D Y PATIL
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

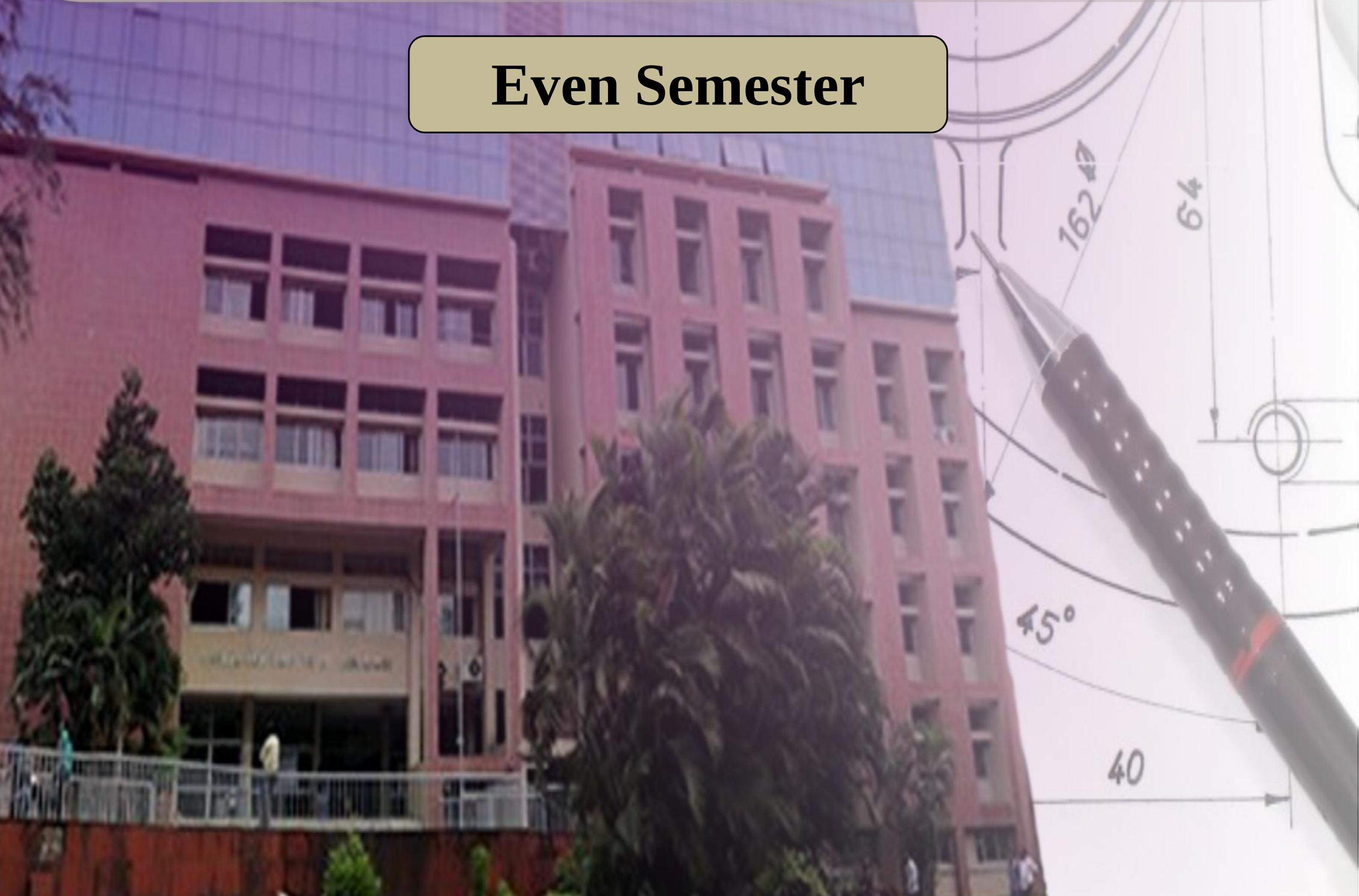
Department of Computer Engineering

Lab Manual

Third Year Semester-VI

Subject: System Software Lab

Even Semester



Institutional Vision, Mission and Quality Policy

Our Vision

To foster and permeate higher and quality education with value added engineering, technology programs, providing all facilities in terms of technology and platforms for all round development with societal awareness and nurture the youth with international competencies and exemplary level of employability even under highly competitive environment so that they are innovative adaptable and capable of handling problems faced by our country and world at large.

