Control Theory and Informatics
ISSN 2224-5774 (Paper) ISSN 2225-0492 (Online) DOI: 10.7176/CTI
Vol.8, 2019

DESIGN AND IMPLEMENTATION OF A LANGUAGE SCANNER GENERATOR (KnitAutoGen)

Chile-Agada B. U. N.[2 *], Offiong N. M.[1], Adehi M. U.[1],

[1]Department of Mathematical Sciences, Nasarawa State University, Keffi, Nigeria.

[2]Department of Computer Science Education, Alvan Ikoku Federal College of Education, Owerri, Nigeria.

Abstract

The need for fast, efficient and simple scanner generator that has the primary responsibility to perform efficiently gave rise to this paper. This is due to the fact that, on daily basis new technologies arise which brings a great improvement on the design of computer architecture. However, attention was given to speed, run time and resource availability of the design machine to be used since lexical analysis has an impact on how the compiler works. This paper seeks to develop a lexical analyzer (scanner generator) automatically by specifying the lexemes patterns to a lexical analyzer generator and compiling those patterns into a code that functions as a lexical analyzer. The scanner accepts characters as input and breaks them down to produce tokens by grouping the characters and not deviating from specifications. The project employs one of the different methods of lexical analyzer generator to perform pattern-matching on text using regular expression over a global character set. The paper shows how input is matched and specifies what to do when a pattern is matched. This was achieved with the use of regular expressions (RE) which were converted to non-deterministic finite automata (NFA) or deterministic finite automata (DFA). The regular expression and the regular grammar were thus joined together mathematically. Various results are presented and further work on micro compilers was proposed.

Keyword: Scanner generator, Regular expression, KnitAutoGen, DFA, NFA.

DOI: 10.7176/CTI/8-02

INTRODUCTION

The world revolves around computing and mankind seeks diverse ways to make computation easy. Programming actually makes computation easy for the user and describes computation to the machine. Without the complier, a computer program cannot run effectively because the compiler helps to translate the source code into machine understandable language and subsequently produce output in another language. To that end, scientists have come up with different compilers and the first complier took about 18 working years to be produced. After the first FORTRAN complier was produced, several other compliers were produced and they are all useful to the computing world.

The compiler is broken down into phases to handle the source code till output is generated. The first phase of a compiler is called lexical analyzer or scanner. The lexical analyzer read the streams of character making up the source program and groups the characters into meaningful sequences called lexemes (Aho A., Lam M., Sethi R and Ullman J, 2007). In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer, *lexer* or scanner. Typically, the lexical analyzer (scanner) is the input routine or the translator, reading successive lines of input programs, breaking them down into individual basic items and feeding these items to the latex stage of the translator to be used in the higher level of analysis. The lexical analyzer must identify the type of each lexical item and attach a type-tag to it. In addition, conversion to an internal representation is often made for items such as numbers, symbols etc.

Developing a language scanner generator to accept language and produce output is very important in the design of a compiler because without accepting inputs, tokens will not be produced and that will affect subsequent phase of the complier. This project deals with analyzing the design of a language scanner generator and coming out with a scanner generator that does the same function of the existing lexical analyzers without deviating from the specifications of a reliable and efficient scanner generator. The study of compliers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency. Some of the most fundamental models are finite-state machines and regular expressions and they shall form the basis of this project. These models are useful for

describing the lexical units of programs (keywords, identifiers, and symbols) and for describing the algorithms used by the complier to recognize those units. The program we are going to use in the scanner design, will harness the parallel nature of the computer architecture to deliver high performance to other phases of the complier.

OVERVIEW OF LEXICAL ANALYZER

The lexical analyzer is at the phase of compiler and its main task is to read the input character of the source code, group them into lexemes and produce as output, a sequence of tokens for each lexeme in the source program. This is done not without interaction with the symbol table.

