INTERPRETATION OF JAVA - OBJECT ORIENTED LANGUAGE: APPLICATION OF CHOMSKY CONTEXT FREE GRAMMAR AND FINITE STATE AUTOMATION

Asoke Nath¹, Anwesha Chakraborty² and Suvasish Dasgupta³

Abstract- The use of programming language is an integral part of Computer Science and Engineering. Human Computer Interaction (HCI) is done through some high level computer language. To make the program readable to a computer the programmer has to compile or to interpret the source code with some compiler or some interpreter. To design any computer language the user must have complete knowledge of computer hardware system and also its implementation strategy. In the present work the authors have made an extensive study on the issue of designing a language. In order to explain the concept, the authors have dealt with Java language since it is the most commonly used language nowadays.

Keywords – HCI, Finite State Automation, Symbols, Formal Language, Grammar

I. INTRODUCTION

We all have dealt with programming vividly and learned enough about it. In this article, our main objective is to realise the technicalities of any language and see how the languages and its grammars are defined and implemented using any machine. In short we discuss the ways to build any language and design a machine(i.e. a translator) that accepts the language and produces some valid output.

To understand what a language is, we first explain the following:

Symbols: 0, 1, a, b, c, ....

Alphabet: It is a finite set of symbols. Eg: {0, 1}, {a, b, c, d, ....z}, ....

Say we consider the alphabets {a, b, c}, then the strings over these symbols are on ordered way of writing these symbols one after another. Eg: baaba, aabbc, bbca, ..... 

Suppose,

Σ denotes the set of symbols (i.e alphabet),
Σⁿ denotes the set of all finite strings over Σ. It is basically an infinite set but all of its members are finite just like that of set of numbers.

For machine language, Σ = {0, 1}. then Σⁿ is a set of all finite binary strings over Σ. i.e. Σⁿ = {00, 10, 101, 1011, 10100, 1101, .......}. So, a Formal language L over the alphabet Σ is a proper subset of Σⁿ.
Example: If $\Sigma = \{0,1\}$; $L = \{01, 110, 0011, 1010\}$ is a set of strings over alphabet $\{0,1\}$ which is a subset of $\Sigma^*$

Mathematically, we can define our own language in the following manner:

$L_1 = \{x \in \Sigma^* | x \ has \ even \ number \ of \ 0's \ and \ even \ number \ of \ 1's\}$. It is to be understood that not all elements of $\Sigma^*$ belong to $L_1$ as $\Sigma^*$ contains all kinds of binary strings. Therefore we say that $L_1$ is a proper subset of $\Sigma^*$.

We shall be concerned with the set membership problem of formal language. When we write a program, the program takes an input which is a string over some alphabet, and the output which is also a string.

We think of some model of computation i.e. an abstract way to specify any algorithm.

In order to design any language, besides above properties, there must be a set of rules which defines that language (commonly known as Grammar of that language). To understand this, we take the example of English language.

We know that for English language,

$$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, \text{'.'}, \text{'}?\}$$

It is to be noted that $\Sigma^* = \{\text{apple, ball, cat, dog, fgh, mnu, shfru, god, dgo, ............}\}$

If we follow the rules for English language which we have studied at elementary level, we see that the elements such as fgh, mnu, dgo though elements of $\Sigma^*$ do not belong to English language. So not all the elements of $\Sigma^*$ belong to English language and hence we say the Language is a proper subset of $\Sigma^*$.

So we define English language as

$$L_{\text{ENGLISH}} = \{x \in \Sigma^* | x \ obeys \ the \ English \ grammar\}$$

Now to understand English grammar, consider the sentence: “GOOD STUDENTS GO TO SCHOOL REGULARLY”

To parse the above language, we consult the tree that defines the grammar :-

A <SENTENCE> can be rewritten as <NOUN PHRASE> and <VERB PHRASE>. A <NOUN PHRASE> can again be rewritten as <ADJECTIVE> and <NOUN>. A <VERB PHRASE> can be rewritten as <VERB>, <PREPOSITION>, <NOUN> and <ADVERB>. An <ADJECTIVE> can be replaced by the word ‘GOOD’ and a <NOUN> by the word ‘STUDENT’. Similarly a <VERB> can be replaced by ‘GO’, a <PREPOSITION> by ‘TO’, <NOUN> by ‘SCHOOL’ and <ADVERB> by ‘REGULARLY’.

So the moment we find that the input string matches with the syntax of the English grammar, we can say that the above sentence belongs to the set of languages generated by above grammar i.e. $L(G)$.

