

A Comparison Between Old Methods of Creating Interpreters and The New Tools Like PLY (Lex and YACC)

Mike Anjomani

December 8th 2022

Preface

Compilers have come a long way since the early days of computing and have gone through a lot of changes and advancements, adopting the newest technology and helping with the task of programming digital computers, which historically has been a tedious task, in a really elegant way by abstracting away the lower layers of code running on digital processor and contributing to task of hardware programming with freedom of programmer from the lower levels of computing. The term assembly code is found in Kathleen and Andrew Donald Booth's 1947 work, *Coding for A.R.C.* ^[1]. The term "assembler" which is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer.* ^[2]. Wilkes, a remarkable computer scientists is known to be the first person who was awarded PhD in computer science. ^[3], the compilers have changed the art of programming and converting mathematical algorithms to low level code to be run in digital computer systems.

As a computer enthusiast who was dreaming of building my very own computer from scratch with off the shelf digital microchips like Apple I which were designed and built in 1975 by the great Steven G. Wozniak the co-founder of Apple computer and IBM 5150 which was created by a team of engineers and designers directed by the late and great Don Estridge of IBM and released on August 12, 1981, I was always fascinated by the way computers do their "magic" of computing, like what happens when a key press appears as a character on the computer screen. I started learning the mystery of computer hardware by the book "Troubleshooting and Repairing Personal Computers" written by Art Margolis who has done a fantastic job of explaining the inner workings of TRS-80 and Z80 microprocessor and its peripherals and "IBM PC advanced Troubleshooting and Repair" by Robert C. Brenner - a naval officer who was a professor at the Naval Post Graduate School in Monterey CA. - not knowing that the city of Monterey is going to be my home for almost 15 years of my life later on, I had a very humble beginning and always excited about the way computers do their magic.

Of course assembly language was very tied up with hardware and naturally my next step would be learning the assembly programming for different chips like Z80, Intel 8085, 80x86 (back when I was in high school 80286 was just becoming a standard). It was then, when I learned the machine code and the way assembly mnemonics represent the machine code and make the tedious task of hand assembling the machine code and entering the code into the processor using micro switches much easier. Of course getting Intel microprocessors was not easy for me as they were not cheap but I could get my hands on Z80 and experiment with it on a breadboard.

1)https://albert.ias.edu/bitstream/handle/20.500.12111/7941/Booth_Britten_Coding_for_ARC_1947.pdf?sequence=1&isAllowed=y

2)https://en.wikipedia.org/wiki/Assembly_language#cite_note-Wilkes_1951-9

3)[https://en.wikipedia.org/wiki/David_Wheeler_\(computer_scientist\)](https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist))

Next step was experimenting with programming languages, as I needed to program the computer. DOS operating system was the standard which shipped with QBASIC and Windows was in its infancy. (Windows 95 was not born yet)

I was learning the concept of interpreter using QBASIC. It was slow due to the fact that it was an interpreter. The code was translated to the machine code one line at a time and then the next line was read and translated and so on. I then experimenting with Borland Pascal and C and it was getting more and more exciting.

But, I always had this question in my mind, “I learned about the hardware and now it makes sense how the a character shows up on the screen, but how do the compilers create the code which does that?”

This was the question I had in my mind for a long time, but unfortunately it was not going to be answered any time soon as life took me away in other journeys and I got far away from what I truly loved and wanted to know.

I can honestly say that the Compiler Design class I took this quarter answered many of those questions and opened up a really interesting chapter of my learning history. I finally learned how those symbols and codes turned into machine code.

C, Pascal and QBasic were looking totally different from Assembly mnemonics representing the machine code and up to the beginning of this class still it was a mystery how the source code of those programs are finally turned into the machine code that is ultimately run by the processor which only understands its own ISA.

Now I understand what lexers and parsers are, and apart from that since processors have many advance components in their newer implementations, x86 family of processors are not working the same way as their ancestors like 8086 and 286 worked. Now they have pipelines, they have large caches the have an implementation totally different than that of their grandparents. There is instruction scheduling and out-of-order instruction execution and the world of computing is much more advanced and exciting that those early days, although still those old processors and compilers are nostalgic and tell us a humble story of how it was when it all started.

Many thanks to Professor Sorensen for offering the Compiler Design class, I consider the knowledge I learned in this class as a remarkable turning point in my adventures in learning about computer hardware. What I always thought of as the “magic” is logic to me now.