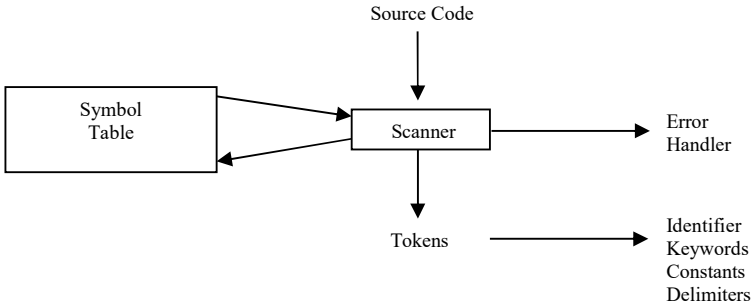

Fig.1: Lexical analysis

Symbol Table: The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Error Handler: This interacts with the lexical analyzer to handle errors that might arise. The error handler is there to handle error so that compilation can continue to allow for further error detection. At the lexical phase can be detected where the characters remaining in the input do not form any token of the language specification.

METHODOLOGY OF GENERATING SCANNERS (lexical analyzer)

Presently, we have two ways of generating scanner namely:

Hand Implementation: Hand implementation can be achieved in the following ways.

Manual Implementation: This is a case whereby we manually generate scanners diagram to produce tokens.

Transition Diagram: This is a case where we convert a transition diagram into a state machine.

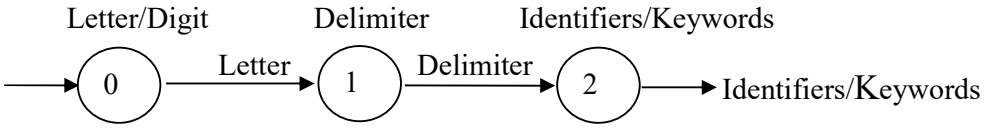

Fig.2: Transition Diagram

Automatic Generator: This is a process of generating scanners automatically by specifying the pattern of the lexemes to a lexical analyzer generator and compiling those patterns into codes that functions as a lexical analyzer. There are many automatic generators already in existence and they include: Flex (Fast Lexical Analyzer), JFlex (Java Flex), GPlex (Garden Point Lexical Analyzer) etc.

SOME TYPE OF SCANNER GENERATORS

There are several types of scanner but I am going to review three of them which are: *Flex, JFlex* and *GPlex.*

Flex: This is also known as *lex* and it works by reading the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules.* Flex generates as output a C source files which defines a routine. This is compiled and linked with library to produce an executable. When the executable is run, it analyzes its input occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code. The following flex input specifies a scanner which whenever it encounters the string "username" will replace it *ith* the user's login name:
Username *printf* ("%S", *getlogin* ( ) ;

By default, any text matched by a flex scanner is copied to the output, so the net effect of this scanner is to copy its output files to its output with each occurrence of "username" expanded. In this input, there is just one rule. "Username" is the pattern and the *"Printf"* is the action. The *"%%"* marks the beginning of the rules.

JFlex: JFlex is a lexical analyzer generator for Java written in Java. It is also a rewrite of the very useful tool *JLex* that was developed by Elliot *Berk* at Princeton University. As Vern *Paxson* states for his C/C++ tool flex: they do not share any code though. JFlex generates a java file with one class that contains code for the scanner. The class will have a constructor taking a java.io.Reader from which the input is read. The class will also have a function *yylex()* that runs the scanner and that can be used to get the next token from the input (in this example the function actually has the name next token () because the specification uses the %cup switch/).

GPlex: Gardens Point Lex *(GPLex)* is a scanner generator which accepts a "LEX-like" specification and produces (# output file. The implementation shares neither code nor algorithms with previous similar programs. The tool does not attempt to implement the whole of the *POSIX* specification for *LEX,* however the program moves beyond *LEX* in some areas, such as support for Unicode. The scanners produced by *gplex* are threading safe, in that all scanner state is carried within the scanner instance. The variables that are global in traditional *LEX* are instance variables of the scanner object. Most are accessed through properties which expose only a gettter. The implementation of *gplex* makes heavy use of the facilities of the 2.0 version of C#. These are two main ways in which *gplex* is used. In the most common case, the scanner implements or extends certain types that are defined by the parser on whose behalf it works. Scanners may also be produced that are independent of any parser, perform pattern matching on character streams.

