ISSN:0975-9646

Design and Implementation of a Customized Compiler

Govind Prasad Arya, Neha Sohail, Pallavi Ranjan, Priya Kumari & Shabina Khatoon
Department of Computer Science & Engineering, Shivalik College of Engineering
Affiliated to Uttarakhand Technical University, Dehradun

***Abstract*— DFA (Deterministic Finite Automata) is designed for the set of customized tokens that we have taken. A grammar in compiler is a set rule that specify how sentences can be structured with the terminals, non-terminals, the set of productions and the start symbol. CFG (Context Free Grammar) is used in compiler for parsing. This paper presents the steps to convert a high level language written according to our customization into assembly language. It also presents what is a compiler, its phases and functions. Basically, the compiler passes through the six phases but here only the implementation through three phases are shown, i.e Lexical Analyzer, Syntax Analyzer and Target Code Generation. The Target Code generated here is in Assembly Language.**

***Index Terms*— Compiler, phases of compiler, DFA, CFG, Assembly Code Generation**

I. INTRODUCTION

A compiler is a program that converts a source program written in high level programming language into target program which is machine understandable language. The most common reason for converting source code is to create an executable program. Generally, the target program is an executable program that can be used by the user to process the input and produce the related output. The compiler works with two prime features of the language **syntax and semantics**. If the compiled program can run on a computer whose CPU or operating system is different from the one on which compiler runs, the compiler is known as a **cross-compiler**. More generally, compilers are the specific type of translator. A program that translates from a low level language to a higher level language is a decompiler.

II. PHASES OF COMPILER

Each phase transforms the source program from one representation into another representation. They communicate with error handlers and symbol table. But in this customized compiler, we are going through only necessary three phases of the compiler i.e. **Lexical Phase, Syntax Phase and Target Code Generation**.

There are six phases of compiler.

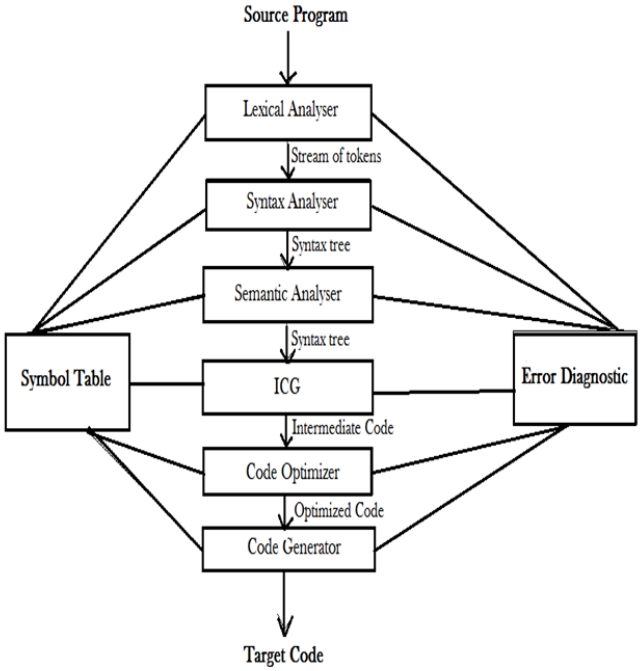


Fig. 2 Phases of Compiler

Phase I (Lexical Analyzer) - This phase reads the source code as a stream of characters and converts it into meaningful lexemes. A token describes a pattern of characters having same meaning in the source program (such as identifiers, operators, keywords, and numbers).

Phase II (Syntax Analyzer) **–** This phase generates the syntax tree according to the already known CFG. A syntax analyzer is also called a parser. A parse tree describes the syntactic structure.

Phase III (Semantic Analyzer) – This phase checks the source program for semantic errors and collects the type of information for the code generation. The main functionality is **type checking**.

Phase IV (ICG) – ICG stands for Intermediate Code Generator. After semantic analysis intermediate code is generated, which is in between high level language and machine language. These codes are generally machine architecture independent, but the level of intermediates code is close to the level of machine codes.

Phase V(Code Optimizer) – This phase removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, Memory).