In this class we covered a lot of advanced topics which the project I am going to represent is only a tiny small part of all the knowledge which was taught in the class and I learned a lot specially the up to date advanced tricks of the trade about different kinds of modern processors which are much more sophisticated than those early processors even though the fundamentals are the same as it applies to every part of human science.

Summary of the Project

For my final project, I chose a mini BASIC interpreter. This project is implemented in two ways. First is in C, which the implementation is not mine but the code is adopted from a C programming book that in one chapter the author is giving an example of implementation of a small BASIC interpreter done in C language.

The original implementation is done in Borland C, which should compile under Borland C version 2. I tried to compile the code in the above mentioned compiler and with some small modifications, the code compiled successfully. The details of the modifications will follow in the rest of this report. The author has also given an example of BASIC code which runs under the implemented interpreter. Dr. Einollah Jafarnezhad Qomi has done a fantastic job in that book as an advanced C programming tutorial, which has been and maybe is one of the de-facto C programming bibles taught in many universities throughout Iran. I personally would highly recommend this book for Farsi speakers interested in learning C programming language, of course along with the great K&R bible which I think is one of the most fantastic books ever written in computer science history.

I tried compiling the code under Borland C first to test it and make sure the code is working. In order to do that, I had to install DOS operating system in VirtualBox under Linux, but since I worried about rapidly getting this done, I chose DOSBOX as I could modify the code in an editor in Linux and run the rescan command inside DOSBOX for the changes to take effect and then run the compiler against the modified code. After some small modifications which will be explained later, the code worked as a compiled binary running under DOS that could take the sample BASIC code and interpret and run it under DOS.

Since the code is written specifically with Borland C in mind, Dr. Sorensen advised me to port the code to modern compilers like GCC, since in that case the new generation of learner community can benefit from it as modern compilers like GCC are the tools of the trade today and also GCC is the compiler that is used in compiling Linux kernel and used on a daily basis by millions of programmers around the world.

Since many common standard C libraries were used in the original code, the only major library needed was the "CONIO" library of Borland C, which after good amount of searching the internet, I ran across a couple of the and after testing them, I picked one that was best documented and worked with no problem. This library uses ncurses library which needs to be installed in Linux for this code to work. I will publish the link to its git-hub repository in the documentation of the code and many thanks to the author of the code "Nowres Rafid" for this wonderful contribution to the community.

The original code implementation does not utilized any modern compiler generators tools and the whole thing has been done in C, but still the learner can notice that a great deal of effort has been made to get a small basic interpreter implemented, which can be prone to bugs and harder to maintain and harder to understand for beginners and also harder to expand.

I took the final syntax and keywords which the author had aimed for in the original code and made the same BASIC interpreter using PLY, as a means of comparison of the implementation.

Hoping that the community can see both the old school way of interpreter implementation and compare that with the modern way of doing it using PLY type of tools, which thanks to Dr. David Beasley, it's available to learner community as a starting point for learning Lex and Yacc tools. This project is also an example of the portability power of C language as it was designed with this aspect in mind. I will explain the three parts of this project in three separate sections:

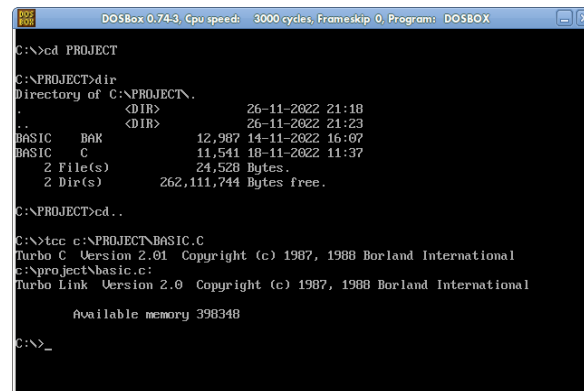
Part I.

In order to compile the code under Borland C version 2, the package was downloaded in disk image files from winworldpc.com. The images were deflated in DOSBOX using its imgmount tool and the content of the image files were copied from A: drive to C: drive in DOSBOX one by one to create the installation directory. Turbo C comes with its tools TCC and TLINK as the compiler and the linker and also the editor, which I chose not to use the editor as the modern editors available in Linux were much easier to use.