METHOD FOR DESIGNING THE SCANNER GENERATOR (KnitAutoGen)

This section targets the method adopted in the development of a lexical analyzer (scanner) automatically. In this section we explored the fastest possible method to the design of the scanner generator. Since the paper is also concerned about how to carry out the implementation on the computer system, we are going to look at the requirements that will enable us do those efficiently. To that end regular grammar, regular expressions (RE), non-deterministic finite automaton, (NFA) and deterministic finite automaton (DFA) will have to come in here.

PROJECT (SYSTEM) ANALYSIS

Scanner:

1. This maps characters into tokens – the basic unit of syntax e.g.

   x = x+y

   This becomes

   <id, x> = <id, x> + <id, y>;

2. Characters string value for a token is a lexeme e.g.
   Score = initial * rate + 30
   The mapping of the above expression is clearly shown below:

Lexemes	**Tokens**
Score	<id, 1>

=	< = >
Initial	<id, 2>
*	<*>
Rate	<id, 3>
+	<+>
30	<30>

3. It should also be noticed that the blanks separating the lexemes would be discarded by the lexical analyzer. It therefore means that the expression above will appear thus:
<id,1> <id,2> <*> <id,3> <const, 4>
4. In the representation above, the token = * + are abstract symbols for the assignment, multiplication and the addition operators respectively. 1, 2, 3 and 4 are internal representation of the identifiers score, initial, rate and the integer 30.
5. The main tokens in our examples are:
Id, =, *, and const

The key issue here is speed and specialize recognizers will be used (as against lex) but we are going to sacrifices a little space to achieve speed.

Specifying Patterns: A scanner must recognize various tokens some of these are quite easy e.g.

White space

<ws> : : = <ws> ‘ ‘

Keywords and operators

Specified as literal patterns: do, end

Comments

Opening and closing patterns delimiters /*….*/

Some are quite hard:

Identifiers

Alphabets followed by k alphanumeric (^, $, &,…)

Numbers

Integers: 0 or digit from 1-9 followed by digits from 0-9

Decimals: integer’. ‘digits from 0-9

Reals: (integer or decimal) ‘E’ (+ or -) digits from 0-9

Complex: ‘(‘ real’, ‘real’)’.

To specify the above patterns, we need a more powerful notation.

Regular expressions are convenient for specifying lexical tokens, but we need a formalism that can be implemented as a computer program. For this we can use finite automata (N.B the singular of automata is automaton). A finite automaton has a finite set of states, edges lead from state to another, and each is labeled with a symbol. One state is the start state, and some of the states are distinguished as final states.

Table 1: OPERATIONS LANGUAGES

OPERATION	DEFINITION
Union of L and M Written as L U M	L U M = {s/s e L or Se M}
Concatenation of L and M Written as LM	LM = {st/se L and t e M}
Kleen closure of L Written as L*	L* = U_t =o ∝
Positive closure of L Written as L^+	

Recognizers: From a regular expression we can conduct a deterministic finite automaton (DFA).

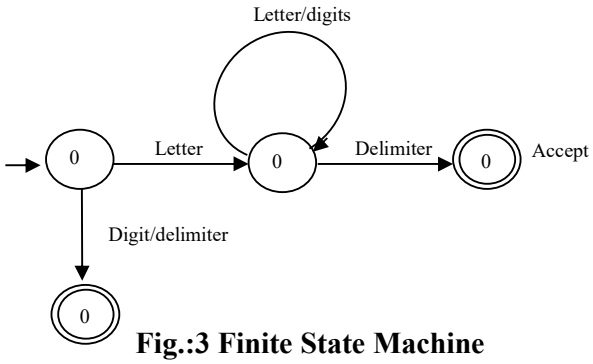

Fig.:3 Finite State Machine

Identifier
Letter ⟶ (a/b/c…/Z/A/B/C/…/Z)
Digits ⟶ (0/1/2/3/4/5/6/8/9)
id ⟶ (letter/digits)*

pseudo code for the recognizer

start: goto state 0
state 0: read x
if x = letter goto state 1
if x = digit goto state 3
accept string
state 1: read x
if x = letter goto state 1
if x = digit goto state 1
state 2: accept string

We will specify lexical tokens using the formal language of regular expressions to implement lexers using deterministic finite automata, and use mathematics to connect the two. This will make simpler and more readable lexical analyzers.

