4, $6); }
    | WHILE exp DO list          { $$ = newflow('W', $2, $4, NULL); }
    | exp
    | LOOP exp DO list{$$ = newflow ('R',$2,$4,NULL);}
;
```

```
list: /* nothing */ { $$ = NULL; }
    | stmt ';' list { if ($3 == NULL)
        $$ = $1;
    else
        $$ = newast('L', $1, $3);
    }
;
```

```
exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
    | exp '+' exp { $$ = newast('+', $1,$3); }
    | exp '-' exp { $$ = newast('-', $1,$3);}
    | exp '*' exp { $$ = newast('*', $1,$3); }
    | exp '/' exp { $$ = newast('/', $1,$3); }
    | exp '%' exp { $$ = newast('%', $1,$3); }
    | exp '^' exp { $$ = newast('^', $1,$3); }
    | exp '!' { $$ = newast('!', $1, NULL); }
    | '|' exp { $$ = newast('|', $2, NULL); }
    | '(' exp ')' { $$ = $2; }
    | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
    | NUMBER { $$ = newnum($1); }
    | NAME { $$ = newref($1); }
    | NAME '=' exp { $$ = newasgn($1, $3); }
    | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
    | NAME '(' explist ')' { $$ = newcall($1, $3); }
    | FUNC2 '('exp','exp')'{ $$ = newfunc2($1, $3, $5);}
;
```

```
string_expr: STRING {$$ = $1;}
```

```
explist: exp
    | exp ',' explist { $$ = newast('L', $1, $3); }
;
```

```
symlist: NAME { $$ = newsymlist($1, NULL); }
    | NAME ',' symlist { $$ = newsymlist($1, $3); }
;
```

Calclist follows symlist in this order, but it has already been show.

5.3 Abstract Syntax Trees (AST)

'One of the most powerful data structures used in compilers is an *abstract syntax tree*' [P51 Flex and Bison book]. Our calculator's grammar is built using an AST. One may have noticed that with each way to evaluate a statement, a function called new__($x,$x,$x) is called. This family of function is doing something very simple, it is creating a node in an abstract syntax tree based on the type of token it has encountered. For example, when we encounter a token that returns as a FUNC2, we look at the statement and assign the node input parameters. For a FUNC2 token, we take the name in $1 – or the first thing in the sequence encountered - $3 is argument 1, and $5 is argument 2. Now we have a FUNC2 node defined and created. An AST is built with all these nodes, and subsequently evaluated.

5.4 buf_header.h

The main functionality of the calculator is done by using AST's. We need code to define the behavior and evaluation of these AST's. In the header file, we declare node types and declare functions for use in the accompanying C file.

Note: not all node and function definitions are included here for reasons of brevity.

```
/* symbol table */
struct symbol { /* a variable name */
    char *name;
    double value;
    struct ast *func; /* stmt for the function */
    struct symlist *syms; /* list of dummy args */
};
/* simple symtab of fixed size */
```

Here we define a symbol table to be able to lookup the value of user variables.

```
struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};

struct fncall { /* built-in function */
    int nodetype; /* type F */
    struct ast *l; //exp or explist
    //struct ast *r;
    enum bifs functype;
};
```

In this code we define an AST with pointers to a left and right leaf as well as a node type. The next function 'fncall' is an example of one of the types of nodes we define. We say it has a node type of 'F', it has an expression or expression list to evaluate, and we say which function it is.

```
enum bifs { /* built-in functions */
    B_sqrt = 1,
    B_exp,
    B_log,
    B_print,
    B_sin,
    B_cos,
    B_tan,
    B_asin,
    B_acos,
    B_atan,
    B_rand
};

enum bifs2 {
    B_pow = 1,
    B_atan2,
};
```

This is what the line 'enum bifs functype' is referring to in the previous code. We have several built in functions and enumerate them.

```
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newfunc2(int functype, struct ast *l, struct ast *r);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);
struct ast *newstring(const char *value);
/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);
/* evaluate an AST */
double eval(struct ast *);
/* delete and free an AST */
void treefree(struct ast *);
/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);
```

Finally, we declare all the functions to be defined in the buf_routines.c file.

5.5 buf_routines.c

This C file contains most of the functionality is evaluating AST's. Since we are using C, we must also manage memory allocation. Every node we create, we must free that space when we are done using it. So, in buf_routines.c we define our node calls, decide what to do with it and how to evaluate the information in them, free the space, then run a parse again according to the grammar defined in the bison file.

Note: Not all code in buf_routines.c is shown for brevity. The file is nearly 550 lines of code. We will be showing important aspects of functionality.