The code was typed up in a file named basic.c in Linux and copied in a directory inside DOSBOX and after every modification, rescan command was issued in DOSBOX for the saved changes to take effect inside DOSBOX for compilation. One thing to consider is since DOS files use Carriage Return/ Line Feed combination at the end of each line in ASCII text files, the typed up code had to be converted using unix2dos utility in Linux in order for Turbo C compiler to read the source code without a problem. After that the text editor in Linux would keep the CR/LF combination at the end of each line to avoid the problem. I used Geany editor to type up the code.

The complete code listing can be found at the end of this report and also can be uploaded to github for the community to use. The change that was needed for the code to work was to move the structure for_stack to the top of the function prototypes after #define lines. For some reason the author had put the structure after the function prototypes which were using it causing the error.

Below is a screenshot of compiled code along with the working BASIC code under the interpreter.



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip: 0, Program: DOSBOX

C:\>cd PROJECT

C:\PROJECT>dir
Directory of C:\PROJECT\
.                <DIR>                26-11-2022 21:18
..               <DIR>                26-11-2022 21:23
BASIC   BAK      12,987 14-11-2022 16:07
BASIC   C         11,541 18-11-2022 11:37
       Z File(s)      24,528 Bytes.
       Z Dir(s)       262,111,744 Bytes free.

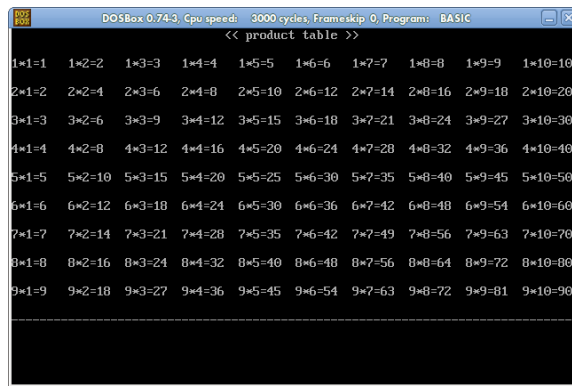
C:\PROJECT>cd..

C:\>tcc c:\PROJECT\BASIC.C
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
c:\project\basic.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

