International Journal of Scientific Research in Computer Science, Engineering and Information Technology
ISSN : 2456-3307 (www.ijsrcseit.com)
doi : https://doi.org/10.32628/CSEIT21761

Layman-Friendly Regional Language Unit Conversion Compiler

Aditya Bhanwadiya*, Gautam Rizwani, Darshil Shah
Department of Information Technology, Dharmsinh Desai University, Nadiad, Gujarat, India

Article Info
Volume 7, Issue 6
Page Number: 38-46
Publication Issue :
November-December-2021
Article History
Accepted : 05 Nov 2021
Published : 14 Nov 2021

ABSTRACT

A compiler converts source language code to machine-understandable code. This entire translation of code happens in different stages. So, one can define the compiler as a collection of many phases or stages, where every phase performs a single task and the code is translated. This paper is about brief information of the compiler on how the language or source code is evaluated and translated, from which phase what information is extracted in order to generate target code as output. For better clarity, an example of an easy-to-understand language is taken, all steps are explained, and a compiler is designed using FLEX and YACC.

Keywords : Compiler, lexical, syntactic, semantic, FLEX, YACC

I. INTRODUCTION

As a normal human being or a software developer, one can understand high-level programming languages such as C/C++/Java/Python etc. As such languages have various English words such as for, while, if-else, etc. for underlying concepts of loops, conditional statements and so on. However, a computer cannot understand such words. The only thing it understands is binary code i.e., 0 or 1. So, in order to run a program, we need to first convert our program to machine understandable code. To do that, we need to pass our code through a special program called a Compiler.

Fig :1. Introduction to Compiler

So, there are two versions of a program now:

- The one written in a higher language which we can understand.
- The second one which is converted and only the machine can understand.

So, the compiler is a complex machine which bridges the gap between human readable code and computer readable code.

II. EXAMPLE DEFINITION

The valid sentences which should be accepted and results should be displayed are:

1. 23 kilograms is how many grams?
2. 40 kilo is how many litres?
3. how many litres is 1 kilogram?
4. How many kilo is one gram?

Above sentences in regional language (Gujarati) is also valid and are as follows:

1. 23 kilograms etle ketla grams?

2. 40 kilo etle ketla litres?
3. ketla litres etle 1 kilogram?
4. Ketla kilo etle ek gram?

So, the above sentences with the mentioned units are valid.

III. ALGORITHM OF A COMPILER

Algorithm of compiler denotes the activity of compiler to translate the high-level language to machine executable language. There is not any single phase for translating the high-level language. It passes through several phases where the language is processed as per their phase-wise rules. By combining all of them we can propose an algorithm for the compiler to work on. The algorithm can be like as shown in Table 1.

IV. PHASES OF COMPILER

For our defined language, it will be like as shown in Table 2 :

A. Lexical Analysis

Phase one of compiler construction is coined as scanning or lexical analysis. A lexical analyzer, also known as the lexer, is a pattern recognizer engine simulated by mathematical computational model known as Finite-State Machine (FSM) or Finite-State Automaton (FSA) that reads a string of individual characters as its input in the source program and clusters read characters into meaningful sequences called lexemes by matching with the token pattern and produces stream of tokens. A token is a sequence of characters having a collective meaning and they are basic units of the programming language, that describes the class or category of input string such as keywords, identifiers, units, literal strings, constants, operators, and punctuation symbols.

Table 1: Algorithm of a Compiler

Name of the Phase	Steps of Phase
Lexical Analyzer	Step - 1: Taking input as Character Stream. Step - 2: Reading the character until the next token. Step - 3: Produce output as token
Syntax Analyzer	Step - 4: Taking input as a token. Step - 5: Pass it to the Syntax analyser, and syntax analyser will create a syntax tree for the token. Step – 6: Produce a syntax tree as output.
Semantic Analyzer	Step - 7: Taking input the syntax tree of a token. Step - 8: Check whether the tree is semantically correct or not. Step - 9: Produce a semantically correct syntax tree by type checking, level checking, flow control checking.
Intermediate Code Generation	Step - 10: Take input from the syntax tree. Step - 11: Produce intermediate code step by step.
Code Optimization	Step - 12: Take input from intermediate code. Step - 13: Minimize the long code to short. Step - 14: The temporary location also reduced here
Code Generation	Step - 15: Take input from code optimizer as optimized code. Step - 16: Process the task by some specialized instructions. Step - 17: Get the targeted machine code.