The process of replacement using the above set of productions should always begin from the start symbol $S$ of the grammar. The process of such replacement or derivation should end at the step where the obtained symbols cannot be further replaced i.e. the obtained symbols are terminals (elements $\in \Sigma^*$).

Mathematically if $L(G)$ be the language generated by grammar $G(S, \Sigma, V, P)$ where $S$ denotes the start symbol, $\Sigma$ denotes the set of terminals, $V$ is the set of non-terminals and $P$ denotes the set of production rules, then

$$L(G) = \{x \in \Sigma^* | S \Rightarrow^* x\}$$

Where $\Rightarrow^*$ denotes after successive replacements or derivations.

We take a step ahead and try to elaborate the language structure of Java using production rules just like we have done for English language.
II. JAVA LANGUAGE AND ITS INTERPRETATION USING FINITE STATE AUTOMATION

In order to explain this article, knowledge of Java is a prerequisite criteria and it is taken for granted that the reader has basic knowledge of Java.

Any Java statement can be classified into three categories namely:

1) **Expression**: These are the important building blocks of the language which contains variables, operators or method call. Following are the examples:
   i. `int a=40;` which is declaration and initialisation
   ii. `a=b+c;` which is performing arithmetic operation and assigning the result in another variable.
   iii. `ob.f1();` which is calling a function f1() of a class using object ob of that class

2) **Declaration**: These are the statements that are used to declare any variable to be used in the program. It includes declaration of functions, objects, variables including arrays. The examples are given below:
   i. `int a;` declares a variable a of integer data type
   ii. `int arr[]=new int[30];` declares a an array arr of integer type which can hold 30 consecutive integer numbers.
   iii. `Class_A ob=new Class_A();` declares an object ob of class A. It is to be noted that Class_A is a kind of data type for the object ob.

3) **Control flow**: These are the statements that direct the flow of program control depending on the fulfilment of required conditions. There are three types of control statements namely Selection, Jump and iteration, which have been explained in the subsequent part of the journal.

- Since anything in java starts from a statement, so `<statement>` (Rule 1) is the start symbol or sentence for the set of production rules. The statement hence can be rewritten as `<expression>` or `<declaration>` or `<Control_flow>.
- Similarly, `<expression>` (Rule 2) is nothing but a combination of operators and variables i.e `<variable><operator><variable>` (as in case of `a=b`) or `<variable><operator><expression>` (as in case of `a=b+c` where `a` is the variable, `=` is the operator and `b+c` is an expression).
- The production rule of `<expression>` also has a non-terminal `<method_call>` (Rule 48) where method is just another name for ‘member functions’ in object oriented programming language viz. java. It is expanded as `<object_name>.<method_name>()`. For example, `BS.bubble_sort(a,id,n);` where `BS` is the object name and `bubble_sort` is the method name.
- In the similar manner, the `<control_flow>` can also be expanded as Selection, Jump, Iteration i.e either `<selection>` or `<jump>` or `<iteration>`.

- Next is the production rule for the `<operators>` (Rule 3) which can be expanded as `<arithmetic>` (Rule 4) (eg: `x++`), `<bitwise>` (Rule 5) (eg: `if(a>50 && a<200)`), `<relational>` (Rule 6) (eg: `while(ch!='-1')`). A language can have numerous arithmetic and logical operations based on these three.
- Furthermore, `<arithmetic>` can be expanded as `+(addition), -(subtraction), ++(increment), --(decrement), %(modulus)` and so on. `<bitwise>` can be expanded as `& &(bitwise AND), ||(bitwise OR)` etc. `<relational>` can be expanded as `==(equals to), !=(not equals to), >=(greater than equals to)` and so on.
Then comes <decoration> (Rule 7) which can be expanded to <data_type><space><symbol> as in “int temp;” where ‘int’ being the data type, ‘temp’ being the variable or symbol with a space in between. This is the convention to declare any sort of symbol be it character or integer in java or any other language.

Next comes <symbol> (Rule 8) in the set of production rules whose expansions are <variables>.new<space><data_type>[<n>] and <data_type><space><symbol><numbers>. Let us consider the expansion ‘new<space><data_type>[<n>]’; in java, arrays can be declared in this manner. For example int a[]=new int[20]. <variables> (Rule 9) can also be expanded which contains any user defined variables containing a single alphabet or a set of alphabets (like a word), or a combination of alphabet and numeric value viz. int a ,x1, p22, num;

After that we have <numbers> (Rule 10) which can be expanded in <digits> and <numbers> which is true for any language. <numbers> can have digits(0,1,2,...), two digits numbers(78,12,90,...), three digit numbers and so on. <digits> (Rule 11) can be further expanded to 0,1,2,3,...,9. Moreover, a number can be signed as well. So we have kept provision for signed number also by incorporating the non-terminal <sign> (Rule 22), which can produce negative numbers like -23,-345,... etc. Similarly, the case of fractional number has also been considered.