Phase VI (Code Generation) – In this phase the code generator takes the optimized representation of intermediate code and converts it into machine understandable code (Target Code).

A. Abbreviations and Acronyms

CFG -Context Free Grammar
ICG -Intermediate Code Generator
DFA -Deterministic Finite Automata
LMD-Left Most Derivative
RMD-Right Most Derivative

III. DESIGNING OF LEXICAL ANALYZER

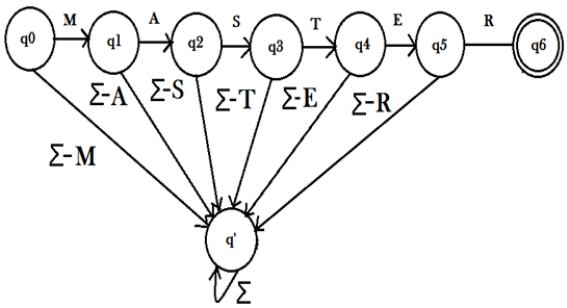
Steps for designing the customized compiler:

i. Draw the DFA (Deterministic Finite Automata) for the tokens, digits and all other customized units taken in our compiler.
ii. Combine all the individual DFAs into one single DFA.
iii. Implement the code of the Lexical Analyzer as per the combined DFA.
iv. Now, write the grammar (Context Free Grammar) for the syntax analyzer.
v. Check and parse the grammar as dry run for the tokens.
vi. Implement the code of the Syntax Analyzer as per the grammar.
vii. Study the assembly language**.**
viii. Implement the target code for the given program**.**
ix. The target code is in assembly language.

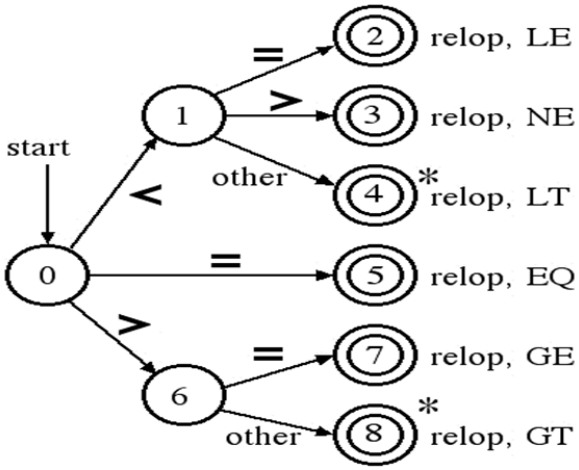
DFA

Following are the few DFAs for our customized compiler as keywords, operators, special symbols, digits and identifiers.