        Available memory 398348

C:\>_
```

Figure 1. Code compiled successfully under Turbo C 2.0 in DOSBOX.



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: BASIC

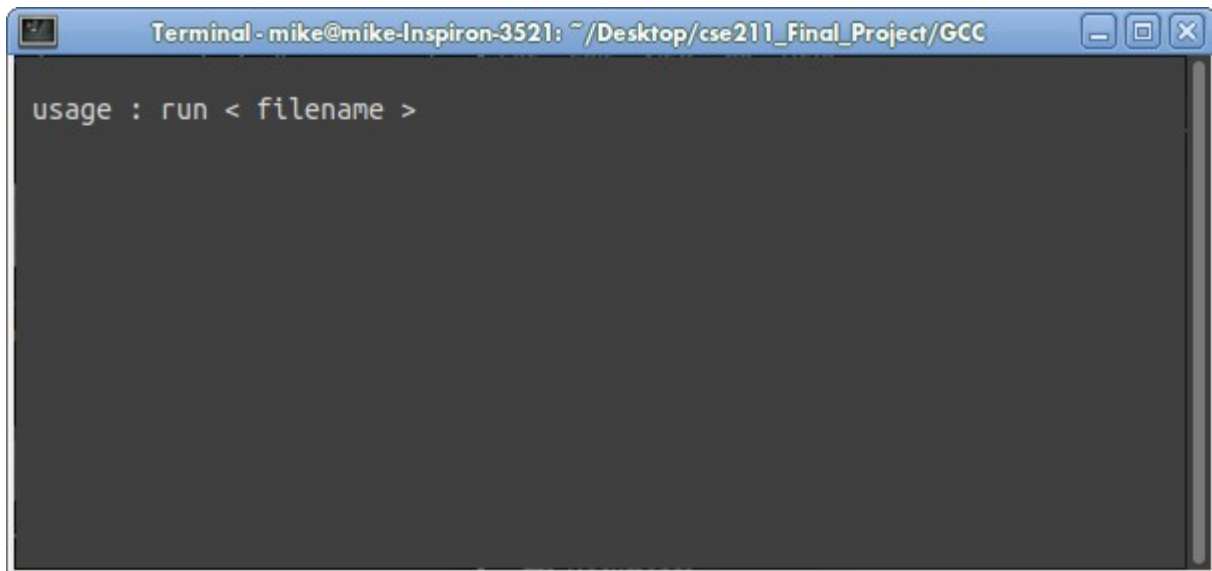
```
<< product table >>
```

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9	1*10=10
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18	2*10=20
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27	3*10=30
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36	4*10=40
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45	5*10=50
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54	6*10=60
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63	7*10=70
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72	8*10=80
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81	9*10=90

Figure 2. A sample run of an example BASIC code calculating and printing the 9 by 9 multiplication table in DOSBOX