```
struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    return a;
}
```

Definition of newest

```
struct ast *
newfunc(int functype, struct ast *l)
{
    struct fncall *a = malloc(sizeof(struct fncall));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->functype = functype;
    return (struct ast *)a;
}
```

Definition of newfunc. We allocate the size of the call, tell the ast the node is of type 'F', and pass in the expression to evaluate in the left leaf of this node.

```
double eval(struct ast *a) {
    double v;
    char* str;
    if (!a) {
        yyerror("internal error, null eval");
        return 0.0;
    }
    switch (a->nodetype) {
        /* constant */
        case 'K':
            v = ((struct numval *)a)->number;
            break;
        /* name reference */
        case 'N':
            v = ((struct symref *)a)->s->value;
            break;
        case 'S':
            str = ((struct ast_string *)a)->value;
            v = strlen(str);
            printf(str);
        /* assignment */
        case '=':
            v = ((struct symasgn *)a)->s->value = eval(((struct symasgn *)a)->v);
            break;
```

As seen in buf_parser.y, we call eval on every line passed in. This is that eval function. Given a node type, we evaluate accordingly.

```
case 'W':
    v = 0.0; /* a default value */
    if (((struct flow *)a)->tl) {
        while (eval(((struct flow *)a)->cond) != 0) // evaluate the condition
            v = eval(((struct flow *)a)->tl); // evaluate the target statements
    }
    break; /* value of last statement is value of while/do */
/* list of statements */
case 'R':
    v = 0.0; //default
    for(int i = 0; i < (eval(((struct flow *)a)->cond)); i++){
        v = eval(((struct flow *)a)->tl); //execute statement certain amount of times
    }
    break;
```

This is the code for evaluating a while and loop flow. We are using C while and for loops and adjusting their behavior according to the line passed in.

```
static double
callbuiltin(struct fncall *f)
{
    enum bifs functype = f->functype;
    double v = eval(f->l);
    switch(functype) {
    case B_sqrt:
        return sqrt(v);
    case B_exp:
        return exp(v);
    case B_log:
        return log(v);
```

This is the code that allows us to define built-in functions. Given a certain function identified, do a certain math function.

Main Function

The main function in our program has one line of code being:

```
return yyparse();
```

All we are doing here is to tell Bison to parse the statement according to our rules.

5.6 Makefile

Since we have 4 user defined files, and 2 files generated for us by Flex and Bison, we must use a makefile to streamline the compilation process. We will walk through it line by line.

```
buffalo:    buf_lexical_analyzer.l buf_parser.y buf_header.h
            bison -d buf_parser.y
            flex -o buf.lex.c buf_lexical_analyzer.l
            cc -o $@ buf_parser.tab.c buf.lex.c buf_routines.c -lm
```

In the first line, we are calling our out file buffalo, and starting the process of compilation by declaring the files we will be using.

In the next line we compile our bison parser with the -d meaning to link the bison library.

Next we compile our flex program by saying our out file will be called buf.lex.c and the file to compile is bif_lexical_analyzer.l.

Finally, we use cc to compile, with the out file being $@ referring to 'buffalo' on line 1. We tell the compiler to use buf_parser.tab.c – a file generated for us by bison – buf.lex.c – our file generated by flex – and our functions file buf_routines.c – where we also use the -lm flag to link the math library.

By running 'make' on a Linux terminal, we get the out file called buffalo which can be run with $./buffalo.

We now have our running buffalo mini compiler up and running!

6. Limitations

It took some time to really understand how the code works in this project. Our first roadblock was trying to add string functionality. While in the end, we were able to get the compiler to recognize strings, we were not able to have it do anything with them. Another limitation we saw was that the syntax for writing complex statements can be tricky. For example, when defining a function, sometimes you must set the parameters equal to something to have it work. And at the end of defining a function, using ';;' as a termination symbol is not ideal. We also wanted to be able to pass in a file to the parser, but since it works line by line, we understood that the code changes would be drastic and out of the scope of this project.

7. Conclusion

We built a mini compiler that functions as a calculator successfully. Our application performs basic arithmetic but also includes higher level math functions like trigonometry as well. It also includes functionality to use flow control like conditional statements and loops. It lets users define variables and functions for use throughout runtime of Buffalo. This project taught us a lot about how programming languages are read by a compiler and how we can implement syntax practically. To do anything in this calculator requires a ton of supporting routines that maintain memory, user variables and functions, and evaluation of any given statement. We were excited about working on this project as we saw some interesting code and learned how programming languages can be implemented.

8. Reference

1. Levine, John R. *Flex and Bison*. O'Reilley, 2009.

9. Appendix

1. Source Code: https://github.com/UnderYourSpell/buffalo
2. Video Presentation Link: https://youtu.be/WdPUu3AjUCc