Next we have <datatypes> (Rule 23) which according to java programming language can be expanded as <primitive> datatypes and <reference> datatypes. <primitive> (Rule 24) can be further sub divided into numeric, fractional, character and Boolean. Where <numeric> (rule 25) have byte, short, long and int. <fractional> (Rule 16) have float and double. <character> (Rule 27) have char and <Boolean> (Rule 28) have boolean.

Again, <reference> (Rule 29) can be expanded as class_name><space><variable>= new<space><class_name>(). For example in java we write a reference as-

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

<class_name> (Rule 30) can also be expanded which contains any variable with a constraint in java language that it must begin with a capital letter. For example: class Bubble_Sort {... }.

As far as <control_flow> (Rule 31) is concerned, <selection> (Rule 32), whose expansions gives us choice as to which type of decision control statement we want to use-<if_else> or <switch>.

1. <if_else> (Rule 33) can be further expanded to <stat1> i.e. an ‘if’ can have a single conditional statement. Or to <stat1><end_of_line>else{<space><statement>} i.e. an ‘if’ can have a conditional statement along with an else which can have another conditional statement; either one of them can be true. Again, <if_else> can have nested if_else statements i.e. if_else inside another if_else with same format as a normal if_else. For example, if (test expression1)

    |
    Statement block1;
    |
else if (test expression 2)
|
    if (test expression 2.1)
|
Statement block 2.1:
} 
else 
{
Statement block 2.2: 
}
}
...
else 
{
Statement block x;
}

<stat1> can be further expanded to if(<condition>){
  <space><statement>}
 i.e. if the if statement is true then it can have another condition followed by it.

2. Another kind of selection statement is switch which has been represented as <switch> (Rule 36). Since lets us select statements on the basis of match with the cases, we have the following syntax:

```
switch(choice) {
  Case 1: statement 1; break;
  Case 2: statement 2; break;
}
```

So, if choice=1, then statement 1 is selected and for choice=2, statement 2 is selected. Thus for production rule for <switch>, a non-terminal <inner_switch> (Rule 37) has been used which deals with the ‘case’ alignment within the switch body.

On the other hand, the jump statement of java language can be of two types namely break and continue.

As far as iteration statements are concerned, there are for, while and do while statements.
1. `for (initialisation expression; test expression; update expression)` leads to the production rule number 30. The non terminals <stat3>,<stat4> and <stat5> represents the initialisation expression, test expression and update expression respectively. (Refer rule 40)
2. `while(condition) {
  statement
}` leads to production Rule 44.
3. `do {
  statements
}while(condition):` is also an iterative statement which executes the statement at least once even if the condition is false. The body of do while is iterated as long as the condition evaluates to true. (Refer rule 35)

So the techniques of expansion continues and we get an elaborate set of production rules for java programming language. Given below are the set of few production rules of Java language that are most commonly used by Java compiler:

1. `<statement> →
  <expression>|<declaration>|<control_flow>|<expression><declaration><control_flow>|<declaration><control_flow><expression>|<expression><control_flow><declaration>|<control_flow><expression><declaration>|€
  
2. `<expression> →
  <variable>|<variable><operator><expression>|<variable><operator><number>|<method>|(|<expression>|)|<number><operator><number>
  
3. `<operator> →
  <arithmetic_operator>|<bitwise_operator>|<relational_operator>
  
4. `<arithmetic_operator> →
  +|->[*]|/ |% |++|--|+=|=|=|=|% |&|
Interpretation of Java - Object Oriented Language: Application of Chomsky Context Free Grammar and Finite State Automation