Table 2 : Separated Tokens of defined language

Keywords	Units	Constants	Special

			Characters
how	kilogram	23, 40, 1 etc.	?
How	kilograms	One, two, hundred, thousand, etc.	
many	kilo	ek, be, so, hajar etc.	
is	grams		
etle	gram		
ketla	litres		
Ketla			

For every recognized lexeme, a token is represented and generated by a pair, <*token-type and token-value*>, is an attribute for the token. Here, the token-type refers to an abstract symbol of the token to be used in the syntax analysis process of compilation and the token-value is a pointer variable to the symbol table entry, in which the token information is stored. For our definition, it is shown in Table 3.

Table 3 : Token Table

Lexemes	**Token Name**	**Token Value**
how	KW	1
How	KW	2
many	KW	3
is	KW	4
etle	KW	5
ketla	KW	6
Ketla	KW	7
kilogram	UT	1
kilograms	UT	2
kilo	UT	3
grams	UT	4
gram	UT	5
litres	UT	6
25,40,1 etc	CO	1
one, two, hundred, thousand etc.	CO	2
ek, be, tran etc.	CO	3
?	SC	1
delim	-	-

Apart from token recognition, lexical analyzer also performs following tasks:

- Removes white spaces and comments from the source program.
- Correlates error messages with the source program.
- Read input characters from the source program.

Lex tool takes a set of formal description of tokens in the form of regular expression and produces a C program lex.yy.c which we call lexical analyzer or lexer that can identify these tokens. The process of identifying tokens is called lexical analysis or lexing. The set of rules or descriptions given to lex is called a lex specification which contains two parts: (1) patterns and (2) corresponding actions. Lex tool automatically converts the lex specification into c statements into a file containing a C subroutine called yylex().

There are three sections of a FLEX code:

1. Definition Section

The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in “%{ %}” brackets. Anything written in this brackets is copied directly to the file lex.yy.c

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval; %}
```

2. Rules Section

The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begins on the same line in {} brackets. The rule section is enclosed in “%% %%”.

```
%%
[0-9]+ {yylval=atoi(yytext); return NUM; } "is how
many"|"how many"|"How many"|"etle
ketla"|"Ketla"|"ketla"|"etle" {return KEYQUE;}
"kilograms"|"kilo"|"kilogram?"
{return UNITKG;} "grams?"|"gram?" {return
UNITGM;} "litres?"|"litres" {return UNITLIT;}
"is" {return IS;}

"one"|"ek" { yylval = 1; return ONE;} "twenty"|"vees"
{yylval = 20; return TWENTY;}
"thirteen"|"ter" {yylval = 13; return THIRTEEN;}
"hundred"|"so" {yylval = 100; return HUNDRED;}
"thousand"|"hajar" {yylval = 1000; return
THOUSAND;}
[\t] {;}
        [\n] {return 0;}
        . {return yytext[0];}
        %%
```

1. User code section

This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