RAIT's firm belief in new form of engineering education that lays equal stress on academics and leadership building extracurricular skills has been a major contribution to the success of RAIT as one of the most reputed institution of higher learning. The challenges faced by our country and world in the 21 Century needs a whole new range of thought and action leaders, which a conventional educational system in engineering disciplines are ill equipped to produce. Our reputation in providing good engineering education with additional life skills ensure that high grade and highly motivated students join us. Our laboratories and practical sessions reflect the latest that is being followed in the Industry. The project works and summer projects make our students adept at handling the real life problems and be Industry ready. Our students are well placed in the Industry and their performance makes reputed companies visit us with renewed demands and vigour.

Our Mission

The Institution is committed to mobilize the resources and equip itself with men and materials of excellence thereby ensuring that the Institution becomes pivotal center of service to Industry, academia, and society with the latest technology. RAIT engages different platforms such as technology enhancing Student Technical Societies, Cultural platforms, Sports excellence centers, Entrepreneurial Development Center and Societal Interaction Cell. To develop the college to become an autonomous Institution & deemed university at the earliest with facilities for advanced research and development programs on par with international standards. To invite international and reputed national Institutions and Universities to collaborate with our institution on the issues of common interest of teaching and learning sophistication.

RAIT's Mission is to produce engineering and technology professionals who are innovative and inspiring thought leaders, adept at solving problems faced by our nation and world by providing quality education.

The Institute is working closely with all stake holders like industry, academia to foster knowledge generation, acquisition, dissemination using best available resources to address the great challenges being faced by our country and World. RAIT is fully dedicated to provide its students skills that make them leaders and solution providers and are Industry ready when they graduate from the Institution.

We at RAIT assure our main stakeholders of students 100% quality for the programmes we deliver. This quality assurance stems from the teaching and learning processes we have at work at our campus and the teachers who are handpicked from reputed institutions IIT/NIT/MU, etc. and they inspire the students to be innovative in thinking and practical in approach. We have installed internal procedures to better skills set of instructors by sending them to training courses, workshops, seminars and conferences. We have also a full fledged course curriculum and deliveries planned in advance for a structured semester long programme. We have well developed feedback system employers, alumni, students and parents from to fine tune Learning and Teaching processes. These tools help us to ensure same quality of teaching independent of any individual instructor. Each classroom is equipped with Internet and other digital learning resources.

The effective learning process in the campus comprises a clean and stimulating classroom environment and availability of lecture notes and digital resources prepared by instructor from the comfort of home. In addition student is provided with good number of assignments that would trigger his thinking process. The testing process involves an objective test paper that would gauge the understanding of concepts by the students. The quality assurance process also ensures that the learning process is effective. The summer internships and project work based training ensure learning process to include practical and industry relevant aspects. Various technical events, seminars and conferences make the student learning complete.

Our Quality Policy

ज्ञानधीनं जगत् सर्वम ।

Knowledge is supreme.

Our Quality Policy

It is our earnest endeavour to produce high quality engineering professionals who are innovative and inspiring, thought and action leaders, competent to solve problems faced by society, nation and world at large by striving towards very high standards in learning, teaching and training methodologies.

Our Motto: If it is not of quality, it is NOT RAIT!

Departmental Vision, Mission

Vision

To be renowned for its high quality of teaching and research activities with a view to prepare technically sound, ethically strong and morally elevated engineers. To prepare the students to sustain impact of computer education for social needs encompassing industry educational institutions and public service.

Mission

- To provide budding engineers with comprehensive high quality education in computer engineering for intellectual growth.
- To provide state-of-art research facilities to generate knowledge and develop technologies in the thrust areas of Computer Science and Engineering.
- Providing platforms of sports, technical, co and extra curricular activities for the overall development which will enable students to be the most sought after in the country and abroad.

Departmental Program Educational Objectives (PEOs)

1. **Learn and Integrate**

   To provide Computer Engineering students with a strong foundation in the mathematical, scientific and engineering fundamentals necessary to formulate, solve and analyze engineering problems and to prepare them for graduate studies.