5. <bitwise_operator> → | & || ^ > >> >>>> | << = &= |= ^= >> >= >>>= << = &
6. <relational_operator> → == != | >= | <=
7. <declaration> → <data_type> <space> <symbol> <end_of_line>
8. <symbol> → <variable> | new <space> <data_type> | <numbers> | <operator> <numbers>
9. <variables> → <alphanums> | <alphanums> <variables> | <alphanums> <digit> | <alphanums> <numbers>
10. <numbers> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
11. <numbers> → <integer> <digit>
12. <numbers> → <n1> <scale> <integer> <frac>
13. <n1> → <integer> <frac>
14. <integer> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
15. <integer> → <integer> <digit>
16. <frac> → <t1> <integer>
17. <t1> → .
18. <scale> → <n2> <integer>
19. <n2> → <t2> <sign>
20. <t2> → e
21. <digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
22. <sign> → + | -
23. <data_types> → <primitive> | <reference>
24. <primitive> → <numeric> | <fractional> | <character> | <Boolean>
25. <numeric> → byte | short | int | long
26. <fractional> → float | double
27. <character> → char
28. <Boolean> → true | false
29. <reference> → <class_name> <space> <variable> = new <space> <class_name>()
30. <class_name> → any variable beginning with capital letter <space> { <statement> }
31. <control_flow> → <selection> | <jump> | <iteration>
32. <selection> → <if_else> | <switch>
33. <if_else> → <stat1> | <stat1> <end_of_line> else { <space> <statement> } | <stat1> <end_of_line> else if <space> <statement> } <if_else>
34. <stat1> → if <condition> { <space> <statement> }
35. <condition> → <condition> <relational> <condition> | <expression> <condition> <expression>
36. <switch> → switch <variable> { <inner_switch> }
37. <inner_switch> → case <variable> : <statement> <end_of_line> <break> <end_of_line>
38. <jump> → break <end_of_line> continue <end_of_line>
39. <iteration> → <for> | <while> | <do_while>
40. <for> → for <stat3> ; <stat4> ; <stat5> { <statement> }
41. <stat3> → <variable> <arithmetic> <variable> <variable> <arithmetic> <number>
42. <stat4> → <variable> <relational> <variable> <variable> <relational> <number>
43. <stat5> → <variable> <arithmetic>
44. <while> → while <condition> <space> { <statement> }
45. <do_while> → do { <statement> } while <condition> <end_of_line>
46. <end_of_line> → ;
47. <method_def> → <variable> () { <statement> } | <variable> ( <variable> ) { <statement> } | <variable> <declaration> { <statement> }
48. <method_call> → <object_name> . <method_name> ()
49. \(<\text{object}_\text{name}>\) → variable that was used to declare object
50. \(<\text{method}_\text{name}>\) → <variables>
51. \(<\text{alphabets}>\) → a|b|c|d|…..|z

In order to understand the use of above rules, we take a code fragment as follows:
1. int i;
2. int s=0;
3. for(i=0;i<n;i++)
4. {
5. s=s+i;
6. }
7. int m=s/n

Now to verify whether the above code obeys the grammar of the language, we examine line by line (table-1):

### Table-1

<table>
<thead>
<tr>
<th>Line number</th>
<th>Statement</th>
<th>Production Rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int i</td>
<td>(7)\text{&lt;declaration&gt;\rightarrow&lt;data_type&gt;&lt;space&gt;&lt;symbol&gt;&lt;end_of_line&gt;}) (SUCCESSFUL PARSING )</td>
</tr>
<tr>
<td>2</td>
<td>int s=0</td>
<td>Rule 7(\text{&lt;declaration&gt;\rightarrow&lt;data_type&gt;&lt;space&gt;&lt;symbol&gt;&lt;end_of_line}&gt;}) followed by rule 8 (\text{&lt;symbol&gt;\rightarrow&lt;operator&gt;&lt;numbers&gt;}, followed by expansion of \text{&lt;symbol&gt;,&lt;operator&gt;,&lt;numbers&gt; and &lt;end_of_line&gt;} (SUCCESSFUL PARSING )</td>
</tr>
<tr>
<td>3</td>
<td>for(i=0;i&lt;n;i++) { …}</td>
<td>Rule 31,40,41,42,43 (SUCCESSFUL PARSING )</td>
</tr>
<tr>
<td>5</td>
<td>s=s+i;</td>
<td>Rule 2. \text{&lt;expression&gt;\rightarrow&lt;variable&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Int m=s/n</td>
<td>Invalid as it does not contain \text{&lt;end_of_line&gt;} (UNSUCCESSFUL PARSING )</td>
</tr>
</tbody>
</table>

### III. CONCLUSION AND FUTURE SCOPE

The above rules more or less define the language and acts as a tool for machines to interpret/compile and hence check the syntactical validity of any statement of the program. Moreover, a mathematical model of a machine can be formulated which will accept the production rules and the program as input, and will produce the output as 1 if the program abides the rule of the language, otherwise 0. So for the above code fragment, the machine will return 0 as the machine failed to parse one line successfully. The machine can be further designed to point out the location of mismatch of the statement with rules if any (fig-1).
REFERENCES