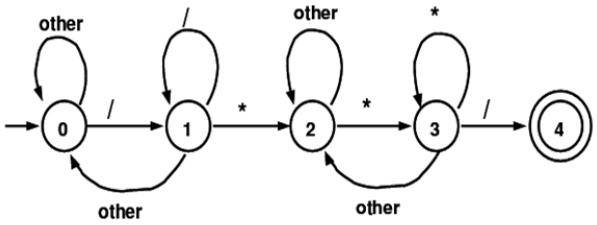
A. DFA for Keyword



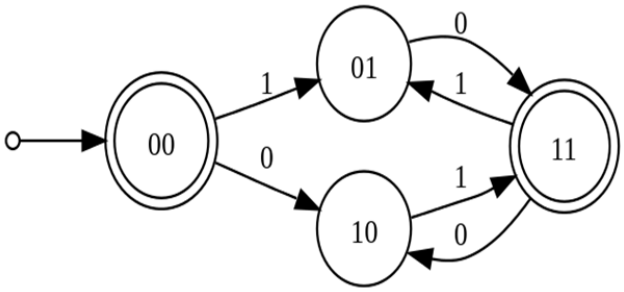
B. DFA for Operator



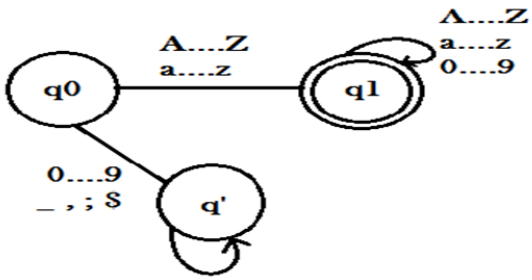
C. DFA for Special Symbol



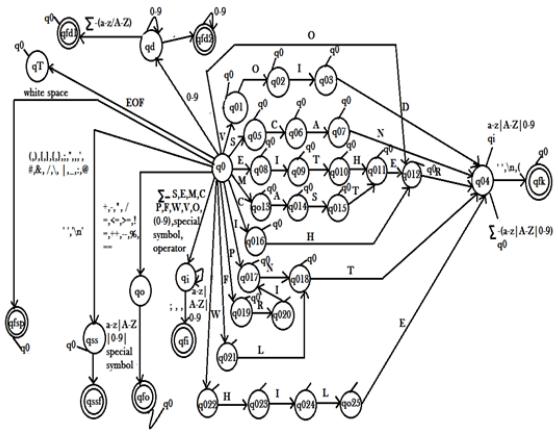
D. DFA for Digits



E. DFA for Identifiers



F. Combined DFA



IV. PARSING

There are two types of Parsing Approaches.

1. Top down Parsing

Top down parsing starts from the root of the parse tree. It is labeled with the start symbol of the grammar. The reason why top down parsing seeks LMD (Left Most Derivation) for an input string **s** and not the RMD (Right Most Derivation) is that the input string **s** is scanned by the parser from left to right, one symbol/token at a time, and the LMD generate the leaves of the parse tree in left to right order, which matches the input scan order.

There are two types of Top down Parsing-

a. Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking.

b. Predictive Parsing

Predictive parser is a recursive descent parser, which has the capability to predict which production should be used to replace the input string. The predictive parser does not suffer from the issue of backtracking. The parser refers to the parsing table to take any decision on the input.

2. Bottom up Parsing

It can be defined as an attempt to reduce the input string **s** to the start symbol of the grammar by tracing out the RMD of **s** in reverse. This is equivalent to constructing a parse tree for the input string **s** by starting with the leaves and proceeding towards the root i.e., attempting to construct the parse tree from the bottom up. The reason why Bottom up Parsing traces out the RMD of an input string **s** in reverse and not the LMD is because the parser scans the input string s from left to right, one symbol/token at a time.

V. GRAMMAR DESIGN FOR PARSING

We have used one of the Top Down Parsing Technique i.e. **Recursive Descent Parsing** for the **Context Free Grammar.**

The context free grammar is written for our compiler as follows:

The starting symbol is M (non terminal), with the productions as follows:

M → MASTER () {S}
S → DS |DCS |LCS |BCS |IOS |IDS |E |FC |empty
DS → DT sp id DS1|id DS1
DT → INT |CHR |FLT
DS1 → =DS2 |,DS1 |id DS1|;
DS2 → 0DS2 |1DS2 |2DS2 |3DS2 |4DS2 |5DS2 |6DS2 |7DS2 |8DS2 |9DS2 |'id' DS1 |.DS2 |DS1
DCS → EITHER (CE) {S}T
CE → id Orel E' E"
Orel → <= |>= |!= |== |< |> |% E' Orel
E' → 0E' |1E' |2E' |3E' |4E' |5E' |6E' |7E' |8E' |9E' |id
E" → Olog CE | empty
Olog → || | &&

T → OR{S} | empty
LCS → WHILE (CE){S}
BCS → id (A){S}
A → DT sp id A |,A |empty
FC → id (A1);
A1 → idA1 |,A1 |0A1 |1A1 |2A1 |3A1 |4A1 |5A1 |6A1 |7A1 |8A1 |9A1 |empty
IOS → PRINT (" "P); |SCAN("FS"SC);
P → id P |empty
FS → %d FS |%c FS |%f FS |empty
SC → ,&id SC |empty
IDS → id Oid;
Oid → ++|--
E → id=YZ;
Y → D Op id Y |id Op D Y |id |D |D Op D Y |id Op id Y
Z → Op id Y| Op D Y |empty
D → 0 |1 |2 |3 |4 |5 |6 |7 |8 |9
Op → + |- |/ |* |%