```
int yywrap()
{
return 1;
}
```

B. Syntax Analysis

Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens that reflects the structure of the text, typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure. A tree structure where the leaves are the token found by the lexical analysis. And if the leaves are read from left to right, the sequence is the same as in the input text. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting *syntax errors*. The two major types of parsers employed are:

- **Top - Down parser**

Top-Down parsers are constructed from the grammar which is free from ambiguity and left recursion. It uses leftmost derivation to construct a parse tree. It allows a grammar which is free from Left Factoring.

- **Bottom-Up parser**

A bottom-up parser builds the parse tree from the bottom to the top. Bottom-up parsers make much less extravagant predictions and can handle grammars that top-down parsers cannot. Although a bottom-up parser reads the sequence of tokens from left to right, it builds the parse tree from right to left. A bottom-up parser can be thought of as creating a rightmost derivation.

It is further divided into various different parsers as shown:

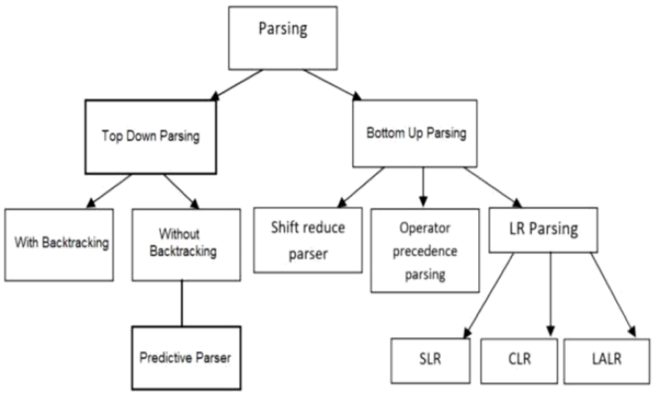


Fig : 2. Types of Parsers

We have used YACC(Yet Another Compiler-Compiler) which is a LALR(1)(LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. So, for our definition, we have used a bottom-up parsing approach.

There are three different parts in the input file of YACC, all are separated by %%, as follows :

1. Definition Part

The definition part includes information about the tokens used in the syntax definition. It also consists of token declarations and C code bracketed by "%{" and "%}".

```
%{
        #include<stdio.h>
        %}
        %token
        NUM %token
        UNITKG %token
        KEYQUE %token
        UNITGM %token
        UNITLIT %token
        IS %token
        ONE %token
        TWENTY %token
        THIRTEEN %token
        HUNDRED %token
        THOUSAND
```

2. Rule Part

The second section of the Bison file consists of the context-free grammar for the language. Productions are separated by semicolons, the "::=" symbol of the BNF is

the parser. There must also be the function yyerror() which is used to report on errors during the parse.

```
void main()
{
printf("Layman-Friendly Regional Unit Conversion Compiler \n");
printf("NOTE :: If string will be valid then output will be displayed else error will be shown. \n \n");
yyparse();
}
void yyerror()
{
printf("Please enter valid values. \n");
}
```

C. Semantic Analysis

Semantic Analysis is the third phase of the compiler. Semantic Analysis makes sure that declarations and statements of a program are semantically correct. Both the syntax tree of the previous phase and symbol table are used to check the consistency of the given code.

Following are the major functionalities of semantic analysis:

1. Type Checking

It ensures that data types are used in a way such that they're consistent with their definition.

2. Label Checking

Labels references in a program must exist

3. Flow Control Check

It keeps a check that control structures are used in a proper manner such as no break statement outside a loop etc.

Few of the errors that are recognized by semantic analyzer includes:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variables in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Apart from these, semantic analyzer also helps to store the type information gathered and save it in symbol table or syntax tree. In case of any type mismatch, it will show the error. It also checks if the source language permits the operand or not.