Part II.

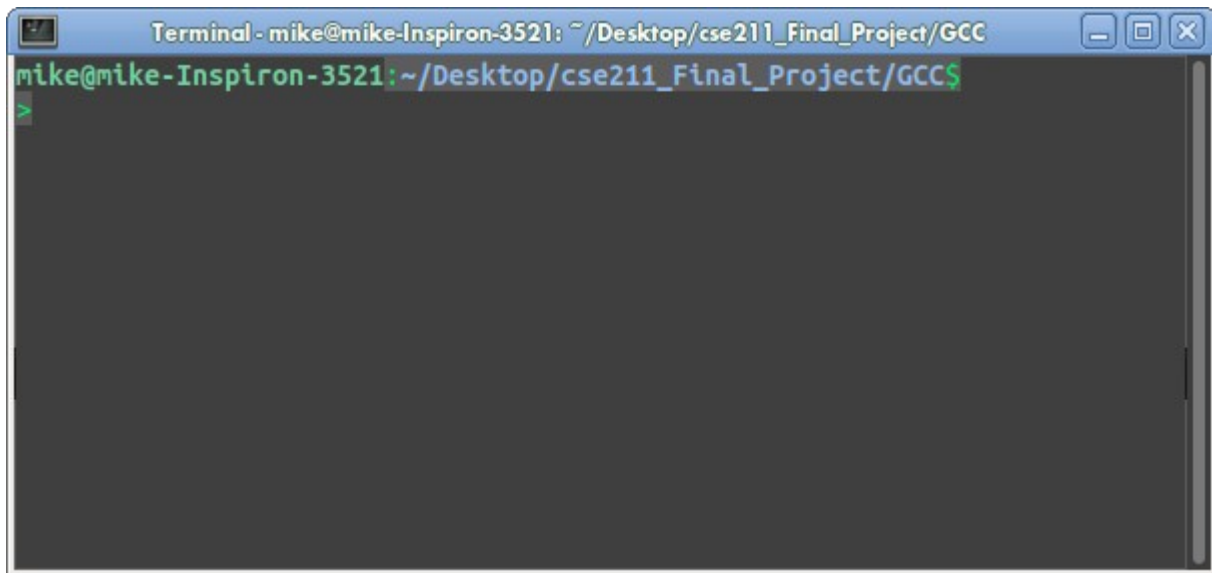
In this part I tried to port the C code to modern GCC, so modifications are done so that the code can compile under GCC and run in Linux. As mentioned above the “conio.h” library has been utilized and the changes have been made accordingly. Some functions like printf have been replaced with cprintf from conio.h implementation, so that it does not conflict with ncurses library.



Terminal - mike@mike-Inspiron-3521: ~/Desktop/cse211_Final_Project/GCC

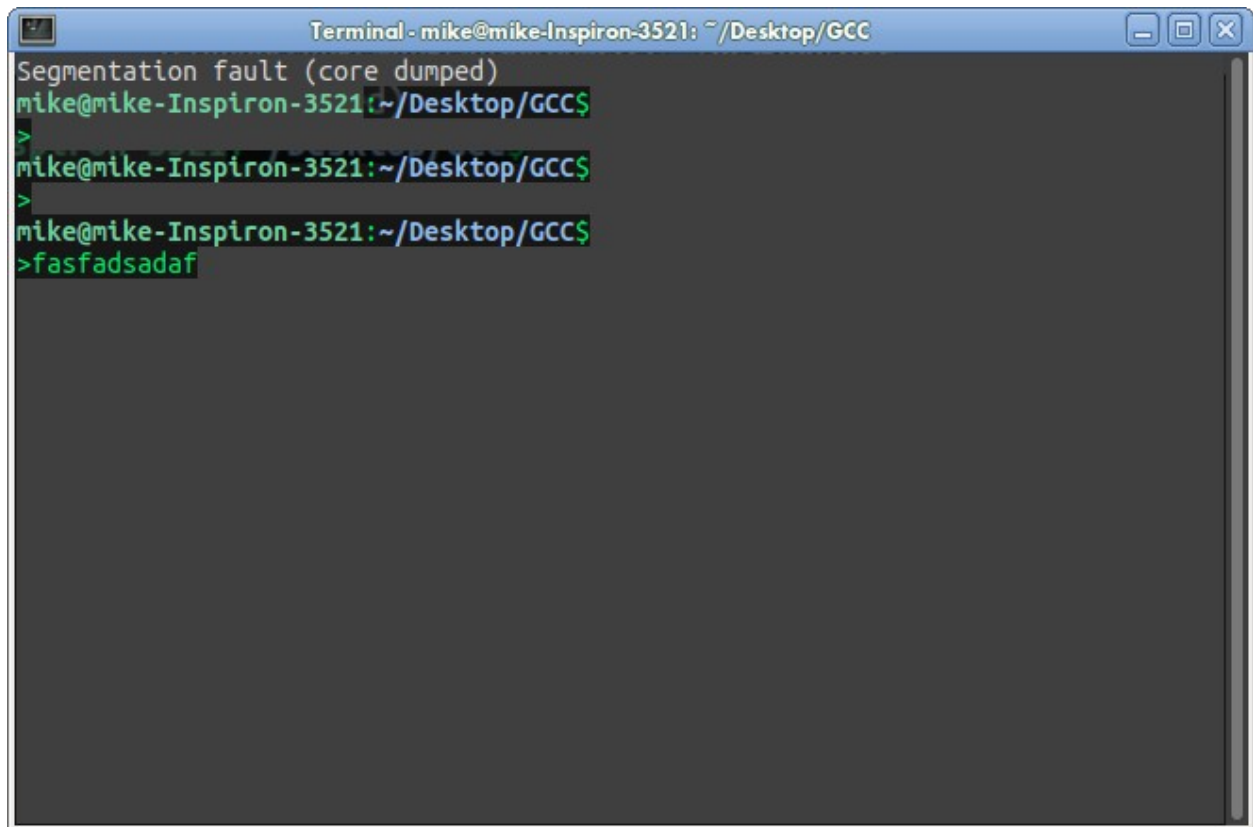
```
usage : run < filename >
```

Figure 3. Code compiled successfully under GCC in Linux Mint.

A terminal window titled "Terminal - mike@mike-Inspiron-3521: ~/Desktop/cse211_Final_Project/GCC". The prompt is "mike@mike-Inspiron-3521:~/Desktop/cse211_Final_Project/GCC\$". A green cursor is visible on the line below the prompt.