Where;

M - Start symbol
S - Statement
DS - Declarative statement
DCS - Decision control statement
LCS - Loop control statement
BCS - Branch control statement
IOS - Input output statement
IDS - Increment/Decrement statement
FC - Function call
E - Expression
DT - Data type
DS1 - Declarative statement1
DS2 - Declarative statement2
CE - Conditional expression
Orel - Relational operator
Olog - Logical operator
E′ - Expression1
E′′ - Expression2
T - Terminal
A - Argument list
A1 - Function call argument list (call by value)
P - Printing the value of anything
FS - Format specifier
SC - Scanning the identifier
Oid - Increment/ Decrement operator
Y - Any type of valid expression
Z - Continuation of the expression if it is long
D - Digits
Op - Arithmetic operator

VI. Working Of Our Compiler

The working of the compiler was divided into three phases:

A. Lexical Phase

The implementation of the Lexical Phase started with the designing of the DFAs for the tokens and then combining them into a single DFA. Then according to the DFA, we designed procedures for each non-terminals and used recursion for its implementation.

After successfully running the code, the output will display the result like-

Valid keyword **MASTER** found
Valid digit **23** found
Valid special symbol **{** found
Valid operator **+** found
Valid identifier **d** found
As according to the tokens written in the program file.

B. Syntax Phase

After Completion of Lexical phase we started to work on the grammar. We wrote our own Context Free Grammar which is in the form of LL (1) grammar.

Then we did parsing of the input symbols in the text file one by one as a testing phase. Later on, according to the non-terminals in the grammar, we wrote the program code using various functions in place of non-terminals recursively.

The successful completion of the syntax phase will yield-
Closing found, Syntactically Correct

C. Target Code Generation

Finally, after completion of the first two phases, the tokens were accepted as valid tokens and results in syntactically correct output. Then we moved to the final phase of the compiler. The target code was generated in assembly language.

VII. TESTING

The following example is shown with the help of two files. One is the input file and the other is the target file. The source code is written in the **input file** which has to be converted into the assembly language and the output is being displayed in the **target file**.

```
MASTER ()
{
INT a=2, b=4, c;
c=a*b;
}
```

Fig. 7.1 Input File

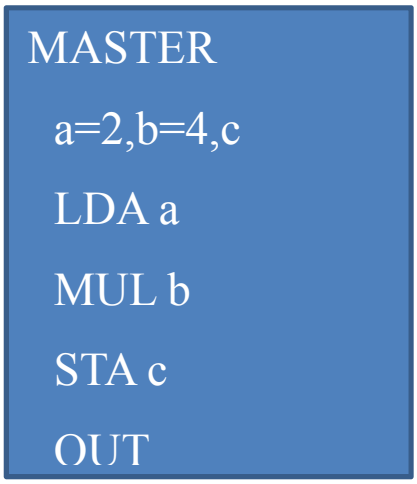


Fig. 7.2 Target File

VIII. CONCLUSION

Here we conclude from the above example that the source code in the high level language in the input file is successfully being converted to assembly language in the target file after passing through all the three phases. This set of implementation is fixed for an expression. It can further be extended and can be made flexible for any type of source code.

ACKNOWLEDGMENT

It is a great pleasure to express our profound sense of gratitude and reverence to our all through guide and teacher **Mr. Govind Prasad Arya, Assistant. Professor, Department of Computer Science & Engineering, Shivalik College of Engineering, Dehradun, Uttarakhand.** He was always a source of encouragement and inspiration, and constantly guided us for the accomplishment of this task with meticulous care. We owe to him the most, to have had the opportunity to accomplish the work under his guidance.

REFERENCES

[1] Jens Nilsson, Proceedings of the 11th International Conference on Parsing Technologies (IWPT), Paris, October 2009 c Association for Computational Linguistics.

[2] Stephen G. Pulman University of Cambridge Computer Laboratory, and SRI International, Cambridge April 1991: To appear in Encyclopedia of Linguistics, Pergamon Press and Aberdeen University Press.