D. Intermediate Code Generation

What is the need of Intermediate code generation? Let's say we have *n* different programming languages and *m* different types of machine. If we want to execute *n* different programming languages on m different machines, then *n*m* compilers need to be implemented. The implementation of *n*m* compiling is not an easy task. The above problem can be transform into *n + m* compiler by introducing a new language IR, known as *intermediate representation*

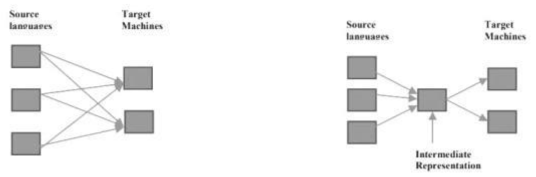


Fig : 3. Intermediate Code Generation

There are different ways of intermediate code representation:

- **Postfix Notation**

Postfix Notation is also known as "Reverse Polish Notation". The application of an operator *op* to sub-expressions *E1* and *E2* is written in postfix notation as *E1 E2 op*. Postfix notation can be mechanically evaluated with the help of stack data structure.

- **Syntax Tree**

Syntax tree is a reduced form of parse tree, which is useful for representing language constructs. It shows the structure of a program by abstracting away irrelevant details from a parse tree. So, a parent node represents a computation to be performed and the child node represents what that computation is performed on.

- **Directed - Acyclic Graph (DAG)**

The data structure with more than one path from starting symbol to terminals is called Directed Acyclic Graph (DAG). DAG gives information as a syntax tree but in a more compact way. DAG has nodes for every sub-expression of the expression. An interior node represents an operator and its children represent an operand. DAGs are useful in optimizing the code by eliminating the sub-expressions and duplicate codes.

4. Three Address Code

The three address code statements are represented in the form *a= b op c a*, *b* and *c* are the variables and will have memory locations (address) and *op* is the operator.

For example: the three address code representation for the expression *x + y * z + s* :

*T1 = y * z T2=*
x + T1
T3= T2 + s , where *T1*, *T2* and *T3* are the temporary variables.

There are three representations of 3-Address codes namely:

- Quadruple
- Triples
- Indirect Triples

E. Code Optimization

Code Optimization is the process of transforming a piece of source code to produce more efficient target code. Efficiency is measured both in terms of time and space. Most of the optimization techniques attempt to improve the target code by eliminating unnecessary instructions in the object code, or by replacing one sequence of instructions by another faster sequence of instructions. Code optimization may either be performed on the intermediate representation of the source code or on the unoptimized version of the target machine code. If applied on the intermediate representation, the code optimization phase will reduce the size of the Abstract Syntax Tree or the Three Address Code instructions. Otherwise, if it is applied as part of final code generation, the code optimization phase attempts to choose which instructions to emit, how to allocate registers and when to spill, and so on.

The optimization process can be broadly classified into two types :

1. **Machine Independent Optimization** – This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization** – Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy. replaced with ":", the empty production is left empty, non-terminals are written in all lower case, and the multi character terminal symbols in all upper case. Within the braces for the action associated with a production is just ordinary C code. If no action is present, the parser will take no action upon reducing that production.

