American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-5, Issue-6, pp-43-47
www.ajer.org
Research Paper Open Access

Design &Development of an Interpreted Programming Language

Shimul Chowdhury[1], Shubho Chowdhury[2]
[1]*Computer Science &Engineering, International University of Business Agriculture &Technology, Bangladesh*
[2]*Computer Science &Engineering, International University of Business Agriculture &Technology, Bangladesh*

ABSTRACT:*Programming Languages are playing one of the key roles in Computer Science, Software Development and other related fields. Learning Programming Language is essential for anyone who wants to be Programmer. But, to really understand the mechanics of how those Programming Languages work internally is difficult for various reasons. One simple solution to this problem is to Design and Develop a new Programming Language or a subset of another Programming Language. In our project we wanted to Design and Develop a learner friendly Programming Language, which will be very easy to recreate. We will show steps of creating such toy language which will help to learn internal works of a Programming Language.*

Keywords: *Programming Language, Interpreter, Finite State Machine, Language Design, Abstract Syntax tree, Backus–Naur Form*

I. INTRODUCTION

Our Programming Language is a toy Programming Language which is very easy to recreate. The purpose of designing this Language was not to be in industry, rather to help programmers to understand how Programming Languages actually works. Designing a Programming Language from scratch can be very tedious work if one does not know where to start. And to know a Programming Language very well one must know some fundamental Computer Science topics such as Lexers, Parsers, Abstract Syntax Tree, Theory of Computation, programming paradigms etc. By recreating this Language one can have a good practical overview on followings -

- How to build a Lexer (without any tool like lex)
- How to build a Parser (without any tool like yacc)
- How Finite Automata can be used for Regular Expressions
- What is the purpose of AST (Abstract Syntax Tree)
- How a basic interpreter works
- How to design a Language with BNF

1.1 Procedure

In order to design and develop a Programming Language we will follow following steps –

- Express the grammar of the Programming Language using BNF
- Develop Lexical Analyzer
- Develop Parser
- Implement an Abstract Syntax Tree for parsed source code
- Implement an Evaluator for evaluate expressions

1.2 Required Programs and Tools

As we are designing a Programming Language which main purpose is to be learner friendly, we should choose a high level Programming Language, C. Also to be friendly with beginner programmers who most probably using IDE's, we will use Code Blocks IDE in order to develop the interpreter. So only following tools are needed to develop the interpreter for our Programming Language –

- A Desktop or Laptop Computer
- Any Operating System (Windows / Linux / Mac)
- A “C” Compiler (MinGW)
- Code-blocks IDE

II. DESIGN

BNF is a standard way to express the grammar of a Programming Language. More specifically BNF or Backus-Naur Form represents a way to Context Free Grammar. An example of how Backus-Naur form can be used is follows-

<integer> ::= <digit>|<integer><digit>

Now as we are trying to design our own language, we need to first define our language constructs. So in our case we will be building a very small programming language which contains –

- Keywords: var, when, if, show
- Assignment operator: =
- Loop: when
- Conditional operators
- Arithmetic operators

Also we implemented Polish Prefix Notation for equations or statements. Because it gave us simplicity for operator precedence checking. Keeping that in mind we define our BNF grammar as followings –

```
<expression> ::= <statement><expression_prime> | <control-flow><expression_prime> | <end-of-file>
<expression_prime> ::= <statement> ‘\n’ <expression> | <control-flow> ‘\n’ <expression> | ‘\n’
<statement> ::= var identifier | ‘=’ identifier <term> | show <term>
<control-flow> ::= if ‘(’ <evaluation> ‘)’ <block> | when ‘(’ <evaluation> ‘)’ <block>
<block> ::= ‘{’ <expression_prime> ‘}’ <expression_prime>
<term> ::= identifier | number | string | ‘(’ <evaluation> ‘)’
<evaluation> ::= operator <term><term>
```

So a program written on our programming language will look like following-

```
var x = x 1
when(< x 100) { show x = x (+ x 1) }
```

III. DEVELOPMENT

Now that we have our grammar for the programming language and all tools available. We can start developing our programming Language. As we discussed before we will be using following steps one by one in order to develop it.