2. **Think and Create**

   To develop an ability to analyze the requirements of the software and hardware, understand the technical specifications, create a model, design, implement and verify a computing system to meet specified requirements while considering real-world constraints to solve real world problems.

3. **Broad Base**

   To provide broad education necessary to understand the science of computer engineering and the impact of it in a global and social context.

4. **Techno-leader**

   To provide exposure to emerging cutting edge technologies, adequate training & opportunities to work as teams on multidisciplinary projects with effective communication skills and leadership qualities.

5. **Practice citizenship**

   To provide knowledge of professional and ethical responsibility and to contribute to society through active engagement with professional societies, schools, civic organizations or other community activities.

6. **Clarify Purpose and Perspective**

   To provide strong in-depth education through electives and to promote student awareness on the life-long learning to adapt to innovation and change, and to be successful in their professional work or graduate studies.

Departmental Program Outcomes (POs)

PO1: **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2: **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4: **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5: **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6: **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7: **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9: **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
P10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11: **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

<u>Program Specific Outcomes:</u>

PSO1: To build competencies towards problem solving with an ability to understand, identify, analyze and design the problem, implement and validate the solution including both hardware and software.

PSO2: To build appreciation and knowledge acquiring of current computer techniques with an ability to use skills and tools necessary for computing practice.

PSO3: To be able to match the industry requirements in the area of computer science and engineering. To equip skills to adopt and imbibe new technologies.

Index

List of Experiments

Sr. No.	Experiments Name
1	Write a program to count number of characters ,words, sentences, lines, tabs, numbers and blank spaces present in input using LEX.
2	Write a program to recognize identifiers in C using symbol table.
3	Write a program to recognize valid arithmetic expression that uses operators + , -,* and / using YACC.
4	Write a LEX-YACC specification program for Scientific calculator.
5	Write a program to implement any parser.
6	Write a LEX-YACC specification program for 3-address intermediate code generation (ICG) and check the output with gray box probing.
7	Write a program to optimize the generated 3 address code.
8	Write a program to implement code generation algorithm for given input statement.
9	Design and implement PASS1 of Two Pass Assembler for given machine.
10	Design and implement PASS2 of Two Pass Assembler for given machine.
11	Design and implement single Pass macro processor.
12	**Case Study on JavaCC.**

Course Objectives, Course Outcome & Experiment Plan,

Course Objectives:

1.	To understand the role of compiler generator tools like LEX and YACC.
2.	To understand basic concepts and designing of assembler, Macro.
3.	To explore design of frontend and backend of compiler.

Course Outcomes:

CO1	Explore various tools like LEX and YACC.
CO2	Identify and validate different tokens for given high level language code.
CO3	Parse the given input string by constructing Top down /Bottom up parser.
CO4	Implement synthesis phase of compiler.
CO5	Generate machine code by using various databases generated in two pass assembler.
CO6	Construct different databases and implement single pass macro processor.

Experiment Plan

Module No.	Week No.	Experiments Name	Course Outcome	Weight age
1	W1	Write a program to count number of characters ,words, sentences, lines, tabs, numbers and blank spaces present in input using LEX.	CO1	5
2	W2	Write a program to recognize identifiers in C using symbol table.	CO2	10
3	W3	Write a program to recognize valid arithmetic expression that uses operators + , -,* and / using YACC.	CO1	5
4	W4	Write a LEX-YACC specification program for Scientific calculator.	CO3	5
5	W5	Write a program to implement any parser.	CO3	5
6	W6	Write a LEX-YACC specification program for 3-address intermediate code generation (ICG) and check the output with gray box probing.	CO4	3
7	W7	Write a program to optimize the 3 generated 3 address code.	CO4	3
8	W8	Write a program to implement code generation algorithm for given input statement.	CO4	2
9	W9	Design and implement PASS1 of Two Pass Assembler for given machine.	CO5	5
10	W10	Design and implement PASS2 of Two Pass Assembler for given machine.	CO5	4
11	W11	Design and implement single Pass macro processor.	CO6	10
12	W12	**Case Study on JavaCC.**	**CO4**	**2**

Mapping Course Outcomes (CO) - Program Outcomes (PO)