```
ADI: A{
printf("Answer is: %d \n",$$);
return 0;
};
|B{
printf("Answer is: %f \n",(float)$$/1000);
return 0;
};
A:NUM' 'UNITKG' 'KEYQUE' 'UNITGM
{$$=($1*1000);}
|NUM' 'UNITKG' 'KEYQUE' 'UNITLIT {$$ = $1;}
|KEYQUE' 'UNITLIT' 'IS' 'NUM' 'UNITKG
{$$=$7;}
|KEYQUE' 'UNITLIT' 'KEYQUE' 'NUM'
'UNITKG {$$=$7;}
B:KEYQUE' 'UNITKG' 'IS' 'ONE'
'UNITGM {$$=$7;}
|KEYQUE' 'UNITKG' 'IS' 'THIRTEEN'
'UNITGM {$$=$7;}
|KEYQUE' 'UNITKG' 'KEYQUE' 'ONE'
'UNITGM {$$=$7;}
|KEYQUE' 'UNITKG' 'IS' 'TWENTY'
'UNITGM {$$=$7;}
|KEYQUE' 'UNITKG' 'KEYQUE' 'TWENTY'
'UNITGM {$$=$7;}
|KEYQUE' 'UNITKG' 'IS' 'ONE' 'HUNDRED'
'UNITGM {$$=$7*$9;} |KEYQUE' 'UNITKG'
'KEYQUE' 'ONE' 'HUNDRED' 'UNITGM {$$=$7*$9;}
|KEYQUE' 'UNITKG' 'IS' 'TWENTY' 'THOUSAND'
'UNITGM {$$=$7*$9;} |KEYQUE' 'UNITKG'
'KEYQUE' 'TWENTY' 'THOUSAND' 'UNITGM
{$$=$7*$9;}
```

3. Auxiliary Routine Part

The third section of the Yacc file consists of C code. There must be a main() routine which calls the function yyparse(). The function yyparse() is the driver routine for Code Optimization can be done in following different ways:

- Compile time evaluation
- Variable Propagation
- Dead-code elimination
- Code motion
- Induction Variable and Strength Reduction

F. Code Generation

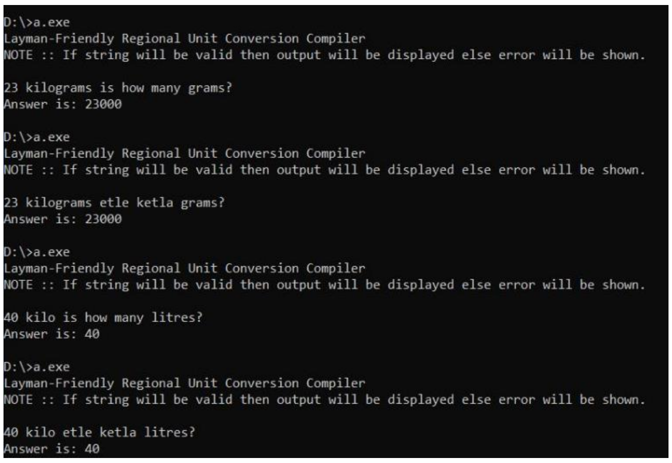
Code generation is the last and final phase of a compiler. Target code generation deals with assembly language to convert optimized code into machine understandable format. Target code can be machine readable code or assembly code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The objective of this phase is to allocate storage and generate relocatable machine code.

Properties desired by the code generation phase are mentioned below.

- Correctness
- High Quality
- Quick Code Generation
- Efficient use of resources of target machine

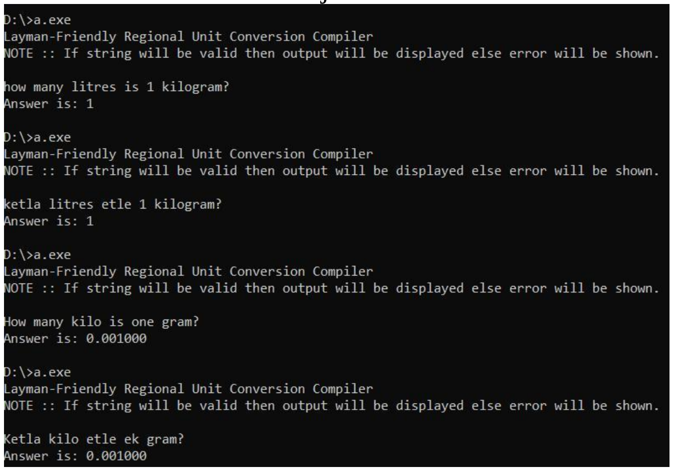
Target code which is now low level language goes into linker and loader.

V. RESULTS AND OUTCOMES



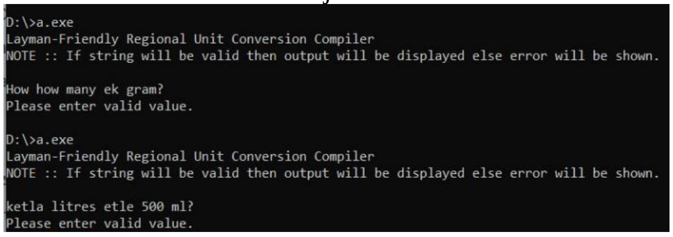
```
D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

