

# INTERPRETER DESIGN AND IMPLEMENTATION

Rajarshi Bhattacharya

Department of Computer Science,  
St. Xavier's College(Autonomous), Kolkata  
rbkumar712@gmail.com

**Abstract**—In this paper, the design and implementation of an interpreter is discussed using software engineering concepts and C language. Some of its components form the basis for different engineering tools. This paper demonstrates the basic implementation of tokenizer, scanner, parser, abstract syntax tree, AST interpreter etc., which can provide a useful track for the evolution of an interpreter.

**Keywords**— *Tokenizer, Interpreter, AST, AST interpreter*

## I. INTRODUCTION

An interpreter translates high-level instructions into an intermediate form, which it then executes. In contrast, a compiler translates high-level instructions directly into machine language. Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long. The interpreter, on the other hand, can immediately execute high-level programs.[2]

An interpreter is just a program. As input, it takes a specification of a program in some language. As output, it produces the output of the input program. It translates code like a compiler but reads the code and immediately executes on that code, and therefore is initially faster than a compiler. Thus, interpreters are often used in software development tools as debugging tools, as they can execute a single line of code at a time. But an interpreter, although skipping the step of compilation of the entire program to start, is much slower to execute than the same program that's been completely compiled.

By designing a new interpreter, we can invent a new language. In this paper, we will explore how to implement an interpreter with C language. The language will look like the following,

```
start
x=2*3;
val=x+6;
print "x=",x," val=",val," ";
print "val*3=",val*3," val*x=",val*x;
end
```

Here, “start”, “end” and “print” are keywords, “x”, “val” are variables and the interpretation of the program is same as any standard algorithm or language.

The paper is organized as follows: section II presents the background and related work, and section III describes the design and development process. The conclusions and future work are discussed in section IV.

## II. BACKGROUND AND RELATED WORK

### A. Background

The main purpose of a compiler or an interpreter is to translate a source program written in a high-level source language to machine language. The language used to write the compiler or interpreter is called implementation language (Here, it is C). The difference between a compiler and an interpreter is that a compiler generates object code written in the machine language and the interpreter executes the instructions. A utility program called a linker combines the contents of one or more object files along with any needed runtime library routines into a single object program that the computer can load and execute. An interpreter does not generate an object program. When you feed a source program into an interpreter, it takes over to check and execute the program. Since the interpreter is in control when it is executing the source program, when it encounters an error it can stop and display a message containing the line number of the offending statement and the name of the variable. It can even prompt the user for some corrective action before resuming execution of the program. There are several types of interpreters that can be designed for a language such as,

- Bytecode interpreters
- Threaded code interpreters
- Just-in-time compilers(hybrid compiler)
- Abstract Syntax Tree(AST) interpreters

In this paper an AST interpreter is designed for the language stated previously.

The interpretation process can mainly be divided into 3 functional increments. Before moving to the next increment, the current increment has to be tested and validated. The increments are:

1. Scanning or tokenizing
2. Parsing(which is basically forming the AST)
3. Interpreting the AST

Interpreters are complex programs, and writing them successfully is hard work. To tackle the complexity, a strong software engineering approach can be used. Design patterns, Unified Modeling Languages (UML) diagrams, and other modern object-oriented design practices make the code

understandable and manageable. But, in this paper it will not be covered as this is for a very small interpreter.

### B. Related Work

Interpreters were used as early as 1952 to ease programming within the limitations of computers at the time (e.g. a shortage of program storage space, or no native support for floating point numbers). Interpreters were also used to translate between low-level machine languages, allowing code to be written for machines that were still under construction and tested on computers that already existed. The first interpreted high-level language was Lisp. Lisp was first implemented in 1958 by Steve Russel on an IBM 704 computer. Russell had read John McCarthy's paper, and realized (to McCarthy's surprise) that the Lisp eval function could be implemented in machine code. The result was a working Lisp interpreter which could be used to run Lisp programs, or more properly, "evaluate Lisp expressions".[3]

While the area of interpreter design, as a subset of compiler design is well-established and documented, it is not typically the subject of formalized software engineering concepts.[1]

## III. DESIGN AND IMPLEMENTATION

In the spectrum between interpreting and compiling, one of the approaches is to transform the source code into an optimized Abstract Syntax Tree or AST, then execute the program following that tree structure, or use it to generate native code just-in-time. In this approach, each sentence needs to be parsed just once. As an advantage over bytecode, the AST keeps the global program structure and relations between statements (which is lost in a bytecode representation), and when compressed provides a more compact representation. Thus, using AST has been proposed as a better intermediate format for just-in-time compilers than bytecode. Also, it allows the system to perform better analysis during runtime.