```
Terminal - mike@mike-Inspiron-3521: ~/Desktop/cse211_Final_Project/GCC
mike@mike-Inspiron-3521:~/Desktop/cse211_Final_Project/GCC$
>
```

Figure 4. A sample run of an example BASIC code calculating and printing the multiplication table in Linux (note unfortunately the code messes up the terminal screen which needs to be fixed later, there is also segmentation fault error given by Linux when running this program, I will need to fix this in the future. The code is on github.com for community to help)

A terminal window titled "Terminal - mike@mike-Inspiron-3521: ~/Desktop/GCC". The prompt is "mike@mike-Inspiron-3521:~/Desktop/GCC\$". The output shows a "Segmentation fault (core dumped)" message, followed by three prompts and a command. A green cursor is visible on the line below the last prompt.

```
Terminal - mike@mike-Inspiron-3521: ~/Desktop/GCC
Segmentation fault (core dumped)
mike@mike-Inspiron-3521:~/Desktop/GCC$
>
mike@mike-Inspiron-3521:~/Desktop/GCC$
>
mike@mike-Inspiron-3521:~/Desktop/GCC$
>fasfadsadaf
```

Figure 5. Segmentation fault that I couldn't fix (but the terminal is working this time).
Very strange!!

Part III.

In this part I have tried to use PLY library with Python to make the BASIC interpreter. Most of the keywords and symbols are those used in the original Turbo C code to keep the BASIC interpreter as close as possible to the original.

Explanation of the BASIC interpreter implemented in C code:

The expressions in this BASIC language contain integers, operations (+ - / * ^ = () <> ; and ,), parentheses and variables. The ^ operator is used to exponentiation and = sign is used for assignment and equality. Expressions can be combined algebraically and create other expressions. The comparison operators (= < >) and , and ; which can be considered operators are used in the IF and PRINT statements. The order of precedence is standard with the highest priority from left to right (), ^, *, /, %, +, -, = and operators with the same precedence will have precedence based on their position in the expression. The one that appears earlier gets processed earlier from left to right.

Variable names are only one character (a-zA-z) and numerical variables are all integers. There are no String variables in this language and the end of the program is marked with “NULL” character.

The tokenizer function should do the following operations. It should ignore blank and tab characters. It should extract each token, if possible it should convert the token to the internal format and it should determine the type of the token.

Each token has an internal and an external format. The external format is the one that is used in the program code, “the BASIC source code”. For example PRINT is the external format of the PRINT statement. In internal format, each keyword is determined by a number for example the number representing PRINT is 1, for INPUT it is 2 and so on. They are defined in the beginning of the code. This speeds up the interpreter as it is dealing with integers instead of strings. The interpreter uses 6 different kinds of data: DELIMITER for operators and parentheses, VARIABLE for variables, NUMBER for numbers, COMMAND for BASIC commands, STRING as an auxiliary variable until the token is completed and QUOTE for strings that are inside quotations.

Explanation of the functions

The following is explanation of the functions as appeared in the book and also explanations that I could reverse engineer.

get_token() deciphers the tokens inside an expression. This function uses other functions: **iswhite()**, **serror()**, **isdelim()** and **loopup()** but before explaining them let's focus on **get_token()** function first.

There are four important variables in this function which are “prog”, “tok”, “token” and “token_type”. The variable “prog” points at the next character to be read from the file. The

variable “tok” stores token’s internal format, the variable “token” contains the token itself and “toke_type” contains the type of the token. For example the result of `get_token()` on an expression like `PRINT A + 100 - (B * C / 2)` would be as follows:

Token	Token type
PRINT	COMMAND
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER
(DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
/	DELIMITER
2	NUMBER
)	DELIMITER
NULL	DELIMITER

iswhite() takes a character as a parameter and if this parameter is the white space character or tab character it will return 1 otherwise 0 to `get_token()`.

seerror() is used for issuing a syntax error message.

isdelim() takes a character as input. It returns 1, if this character is a DELIMITER or 0 otherwise back to `get_token()` .

look_up() searches for a variable in a table which stores the variables. If this variable is found in the table returns its internal format, otherwise returns 0 to `get_token()` .

Explanation of Other Functions:

get_exp() determines the existence of an expression and calls other functions

The next 5 functions are production functions (for production actions) as we called in our class.

level2() does addition and subtraction operations on two **factors**.

level3() does multiplication and division operations on two **factors**.

level4() does the exponentiation operation on integers.

level5() determines the unary operators + and - and their effect on the expression.

level6() processes the expressions inside parentheses.

primitive() finds out the value of integers and variables.

arith() does the arithmetic operations.

unary() changes the sign from + to - and vice versa.

find_var() determines the value of a variable from a look up table.

putback() returns the last read token back to the input expression. Some functions in the interpreter need prediction of a token for determining the next operation. In many cases, if procedures need tokens, tokens should be returned back to the input string.

look_up() determines the internal format of tokens by looking them up in the lookup table.

load_program() loads the BASIC source code into memory.

There is the variable table used that the analyzer uses to find variables and their values, which the values are initially set to 0.

Since only one-character variables are evaluated in the analyzer, in order to find a variable in the table, the only thing to do is to subtract the ASCII code of the variable name from the ASCII code of letter A (65).

Keywords used in this language are “PRINT, INPUT, IF, THEN, FOR, NEXT, TO, GOTO, GOSUB, RETURN, END”

In #define lines of the C code EOL is for end of line and FINISHED is to determine the end of file.

Since in order to convert external format of tokens to internal format, there is a need for external format of the tokens, the internal and external formats are saved in the table named “table” as in the following structure:

```
struct commands {  
    char commands[20];  
    char tok;  
} table[]=  
{  
    "print", PRINT, "input", INPUT, "if", IF,  
    "then", THEN, "goto", GOTO, "for", FOR,  
    "next", NEXT, "to", TO, "gosub", GOSUB,  
    "return", RETURN, "end", END, "", END  
};
```

Figure 3. Internal and external format table

The interpreter operates this way: It reads the next token of the expression, then it invokes a suitable function to process that token, in the main iterating loop. The main loop of the BASIC interpreter looks as follows:

```
do {
    token_type = get_token();
    if(token_type == VARIABLE) {
        putback();
        assignment();
    }
    else
    {
        switch(tok) {
            case PRINT : print(); break;
            case GOTO : exec_goto(); break;
            case IF : exec_if(); break;
            case FOR : exec_for(); break;
            case NEXT : next(); break;
            case INPUT : input(); break;
            case GOSUB : gosub(); break;
            case RETURN : greturn(); break;
            case END : exit(0);
        }
    }
}while (tok!=FINISHED);
```

Figure 4. The main loop of the interpreter

The way the main iterating loop operates is that the first token is read from each line of the BASIC program. Assuming that there is no syntax error. Now either this token is a variable which should be assigned a value, (the BASIC keyword LET is not used in this language) or it is a statement for which a suitable function should be invoked to execute that statement.

Assignment Statements

In BASIC in order to assign value to a variable, this kind of relation is used:

variable name = expression

The function that does the assignment is **assignment()** which reads a token (this token is a variable which a value should be assigned to it) and then it will read “=” symbol. If the previously read token is not a variable or the second token is not “=” an error is issued. After reading “=”, the function **get_exp()** is invoked to assign the value to the variable.

The PRINT statement

This interpreter has a simple form of **PRINT** command: **PRINT <arg>** which is simpler than

most flavors of BASIC which have more complicated form like having the keyword **USING** to format the output. “**arg**” includes variables or strings separated from each other by comma or semicolon. This is done in **print()** function.

INPUT command

INPUT command is used in order to read the input information which is used in two ways:

INPUT <var> or **INPUT “prompt”, <var>**

In both methods, **var** is the variable name which is supposed to be read from input, and **prompt** is the message that is to be appeared on the screen while getting the input from the user.

The **INPUT** command is implemented in **input()** function.

GOTO command

The **GOTO** statement is used in order to transfer the control from one point of program to another point. The syntax is as follows:

GOTO <label>

The **label** is the line number to which the control flow should be transferred. The problem in executing **GOTO** statement is that the control flow should be transferable from the beginning of the program to the end and vice versa. In order to do this, before execution of the program, labels in the program should be determined and their name along with pointers to their location should be kept in a table. The table which keeps the labels and their corresponding addresses is defined in the following structure:

```
struct label{
    char name[LAB_LEN];
    char *p;
};
struct label label_table[NUM_LAB];
```

Figure 5. Labels and their addresses

The **scan_label()** function puts the labels and their addresses in the **label** table. This function initializes the **label** table using the function **label_init()** . It invokes functions **get_token()** to get the next token and **find_eol()** to find the end of line. If there are duplicate labels in the program or the **label** table is not filled, an error message will be issued.

exec_goto() function is used for execution of **GOTO**. The function **find_label()** searches the table for a label. In case it finds the label, the pointer to that label is put in the variable **prog** otherwise **NULL** is returned and an error message will be issued.

IF command

The if command is used for conditional control flow and the following for is used in the interpreter.

IF <expression> <operator> <expression> THEN <command>

The operators that are investigated in the **IF** command are: = < and >

The function **exec_if()** executes the **IF** command. This function operates this way:

- 1- The left hand side expression is evaluated.
- 2- The operator is read.
- 3- The right hand side expression is evaluated.
- 4- The condition gets tested.
- 5- If the condition holds, the next command after **THEN** is executed, otherwise **find_eol()** function is invoked to find the next line.

FOR command

For command is used to repeat a collection of statements and its format is as follows:

FOR counter = initial_value TO final_value

.
.
.

NEXT

The standard **STEP** keyword in BASIC is not used in this language. FOR loops can be nested which makes keeping the information in case of each FOR command necessary which needs using a stack.

In the beginning of the **FOR** loop, information like the initial and final state of the counter and the beginning of the address of the **FOR** loop are kept in the stack. After encountering **NEXT**, these values are popped off the stack, the stack is reinitialized and compared with the final value, in case the current value is more than the final value, the loop ends and the control returns to the statement after **NEXT**, otherwise the new value is being put to stack and the execution of the following statements, resume from the beginning of the loop. This trend is also suitable for nested **FOR** loops.

The stack needed for execution of **FOR** command is defined in the **for_stack** structure and the functions **fpush()** and **fpop()** are used for manipulating this stack. The variable **FOR_NEST** keeps the number of nested **FOR** loops. **GOTO** can be used to exit **FOR** loops but it is error prone.