Subject Weightage	**Course Outcomes**	**Contribution to Program outcomes PO**											
		1	2	3	4	5	6	7	8	9	10	11	12
PRATICAL 80%	CO1: Explore various tools like LEX and YACC.	1	2	2	2	3							
	CO2: Identify and validate different tokens for given high level language code.	2	2	3	1	1							1
	CO3: Parse the given input string by constructing Top down /Bottom up parser.	1	2	3	2	1							1
	CO4: Implement synthesis phase of compiler.		2	3	2	2							1
	CO5: Generate machine code by using various databases generated in two pass assembler.	2	3	3	2								
	CO6: Construct different databases and implement single pass macro processor.	2	3	3	1		1						

Mapping of Course outcomes with Program Specific outcomes

Course Outcomes		Contribution to Program Specific outcomes		
		PSO1	PSO2	PSO3
CO1	Explore various tools like LEX and YACC.	2	3	2
CO2	Identify and validate different tokens for given high level language code.	3	2	
CO3	Parse the given input string by constructing Top down /Bottom up parser.	3	2	3
CO4	Implement synthesis phase of compiler.	3	2	1
CO5	Generate machine code by using various databases generated in two pass assembler.	3		1
CO6	Construct different databases and implement single pass macro processor.	3		1

Study and Evaluation Scheme

Course Code	Course Name	Teaching Scheme			Credits Assigned			
CSL602	System Software Lab	Theory	Practical	Tutorial	Theory	Practical	Tutorial	Total
		--	02	--	--	01	--	01

Course Code	Course Name	Examination Scheme		
CPC601	System Software Lab	Term Work	Practical & Oral	Total
		25	25	50

Term Work:

1. The distribution of marks for term work shall be as follows:

- Laboratory work (experiments/case studies): ……………………………..(15) Marks.
- Assignment: ...………..…………………………………………………... (05) Marks.
- Attendance ………..……………………………………………………… (05) Marks

TOTAL: ……………………………………………………………………….. (25) Marks.

Practical & Oral:

1. Practical exam will be based on the entire syllabus of System Programming & Compiler Construction.

System Software Lab

Experiment No. : 1

Introduction to LEX

Experiment No. 1

1. **Aim:** Write a program to count number of characters, words, sentences, lines, tabs, numbers and blank spaces present in input using LEX.

2. **Objectives:** From this experiment, the student will be able to
    - Study the role and functioning of Lexical analysis.
    - Learn software tools used in compiler construction such as lexical analyzer generator (LEX).
3. **Outcomes:** The learner will be able to
    - Explore various tools like LEX and YACC**.**
4. **Hardware / Software Required :** Ubuntu, Flex
5. **Theory:**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.
The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

A Lex program consists of three parts:

```
Declaration
%%
      Translation rules
 %%
Auxiliary procedures
```

a. Declaration

Include declarations of variables, manifest constants and regular definitions

b. Translation Rules

```
p1   {action1} /*p—pattern(Regular exp) */
…
pn    {actionn}
```

c. Auxiliary procedures

```
install_id() {
    /* procedure to install the lexeme, whose first character is pointed to by yytext and
    whose length is yyleng, into the symbol table and return a pointer thereto*/
  }
```

The Lex program is processed as shown in following diagram.