[3] J.H Edmondson P. Rubinfeld R. Preston and Rajagopalan, Superscalar instruction execution in the 21164 Alpha microprocessor "IEEE micro pp. 33-43 April 1995.

[4] Gepard DSP core, Austria Mikro Systeme International 2000 Online on http://asic.amsint.com/databooks/digital/gepard.html.

[5] A.V. Aho M.Ganapathi and S.W.K Tjiang "code Generation using tree matching and dynamic Programming". ACM Trans Program Lang. Syste.Vol 11, no. 4, pp, 4911-516 2009.

[6] About Ghazaleh N, Childres B, Mosse D, Melhem R, Craven M. Collaborative Compiler OS power management for applications. TR-02-103, 2002 http://www.cs.pitt.edu compiler design.

[7] Azevedo A Issenin I, Cornea R, Gupta R, Dutt N, Veidenbaum A, Nicolau A dyamic voltage scheduling using Program Design automation and test in Europe 2005.

[8] Rajan, A; Joshi, B.K.; Rawat, A; Jha, R.; Bhachavat, K., "Analysis of process distribution in HPC cluster using HPL." 2nd IEEE International Conference on Parallel Distributed and Grid Computing(PDGC), 2012, pp.85,88, 6-8 Dec. 2012 Solan India.

[9] J. Cohen, Stuart Kolodner; "Estimating the Speed up in Parallel Parsing"; IEEE Transactions on Software Engineering, January 1985.

[10] M. Chandwani, M. Puranik , N.S. Chaudhari, "On CKY- Parsing of Context Free Grammars in Parallel"; Proceedings of the IEEE Region 10 Conference, Tencon 92, Melbourne Australia, pp. 141- 145, 1992.

[11] Valeriy Shipunov, Andrey Gavryushenko, Eugene Kuznetsov," Comparative Analysis of Debugging Tools in Parallel Programming for Multi-core Processors" CADSM'2007, February 20-24, 2007, Polyana, UKRAINE IEEE.

[12] Mary Hall, David Padua and Keshav Pingali,"Compiler Research:The Next 50 Years", Communication of the ACM Feb 2009,Vol. 2.

[13] Amit Barve and Dr. Brijendra Kumar Joshi;"A Parallel Lexical Analyzer for Multi-core Machine"; Proceeding of CONSEG-2012,CSI 6th International confernece on software engineering; pp 319-323;5-7 September 2012 Indore,India.

[14] Amit Barve and Brijendrakumar Joshi, "Parallel lexical analysis on multi-core machines using divide and conquer," NUiCONE- 2012 Nirma University International Conference on Engineering , pp.1,5, 6-8 Dec. 2012. Ahmedabad, India.

[15] Amit Barve and Brijendrakumar Joshi; "Parallel lexical analysis of multiple files on multi-core machines"; International Journal of Computer Applications; Vol. 96, No.8, June 2014.

[16] David R. Butenhof, "Programming with POSIX Threads", Addison-Wesley Longman Publishing Co., USA 1997.

[17] Rajan, A; Joshi, B.K.; Rawat, A; Jha, R.; Bhachavat, K., "Analysis of process distribution in HPC cluster using HPL," 2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC), 2012, pp.85,88, 6-8 Dec. 2012 Solan India.

[18] Rajan A., Joshi B.K., Rawat A., Gupta S."Analyitical Study of HPCC Performance Using HPL";International Journal of Computer Science and its Applications, Vol. 2, no. 1, p. 47-49, Apr. 2012.

[19] Rajan A., Joshi Brijendra Kumar, Rawat A."Critical Analysis of HPL Performance under Different Process Distribution Patterns".CSI 6th International Conference on Software Engineering (CONSEG- 2012), DAVV, Indore, Sep., 5-7, 2012

[20] Chuanpeng Li, Chen Ding, Kai Shen;"Quantifying the cost of context switch",ExpCS'07' Proceeding of the 2007 workshop on Experimental computer science; article 2; ACM New York USA;2007.