- Lexical Analyzer
- Parser
- Abstract Syntax Tree
- Evaluator

3.1 Lexical Analyzer

Lexical Analyzer takes a source code file as input and outputs a sequence of token. It reads characters from input source file one by one and generates token. Usually a Lexical Analyzer makes use of Finite State Machines to identify tokens from sequence of characters. So for our programming language we defined an enum type in C for tokens.

```
typedefenum { IDENTIFIER, STRING, NUMBER, ASSIGNMENT_OPERATOR, ADDITION_OPERATOR,
SUBSTRUCTION_OPERATOR,MULTIPLICATION_OPERATOR,VISION_OPERATOR,LT_OPERATOR,
BT_OPERATOR, LEQ_OPERATOR, BEQ_OPERATOR, EQ_OPERATOR,
NE_OPERATOR, VAR, IF, WHEN, SHOW, LP, RP, LCB, RCB, NEWLINE
} TOKEN_TYPE;
```

For identifying keywords are easy. We can just use strcmp function in C. Like - strcmp(chunk, "var"). Also we can easily identify operators like +, -, *, /, % and brackets, as we just need to check one character. But we need to check identifiers and numbers too. For that we can make use of finite state machine. Here also we tried to be as simple as possible, so we defined couple of rules for that.

For Identifiers - Identifier can have underscore (_) or an alphabet as first character. After that it can have any combination of underscores (_) or alphabets or numbers. So we got our state diagram for Identifiers -

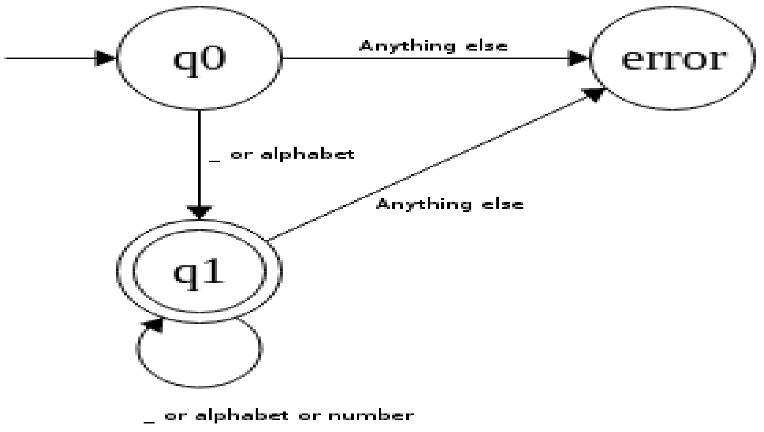

Figure: FSM for identifiers

For Numbers: Numbers can have two variations, Integers and Decimals. So the rule is - if there only numeric character, accept it. If we get a dot (.) then we will look for at least another numeric value after it. So we got state diagram for numbers –

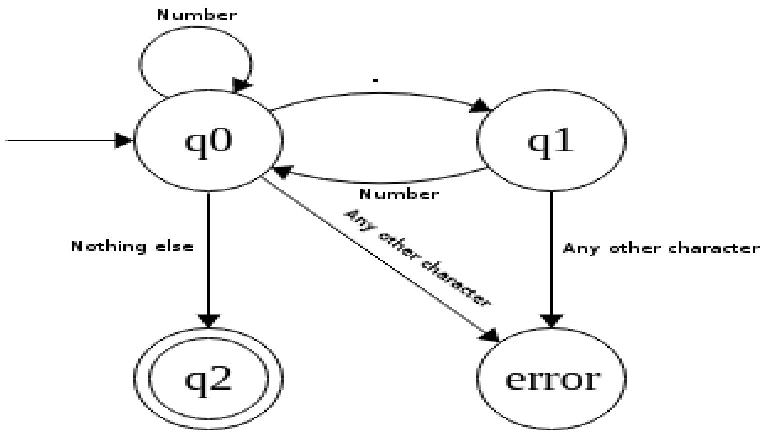

Figure: FSM for Numbers

Finally, after finishing generation of tokens Lexical Analysis gives us an array of tokens as output, which we will use in next step to build Abstract Syntax Tree.

3.2 Parser

Parser is another phase in both compiler and interpreter, which takes array of tokens as input and builds Parse Tree / Abstract Syntax Tree. The grammar we defined before will be used to build that Parse Tree here. There are mainly two types of parsing - Top Down and Bottom Up. We will use Top Down parsing as we want it to be simple to understand.

We can imagine every rule of our grammar as sequence of function calls. For example -

```
AST_NODE* control_flow(){
        if(current_token_type == IF){
                AST_NODE* ctrl_flow = create_ast_node(CONTROL_FLOW);
                ctrl_flow->childs[0] = if_t();
                lp_t();
                ctrl_flow->childs[1] = evaluation();
                rp_t();
                ctrl_flow->childs[2] = block();
                returnctrl_flow;
        }else if(current_token_type == WHEN){
                AST_NODE* ctrl_flow = create_ast_node(CONTROL_FLOW);
                ctrl_flow->childs[0] = when_t();
                lp_t();
                ctrl_flow->childs[1] = evaluation();
                rp_t();
                ctrl_flow->childs[2] = block();
                returnctrl_flow;
        }else{
                show_parser_error_and_exit(PARSER_ERROR_UNEXPECTED, line_number, IF,
        token_type);
        }
        return NULL;
}
```

Note that as our Parser going to provide a parse tree / Abstract Syntax Tree as output, we build a tree data-structure. AST_NODE is a node in that tree. Create_ast_node function adds a new node in existing tree and that's how we are building our Syntax Tree. In this example we have shown how we are parsing control flow in our implementation. Like this one every rule is being parsed and Syntax Tree is getting build. Also we can see that those rules are actually function calls in Top Down Parsing, like - block(), if_t() etc.