However, for interpreters, an AST causes more overhead than a bytecode interpreter, because of nodes related to syntax performing no useful work, of a less sequential representation (requiring traversal of more pointers) and of overhead visiting the tree.

The basic design of the interpreter will be like the following,

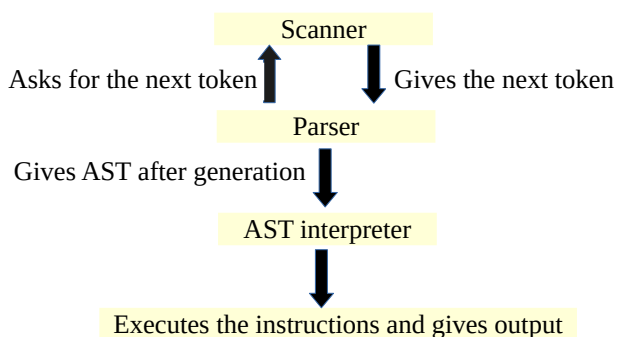


FIGURE-I: DESIGN THE THE INTERPRETER

### A. Syntax and Semantics

The syntax of a programming language is its set of grammar rules that determine whether a statement or an

expression is correctly written in that language. The language's semantics give meaning to a statement or an expression.

Like, in C, the statement:

$a = b + c;$

is a valid assignment statement. The semantics of the language tells that the statement says to add the value of variables 'b' and 'c' and assign the sum's value to the variable 'a'. A parser performs actions based on both the source language's syntax and semantics. Scanning the source program and extracting tokens are syntactic actions. Looking for '=' token is a syntactic action, entering the identifiers 'a', 'b', and 'c' into the symbol table as variables, or looking them up in the symbol table, are semantic actions because the parser had to understand the meaning of the expression and the assignment to know that it needs to use the symbol table. Syntactic actions occur in the front end, while semantic actions can occur on either the front end or the back end.[1]

One of the example for the language, which is going to be designed here, was given before. So, basically the language contains 3 keywords,

1. print
2. start
3. end

Also, the language will support basic arithmetic operators like '+', '-', '\*', and '/'. Also, it will support strings, left parenthesis('('), right parenthesis(')'), comma(','), semicolon(';') and assignment('=').

The grammar of the language is simple as following,

- Every program must start with a "start" keyword and end with "end" keyword and in between these two keywords the main body of the program must take place.
- The body of the program will contain list of statements which can be either a print statement or declaration of variables and every statement must end with a semicolon.
- Only int type variables are supported.
- Multiple declarations of variables in a single line is not allowed.
- The print statement may contain a string or a variable which is declared before or an expression or a string with an expression or variable. Each of these must be separated by comma.
- The newlines and spaces will be ignored except for print keyword where, there must be a space after the print keyword and the contents to be printed.

### B. Design of the scanner

Scanner performs lexical analysis or tokenization, which is the process of converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning). A lexer is generally combined with a parser, which together analyze the syntax of programming languages.[3]

As stated before, the scanner will tokenize the whole program into the following tokens.

1. ID: This contains all the “print”, “start”, “end” keywords and the variable names
2. LPAREN: Which is left parathesis.
3. RPAREN: This is the right parathesis.
4. INT: This is for simple integers.
5. SUM: This is for ‘+’.
6. SUB: This is for ‘-’.
7. MULT: This is for ‘\*’.
8. DIV: This is for ‘/’.
9. START: This is for the “start” keyword.
10. END: This is for the “end” keyword.
11. ASSIGN: This is for ‘=’.
12. SEMI: This is for ‘;’, which will indicate the end of a statement.
13. PRINT: This is for the start of print statements.
14. STRING: This is for strings.
15. COMMA: This is for ‘,’. This will distinguish between a print statement and variables and expressions.
16. EXPR: This is any expression. This token will help to determine if a variable or print statement contains an expression or simple integer which will be helpful for the interpreter.