```

struct for_stack{
    int var;
    int target;
    char * loc;
}fstack[FOR_NEST];

int ftos;

void fpush(struct for_stack i)
{
    if (ftos > FOR_NEST)
        serror(10);
    fstack[ftos] = i;
    ftos++;
}

struct for_stack fpop()
{
    ftos--;
    if(ftos < 0) serror(11);
    return(fstack[ftos]);
}

```

Figure 6. Stack structures for FOR loops and nested FOR loops

GOSUB command

GOSUB is used to execute the sub-programs and its form is as follows:

GOSUB <line>

.
.
.

RETURN

Like **FOR** command, **GOSUB** command also needs a stack. This stack is defined as follows. The variable **SUB_NEST** defines the number of nested sub-programs:

```

char *gstack [SUB_NEST];
int gtos;

```

Figure 7. stack defined for sub-programs

The “Production Rules” rules :)

In order to analyze and evaluate the expression, a recursive structure has been created. Assuming that expressions will only use * / + - and () the following table is determined as the Production Rule for the parser. * and / have precedence over + and -.

Symbol	Production Name	Production Rule
+, -	Expression	: Term Term + Term Term - Term
*, /	Term	: Factor Factor * Factor Factor / Factor
Numbers Variables	Factor	: Var NUM Expression

As mentioned before the code listing for both old code and the ported version will be made available through git-hub for free public use.

Conclusion:

Comparing the amount of code needed to implement a BASIC interpreter purely in C programming language, with almost the same thing implemented in Python utilizing PLY, clearly shows, the amount of work and code and the efficiency and the flexibility of the modern tools, and the power and elegance of the end results which is also expandable and flexible with a lot of room to improve for the future needs.

Although many parts of this project proved to be a stretch, and I was not able to finish them completely, I hope with the code being released to the public, others would learn from it and expand on it, with an appreciation of the great changes which modern tools have provided us.

The GCC part of project is still not functional, as every time, I changed something a new strange thing happened. I managed to fix the segmentation fault but now when the basic source code is fed into the interpreter, it exits. But well with time I had I couldn't finish it, but the code is there on git-hub for me and the community to fix it and share it.

For the python part of the project, again I couldn't finish it, but the goal, as can be seen in the code, is to follow the philosophy of interpreters – read one line of the code, execute it and go for the next line to the end – of course there should be mechanisms for scanning the whole source code for labels for control flow. But as it can be seen in the code, the parser and lexer are organized in two files and the main part of the program reads the source code line by line and shows the lexems for the the input code line by line. It can be clearly seen that the amount of code for implementing the same thing and also the organization of the code and the its ability to

expand is very visible and any body working with PLY can see how easy it is to create really complicated languages using PLY as I've seen on the internet, there are Pascal implementations done with PLY.

For me, learning PLY is a great introduction to Lex and Yacc and Dr. Beasley has also recommend the Lex and Yacc book by O'Reilly, which I have and will use as a reference.

It would also be tricky to implement the for loop structure, that I think would require a stack mechanism to keep track of statements inside the loop and also breaking out of the loop would be another tricky thing that would require more thought later on to expand on the interpreter.

One other thing that I would suggest about the expansion of the interpreter is to add graphics capabilities to the language as graphics are a big part of most of basic interpreters I have worked on before and I would suggest, it can be done with John Zelle's graphics module which can be found here: <https://github.com/drypycode/zelle-python>

This tool is also a training tool for learners and a really fun module to work with and it is mainly targeting learners, almost the way PLY has been implemented. It's a fantastic library that makes working with graphics much easier than other serious larger projects like PyOpenGL which are all great tools for doing almost any kind of computer "magic".

I hope this project will be beneficial for the community and fun to work on as I really learned a lot from it and will work on it every chance I get. One thing that I will try in the future is utilizing debuggers like GDB and its graphical front end DDD and also I would recommend debugging the C code in Code::Blocks to hopefully fix the bugs in the GCC port.

My code and this documentation is published under GPL license and I gladly grant all the necessary permissions to whoever interested in this project. Unfortunately I don't have any address from Dr. Jafarnezhad – which I'm not sure if he is still active in academia but hereby I would like to thank him for his great book and Dr. Sorensen for his fantastic fun class and for many many things he taught us about advanced technology of compilers and computer architecture in class of Fall of 2022. A really fun class that created lots of fun memories for me for life.

The code to the project can be found here: https://github.com/MikeAnj/cse211_Final_Project

Mike (Mahyar) Anjomani
December 8th of 2022