3.3 Abstract Syntax Tree

We have seen how we are building abstract syntax tree from our tokens. While building Syntax Tree we can match source program with our defined rule and determine if there any syntax error. For example -

```
voidlp_t(){
        if(current_token_type==LP){
                open_p++;
                …
        }
}
voidrp_t(){
        if(current_token_type == RP){
                open_p--;
                …
        }
}
```

These two functions are for left and right parenthesis token matching. As per our grammar; if we get a left parenthesis, we will go inside a block. And to come out from a block we need to get a right parenthesis and there might be nested blocks in source program. So we are keeping track of how many parentheses currently open and if we get right parenthesis we will decrease number of open parenthesis. So if our source program has error regarding parenthesis, we can easily determine and throw a syntax error.

As we got our Abstract Syntax Tree we can now go for evaluation or interpreting and actually giving output as per source program.

3.4 Evaluator

We are calling this phase Evaluator as this phase takes Syntax Tree as input and run the necessary actions described in source program. Evaluator will take Syntax Tree and work like a Depth First Search on the Tree. It will go deep inside the tree and evaluate child nodes first and return back the value. After evaluating every child node of root it will give actual output. Only Exception is the "show" keyword. Evaluator will print whenever it gets "show" keyword in the syntax tree. For example -

```
= x 1
show x
```

It is a small code which will go throughLexer first. We will get an array of tokens which will be following -

```
token_array[0] : {token_type =ASSIGNMENT_OPERATOR, value = null, token_id = 0}
token_array[1] : {token_type =IDENTIFIER,value = "x"token_id =}
token_array[2] : {token_type =NUMBER, value = "1"token_id = 2}
token_array[3] : {token_type =NEWLINE, value = nulltoken_id = 3}
token_array[4] : {token_type = SHOW, value = nulltoken_id = 4}
```

token_array[5] : {token_type = IDENTIFIER, value = “x”token_id =}

So this token array will go throw our Parser which will build syntax tree -

```
Expression
    Statement
        Assignment_Operator
        Identifier
        Number
    EndStatement
    expression_Prime
        Statement
            Show
            Term
                Identifier
            EndTerm
         Endstatement
        Expression
            End_Of_File
        EndExpression
    EndExpression_Prime
End Expression
```

Now our Evaluator will visit this tree in left to right manner and will evaluate those expressions. Our Evaluator will maintain a data-table to keep track of variables and their values. In our implementation, this data-table is a simple binary tree. As we know the complexity of binary tree is O(logn), it gives a decent performance.

There are somehelper functions written for our Evaluator. Like string based addition, subtraction. By this way we are leaving complexity of selecting proper data-types like - int, long, float. Everything is string manipulation in our interpreter. It gives us a simple way to build a language quickly.

IV. FUTURE PLAN

There are number of improvements and features can be added to this Programming Language. But we are very careful about choosing what improvement we should do next. Because we do not intend to compete against industry standard language, rather we want this Language to be simple and easy to recreate. We are not solving problems for software industry with this programming language. We are trying to build a language which can be easy to develop for a new programmer, so that he/she can understand programming languages more than he/she used to be. Being said that, followings are the improvements we think we will work on future -

- Re-factor code and provide more useful comments
- Implement support for functions
- Implement scope of variable
- Provide build in server for web (So we can create user interface in web) etc

V. CONCLUSION

There are number of Education purpose Programming Languages out there. But not every language is suitable for a new comer to implement on his/her own. Design and Development of Programming Languages are mystery for most of students,as there are not much practical resources out there. We believe and hope that this Programming Language can help them learn more about Programming and how programming languages work. One can easily create aCompiler after he/she can implement a lexer and parser. Our work can be found on following link - *https://github.com/theprog/oboni*.

REFERENCES

[1] Abstract Syntax Tree https://en.wikipedia.org/wiki/Abstract_syntax_tree

[2] Alfred V. Aho, Ravi Sethi, Jeffery D.Ullman, Compilers Principles, Techniques, and Tools - 2006-2007 pp 172 - 180

[3] Compiler Construction/Lexical analysis https://en.wikibooks.org/wiki/Compiler_Construction/Lexical_analysis

[4] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, Clifford SteinIntroduction to Algorithmsthird editionpp 286 -