These tokens will be declared in the global scope as integers using preprocessor directives or enum.

So, now for converting the stream of characters to the tokens, if-else or switch statements can be used. A portion of the scanner will typically look like the following,

```
Token scanner(char *str,char ch){
    Token token;
    while(ch== ' '||ch=='\n'){
        ch=str[++pos];
    }
    if(ch==';'){
        token.type=SEMI;
        token.value=0;
        token.id=NULL;
        return token;
    }
    if(ch=='='){
        token.type=ASSIGN;
        token.value=0;
```

```
token.id="=";
return token;
```

```
}
}
```

Here, Token is a user-defined data-structure which is like the following,

```
typedef struct token{
    int type;
    int value;
    char *id;
}Token;
```

Now, the scanner can convert the program to a stream of tokens, which can be used by the parser to generate the syntax tree.

### C. Design of the parser

A parser is a software component that takes input data (frequently text) and builds a data structure (which is AST for this paper) giving a structural representation of the input while checking for correct syntax. The parsing may be preceded or followed by other steps, or these may be combined into a single step. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Here, the parser will be programmed by hand.[3]

So, for our language the rules were given before. So, if we create our grammars for the language for the given rules then we can get the following set of rules:

- program : START compound\_statements END
- compound\_statements : statement  
| statement SEMI compound\_statements
- statement : print\_statement SEMI  
| assignment\_statement SEMI  
| empty
- print\_statement: PRINT STRING  
| PRINT STRING COMMA EXPR  
| PRINT STRING COMMA variable
- assignment\_statement : variable ASSIGN expr
- empty :
- expr: term ((PLUS | MINUS) term)\*
- term: factor ((MUL | DIV) factor)\*
- factor : PLUS factor  
| MINUS factor  
| INTEGER  
| LPAREN expr RPAREN  
| variable
- variable: ID

Now, we can create functions for each of the rules stated above and make the program generate the AST. A small portion for the parser is given below:

- For program:

```
Compound* program
(char *str,Compound *root,Token current_token){
    match(current_token.type,START);
    current_token=scanner(str,str[pos]);pos++;
    root=compound(str,NULL,current_token);
    current_token=scanner(str,str[pos]);
    match(current_token.type,END);
    return root;
```

//match() function is for matching the tokens

- For compound\_statements:

```
Compound* compound
(char *str,Compound *root,Token current_token){
    root=(Compound*)malloc(sizeof(Compound));
    root=set_comp(root,NULL,
        statement(str,NULL,current_token));
    Compound *temp=root;
    int rev_pos=pos;
    current_token=scanner(str,str[pos]);
    while(current_token.type==SEMI){
        match(current_token.type,SEMI);
        rev_pos=pos;
        current_token=scanner(str,str[pos]);pos++;
        if (current_token.type!=ID
            && current_token.type!=PRINT)
            break;
```

```
temp->child= set_comp((Compound*)
    malloc(sizeof(Compound)),NULL,
    statement(str,NULL,current_token));
temp=temp->child;
current_token=scanner(str,str[pos]);
}
pos=rev_pos;
return root;
```

//set\_comp() is for setting a Compound structure

//statement() is for matching statements and creating them  
//as nodes in the AST