23 kilograms is how many grams?
Answer is: 23000

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

23 kilograms etle ketla grams?
Answer is: 23000

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

40 kilo is how many litres?
Answer is: 40

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

40 kilo etle ketla litres?
Answer is: 40
```

Fig : 4. Output of valid sentences in English and Gujarati



```
D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

how many litres is 1 kilogram?
Answer is: 1

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

ketla litres etle 1 kilogram?
Answer is: 1

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

How many kilo is one gram?
Answer is: 0.001000

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

Ketla kilo etle ek gram?
Answer is: 0.001000
```

Fig : 5. Output of few more valid sentences in English and Gujarati



```
D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

How how many ek gram?
Please enter valid value.

D:\>a.exe
Layman-Friendly Regional Unit Conversion Compiler
NOTE :: If string will be valid then output will be displayed else error will be shown.

ketla litres etle 500 ml?
Please enter valid value.
```

Fig : 6. Output of some invalid sentences

V. CONCLUSION

Throughout the process of translation, code written in high-level language passes through various phases and as a result, is converted to machine-understandable code but the original meaning of code never changes. So basically, it's like a language processing system. Entire procedure is divided into two parts, frontend and backend. Front-end part of the compiler includes lexical, syntactic and semantic phases of translation. On the other hand, the back-end part consists of intermediate code generation, code optimization and target code generation. We have also presented our own definition which is quite simple but easy to understand and tried to construct a compiler using FLEX and YACC tools.

VI. REFERENCES

[1]. Vishal Trivedi. 2018. International Journal of Creative Research Thoughts. (Jan 2018), ISSN NO: 2320-2882

[2]. Md. Alomgir Hossain, Rihab Rahman, Md. Hasibul Islam, Mahabub Azam.2019.American Journal of Engineering Research. (Dec 2019), e-ISSN NO: 2320-0847

[3]. Nisha N. Shirvi, Mahesh H. Panchal.2014. International Journal of Computer Science and Mobile Computing. (Feb 2014), ISSN 2320–088X

[4]. Vaikunta Pai T., A. Jayanthila Devi, P. S. Aithal. 2020.International Journal of Applied Engineering and Management Letters. (Dec 2020), ISSN: 2581-7000

[5]. T.Æ. Mogensen. 2011. Springer-Verlag London Limited. DOI 10.1007/978-0-85729-829-4_2

[6]. Neha Bhateja, Nishu Sethi. 2018. Journal of Emerging Technologies and Innovative Research. (June 2018), ISSN:2349-5162

[7]. Anjan Kumar Sarma. 2015. International Journal of Computer Applications. (Dec 2015), ISSN NO: 0975 - 8887

[8]. John Smit, Lexical Analysis (Analyzer) in Compiler Design with Example, Nov. 2021.Online]. Available:https://www.guru99.com/compiler-design-lexical-analysis.html

[9]. Shivani Mittal, Flex (Fast Lexical Analyzer Generator,Aug.2021.Online].Available:https:// www.geeksforgeeks.org/flex-fast-lexical-analyzer-generat or/

[10]. Sanjay Monu, Classification of Top Down Parsers,

Nov.2019.Online].Available:https://www.geeksf org eeks.org/classification-of-top-down-parsers/

[11]. “Bottom-UpParsing”,Online]. Available:http://www.cs.ecu.edu/karl/5220/spr1 6/No tes/Parsing/bottomup.html

[12]. Thakur Aman, Introduction to YACC, April. 2021.Online].Available:https://www.geeksforge eks. org/introduction-to-yacc/

[13]. John Smit, Syntax Analysis: Compiler Top Down & Bottom Up Parsing Types, Oct. 2021.Online]. Available:https://www.guru99.com/syntax-analysis-p arsing-types.html

[14]. “Compiler Design Semantic Analysis-Compiler-Design”.Online].Available:https://www.w isdomjobs.com/e-university/compiler-design-tutorial -1144/compiler-design-semantic-analysis-25305.htm l

[15]. Palak Singhal, Semantic Analysis in Compiler Design,April.2020.Online].Available:https://ww w.geeksforgeeks.org/semantic-analysis-in-compiler-design/

[16]. John Smit, Phases of Compiler with Example: Compilation Process & Steps, Oct. 2021.Online]. Available:https://www.guru99.com/compiler-design-phases-of-compiler.html#4

[17]. C.Naga Raju, Intermediate Code Generation, June. 2020.Online].Available:https://www.jntua.ac.in /gate -online-classes/registration/downloads/material/a159 254722029.pdf

[18]. “Code Optimization in Compiler Design”, July.2020.Online].Available:https://www.geeksf orge eks.org/code-optimization-in-compiler-design/

[19]. Tom Niemann. “Lex And Yacc Tutorial” Online]. Available:https://cse.iitkgp.ac.in/~bivasm/notes/ Lex AndYaccTutorial.pdf

Cite this article as :

Aditya Bhanwadiya, Gautam Rizwani, Darshil Shah, "Layman-Friendly Regional Language Unit Conversion Compiler", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 7 Issue 6, pp. 38-46, November-December 2021. Available at doi : https://doi.org/10.32628/CSEIT21761
Journal URL : https://ijsrcseit.com/CSEIT21761