In writing regular expressions, we will sometimes omit the concatenation symbol or the epsilon, and we will assume that Kleene closure "binds tighter" than concatenation, and concatenation binds tighter alternation; so that ab/c means (a.b) / c, and (a/). Thus, using regular expression we specified lexical tokens of any source code.

1.	If	If
2.	[a-z] [a-z0-9]*	ID
3.	[0-9]+	REAL
4.	[0-9] + ". "[0-9]*" / [0-9]*". "[0-9]+)	no token, just white space
5.	("--"[A-Z]*\N") | (" "|"\N"|" \N")+	no token, just white space
6.		error

The fifth line of the description recognizes comments or white spaces but does not report back to the parser. Instead, the white spaces are discarded and the *lexer* is resumed. The comments for this *lexer* begin with two dashes, contain only alphabetic characters, and end with new line.

Finally, a lexical specification should be complete, always matching some initial substring of the input; we can always achieve this by having a rule that matches any single character (and in this case, "illegal character" error message and continues).

In a deterministic finite automaton (DFA), no two edges leaving from the same state are labeled with the same symbol. A DFA accepts or rejects a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character.

After making *n* transitions for an *n*-character string, if the automaton is in a final state, then it accepts the string. If it is not in a final state, or if at some point there was no appropriately labeled edge to follow, it rejects. The *language* recognized by an automaton is the set of strings that it accepts.

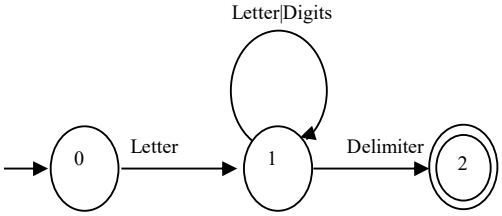

Fig. 4 Deterministic Finite Automata

It is clear that any string in the language recognized by automaton ID must begin with a letter. Any single letter is state 2, which is final; so a single-letter string is accepted. From state 2, any letters and digits or also accepted.

A non-deterministic finite automaton (NFA) is one that has a choice of edges – labeled with the same symbol – to follow out of a state. Or it may have special edges labeled with (the Greek letter epsilon) that can be followed without eating any symbol from the put.

Below is an example of an NFA.

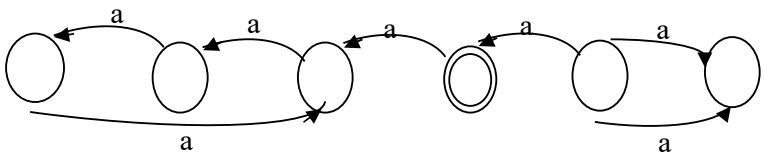

Fig. 5 Non-deterministic Finite Automata

In the start state, on input character a, the automaton can move either right or left. If left is chosen, then strings of a's whose length is a multiple of three will be accepted. If right is chosen, then even-length strings will be accepted. Thus, the language recognized by this NFA is the set of all strings of a's whose length is a multiple of two or three.

On the first transition, this machine must choose which way to go. It is required to accept the string if there is any choices of paths that will lead to acceptance. Thus, it must "guess" and must always guess correctly.

Edges labeled with ε may be taken without using up a symbol from the input.

DESIGN APPROACH

The design approach adopted here is very simple and easy to implement on computer. Implementing deterministic finite automat (DFA) as computer programs is easy. But implementing NFAs is a bit tough, since most computers do not have good "guessing" hardware. So this paper adopts the simplest approach. Here we are going to convert a regular grammar to finite automata.

CONVERSION FROM REGULAR GRAMMAR TO FINITE AUTOMATA

Given: G (V_N, V_T, P, S)

Out: M (K, V_T, δ, s, F)

Where;

(i) State of M is the variables of G plus additional state A not in V_N

(ii) Initial state of M is S

(iii)If P contains the production S ε ⟶

Then F = {S, A}

Otherwise F = {A}

(iv) State A is δ (B, a) if B a⟶

In addition, δ(B, a) contain all C such that

B ⟶ aC is in P

(v) δ(A, a) = ϕ for all a ε V_T

With the above information, we can convert a regular grammar to a DFA and subsequently implement it on a computer after converting it to a pseudo code.

e.g.