- For statement:

```
Node* statement
(char *str,Node *root,Token current_token){
    if(current_token.type==ID){
        root=assign(str,root,current_token);
        g_size++;
    }
    else if(current_token.type==PRINT){
        root=leaf(NULL,current_token);pos++;
        Node *temp=root;
        int rev_pos=pos;
        current_token=scanner(str,str[pos]);
        if(current_token.type==ID||
            current_token.type==INT){
            temp->right=
                (Node*)malloc(sizeof(Node));
            temp=temp->right;
            (temp->item).type=EXPR;
            temp->left=
                expr(str,NULL,current_token);
            temp->right=NULL;
        }
        else{
            pos++;
            temp->right=
                leaf(NULL,current_token);
            temp=temp->right;
        }
        current_token=scanner(str,str[pos]);
        while(current_token.type!=SEMI){
```

```
match(current_token.type,COMMA);
current_token=scanner(str,str[pos]);
if(current_token.type==ID||current_token.type==INT){
    temp->right=(Node*)malloc(sizeof(Node));
    temp=temp->right;
    (temp->item).type=EXPR;
    temp->left=expr(str,NULL,current_token);
    temp->right=NULL;
}
else{
    temp->right=leaf(NULL,current_token);pos++;
    temp=temp->right;
```

```

}
current_token=scanner(str,str[pos]);
}
}
else
    pos++;
    return root;

//assign() is for matching assignment statements and
//creating them as nodes in the AST and leaf() is for
//creating leaf nodes in the AST

```

Here, Compound and Node are user-defined data-structures, which are like the following,

```

typedef struct compound{
    Node *item;
    struct compound *child;
}Compound;

typedef struct Node{
    Token item;
    struct Node *left,*right;
}Node;

```

These are the implementations of the 3 basic grammars given above and similarly by implementing all of the above grammars, the AST can be generated successfully.

If we consider a program like,  
start  
print "This is a parser of ", (2000+21);  
end

Then, the AST created by the parser will look like the following,

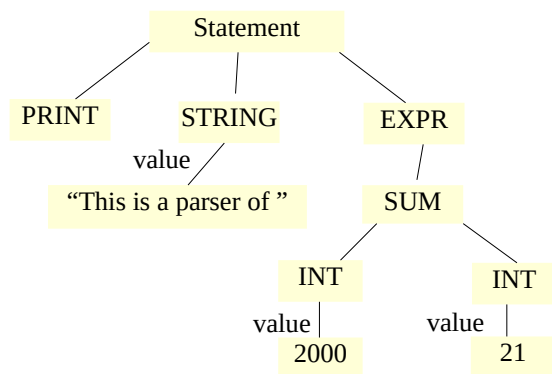


FIG-II: ABSTRACT SYNTAX TREE FOR A PROGRAM OF THE NEW LANGUAGE

Now, as the AST generation is completed it is finally time for the AST interpreter to execute the instructions stated in the AST.

#### D. Design of the AST interpreter

The final work now is to interpret the AST given by the parser. The work can be categorised in the following aspects,

1. Maintaining a Symbol Table.
2. Visiting each node of the AST.
3. Executing the instructions of the node.

a) *The Symbol Table*: The parser of a compiler or an interpreter builds and maintains a symbol table throughout the translation process as part of semantic analysis. The symbol table stores information about the source program's tokens, mostly the identifiers. The symbol table is a key component in the interface between the front and back end.

To, implement the table an user-defined data type, "Table" is defined which looks like the following,

```

typedef struct Table
{

```

```

    char *id;
    int value;

```

```

}Table;

```

For the entry and updation of the table malloc() and realloc() is used to create an dynamic array of the "Table" type. The implementation looks like the following,

```

int i=0;
while(i<size_sym){
    if(!strcmp(
        sym_table[i].id,((root->left)->item).id)){
        sym_table[i].value=
            value_expr(root->right);
        return;
    }
    i++;
}
if(!size_sym)
    sym_table=(Table*)malloc(sizeof(Table));
else
    sym_table=(Table*)realloc(
        sym_table,(i+1));
sym_table[i].id=(char*)malloc(sizeof(
    ((root->left)->item).id));
strcpy(sym_table[i].id,((root->left)->item).id);
sym_table[i].value=value_expr(root->right);
size_sym++;
//Here, size_sym is the size of the symbolic //table which
// is initially 0 and increments by 1 every time a
// new entry is added.

```

The value of an id in the symbol table is used for the expression evaluations.

b) *Visiting and Executing the nodes*: The executing of the AST instructions can be easily done by simply going through the nodes by an in-order traversal and execute accordingly. A small portion of the AST interpreter, which can be used for the expression evaluations is given below,

```

int value_expr(Node *root){
    if(root==NULL)
        return 0;
    if((root->item).type==SUB &&
        root->right==NULL)
        return -value_expr(root->left);
    if((root->item).type==SUM &&
        root->right==NULL)
        return +value_expr(root->left);
    if((root->item).type==SUM)

```

```

        return value_expr(root->left)+
                value_expr(root->right);
    if((root->item).type==SUB)
        return value_expr(root->left)-
                value_expr(root->right);
    if((root->item).type==MULT)
        return value_expr(root->left)*
                value_expr(root->right);
    if((root->item).type==DIV)
        return value_expr(root->left)/
                value_expr(root->right);
    if((root->item).type==ID){
        int i=0;
        while(i<size_sym){
            if(!strcmp(sym_table[i].id,
                        (root->item).id)){
                return
                    sym_table[i].value;
            }
            i++;
        }

        printf("%s not found!\n",sym_table[i].id );
        return '\0';
    }
    return (root->item).value;
}

```

By implementing all of the execution procedures for all of the instructions, the interpreter can be able to interpret the new language.

#### E. Error handling

All of the error handling is performed by the individual components itself and the error message will be generated whenever,

- The scanner finds an undeclared token.
- The parser finds the grammar is incorrect i.e. the parsed token is incorrect.
- The interpreter finds an undeclared identifier i.e. the entry of the identifier has not been done in the symbol table.

#### F. Execution of a programme

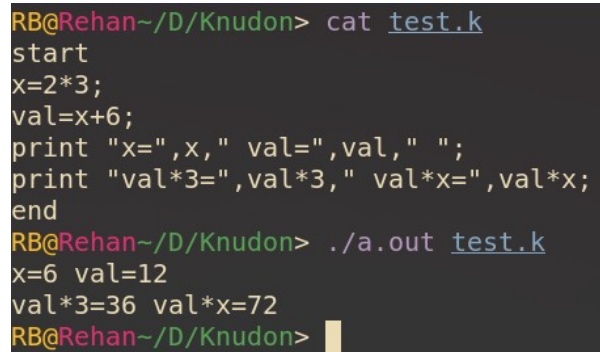
By combining all of the things together, if we execute the program stated before which is,

```

start
x=2*3;
val=x+6;
print "x=",x," val=",val," ";
print "val*3=",val*3," val*x=",val*x;
end

```

Then the output will look like the following,



```

RB@Rehan~/D/Knudon> cat test.k
start
x=2*3;
val=x+6;
print "x=",x," val=",val," ";
print "val*3=",val*3," val*x=",val*x;
end
RB@Rehan~/D/Knudon> ./a.out test.k
x=6 val=12
val*3=36 val*x=72
RB@Rehan~/D/Knudon>

```

FIG-III: OUTPUT OF THE PROGRAM INTERPRETED BY THE DESIGNED INTERPRETER

#### IV. CONCLUSION AND FUTURE WORK

In this paper, the design of an interpreter for a subset of a completely new programming language in the context of software engineering project has been presented. The paper also has demonstrated, some of the standard concepts such as scanner design, parser design, Abstract Syntax Tree, AST evaluation etc., which can provide a useful track of the evolution of an interpreter.

Future work will focus on creating a complete programming language interpreter that can perform all basic operations as a programming language. Also, object-oriented paradigms can also be included for the language. An interactive source-level debugger for the new language that enables the use of command lines to interact with the interpreter as well as an Integrated Development Environment (IDE) with a graphical user interface (GUI) can also be designed. If time is not a constraint, the interpreter will be extended to a compiler that generates object code for a Virtual Machine (VM). Then the compiled programs will then be able to run on multiple platforms.

#### ACKNOWLEDGMENT

I would like to thank Professor Dr. Anal Acharya from St. Xavier's College(Kolkata) for his valuable inputs and suggestions.

#### REFERENCES

- [1] Fan Wu, Hira Narang, Miguel Cabral, "Design and Implementation of an Interpreter Using Software Engineering Concepts", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 5, No. 7,2014
- [2] Alfred Aho and Jeffrey Ullman, "Principles of Compiler Design", 1977
- [3] [https://en.wikipedia.org/wiki/Interpreter\\_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))
- [4] Faisal Chughtai, "Compiler and Interpreter"