G: P: S ⟶ 0

B ⟶ 0B | 1s

B ⟶ 0

From the example above, we can obtain the following result:

V_T = {0, 1}

S = {S}

V_N = {S, B}

The finites state machine above can be represented using a transition table:

Table 2: A DFA transition table

STATE	0	1
S	A	-
B	A	S
A	-	-

SCANNER IMPLEMENTATION (AUTOMATIC)

A finite state scanner takes the form of a big loop. We can sketch it roughly as follows:

Repeat
Pick the next input character
Find the new state-table entry.
If this is a final state for some token,
Isolate the token; pass it to the parser,
And may be decrement character.
Until input used up.

Handling accepting states was done by a huge case statement with branches for all the final states.

Conversion of NFA to DFA

Implementing NFA on the computer is not a straight forward task, but it does not mean that we cannot implement NFA on a computer system. We are going to look for a way to convert NFA to DFA before we can successfully implement it on a computer system.

In this section we shall first show how to convert NFA's to DFA's. Then, we use the technique, known as "the subset construction," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation. The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states. After reading input $a_1a_2a_3...a_n$, the DFA is that state which corresponds to the set of states that the NFA can reach. From its start state, following paths labeled $a_1a_2a_3....a_n$, it is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA.

However, part of the power of the automaton-based approach to lexical analysis is that for real languages, the NFA and DFA have approximately the same number of states, and the exponential behaviour is not seen.

Algorithm: the subset construction of DFA from an NFA

OUTPUT: A DFA **D** accepting the same language as **N**

METHOD: Our algorithm constructs a transition table Dtran for **D**. each state of **D** is a set of NFA states, and we construct Dtran so **D** will simulate "in parallel" all possible moves **N** can make on a given input string. Our first problem is to deal with ε-transitions of **N** properly. In the table below, we see the definitions of several functions that describe basic computations on the states of **N** that are needed in the algorithm. Note that **S** is a single state of **N**, while **T** is a set of states of N.

Table 3: State Table

Operation	Description
ε-closure (s)	Set of NFA states reachable from NFA set ε-transitions only
ε-closure (T)	Set of NFA states reachable from some NFA sets in set T on ε-transitions only; U_s in Tε-closure (s).
Move (T,a)	Set of NFA states to which there is a transition on input symbol a from some state**s** T

Suppose the **start state** of the NFA is S, then the **start state** for its DFA is ε-closure (s), the final states of the DFA are those that include a **NFA-final-state.**

Algorithm: converting an NFA **N** into a DFA **D**……

Dstates= {**ε**-closure (s_0), s_0 is **N**'s start state}
Dstates are initially "unmarked"
While there is an unmarked D-state X **do** {
Mark X
For each a **in** S **do** {
T= {states reached from any s_i in X via **a**}
Y= ε-CLOSURE (T)
if Y **not in** Dstates **then** add Y to Dstates "unmarked"
Add transition from X to Y, labelled with **a**
}
}
We can also convert regular expressions directly to NFA and subsequently convert it to DFA so that it would be easily implemented on the computer using the algorithm above.

CONCLUSION

This paper has developed a strategy for efficient and faster scanner generator implementation. It accepts a LEX-like specification but produces a Java output file. The implementation shares neither code nor algorithm with any existing similar program. The tool does not attempt to implement the whole of the POSIX specification for LEX, however the program moved beyond lex in some areas, such as support for Unicode. The scanner produced by this scanner generator are thread safe, in that all scanner state is carried within the scanner instance.

REFERENCES

Aho A., Lam M., Sethi R and Ullman J., Compliers Principle techniques and Tools, Addison Wesley, 2007.
Elliot B., Jflex scanner generator manual, Princeton University Press, Princeton 2006
Hennessy, J.L. and D.A. patterson, Computer Organization and Design: The hardware/Software Interface, morgan-Kaufmann, San Francisco, CA, 2004.
Hopcroft, J.E., R. Motwani, and J.D. Ullman, Introduction to Automata Theory Language, and Computation, Addison-Wesley, Boston MA, 2006.
Huffman, D.A., "The synthesis of sequential machines," J. Franklin Inst. 257 (1954), pp 3-4, 161, 190, 275-303
John G., The Garden point LEX Scanner generator documentation, Queensland University of Technology Press, QueensLand, 2009.