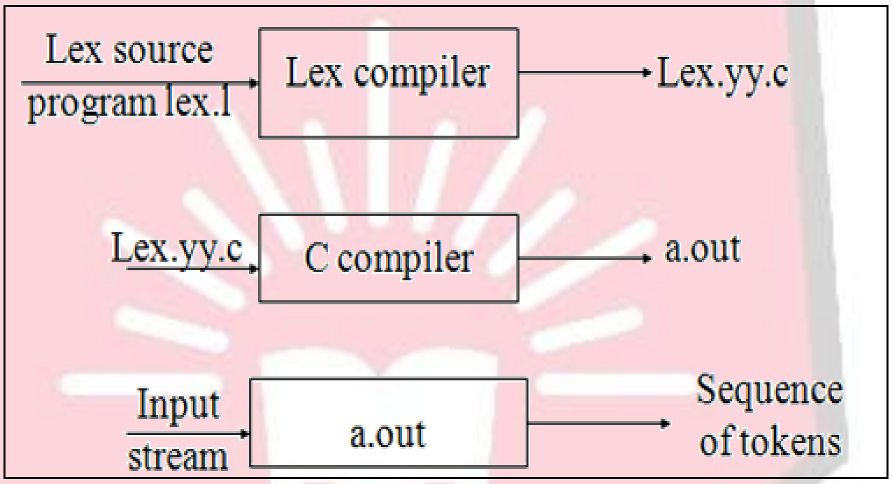


Figure 1.1 : Processing of Lex Program

6. Algorithm:-

1. Start
2. Initialize all counters.
3. Open the input file in read mode.
4. Write regular expression to recognize characters, words, sentences, lines, tabs, numbers and blank spaces in input file.
5. Increment the specific counter after recognition of characters, words, sentences, lines, tabs, numbers and blank spaces.
6. Display the count of each entity.
7. Stop

7. Conclusion :

Thus we have designed the lexical analyzer to recognize different entities like characters, words, sentences, lines, tabs, numbers and blank spaces present in any input file.

8. Viva Questions:

- What is use of Lex?
- What is rgular expression?
- What is use of yylex()?
- What is format of Lex file**?**

9. References:

- https://en.wikipedia.org/wiki/Lexical_analysis
- www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/Cours/.../l3-0708-LexA.pdf
- www.iith.ac.in/~ramakrishna/Compilers-Aug15/slides/02-lexical-analysis-part-1.pdf
- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003

System Software Lab

Experiment No. : 2

Lexical Analyzer

Experiment No. 2

1. **Aim:** Write a program to recognize identifiers in C using symbol table.

2. **Objectives:** From this experiment, the student will be able to
   - Learn Token, pattern and Lexemes.
   - Study the use of symbol table in lexical analysis.

3. **Outcomes:** The learner will be able to

   - Identify and validate different tokens for given high level language code.

4. **Hardware / Software Required :** Ubuntu, Flex

5. **Theory:**

   The very first phase of compiler is lexical analysis. The lexical analyzer read the input characters and generates a sequence of tokens that are used by parser for syntax analysis. The figure 2.1 summarizes the interaction between lexical analyzer and parser.

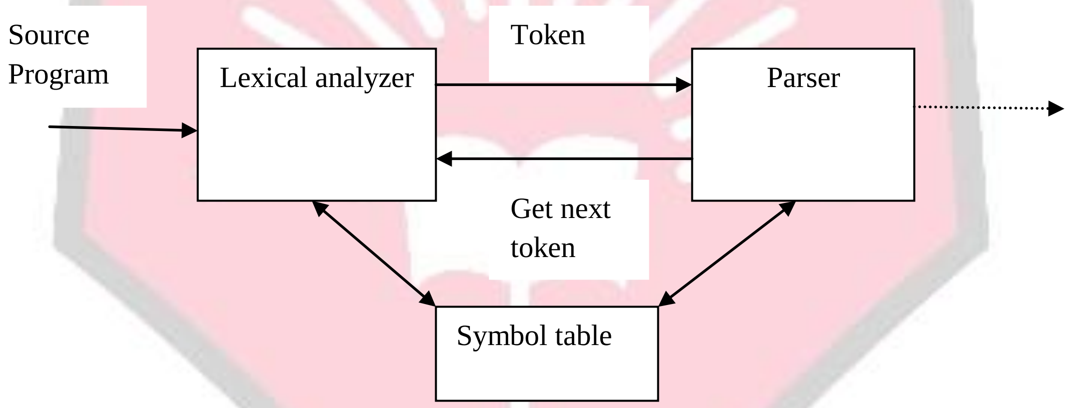


Figure 2.1: Interaction between lexical analyzer and parser

The lexical analyzer is usually implemented as subroutine or co-routine of the parser. When the "get next token" command received from parser, the lexical analyzers read input characters until it identifies next token. Lexical analyzer also performs some secondary tasks at the user interface, such as stripping out comments and white spaces in the form of blank, tab and newline characters. It also correlates error messages from compiler to source program. For example lexical analyzer may keep track of number of newline characters and correlate the line number with an error message.

In some compilers, a lexical analyzer may create a copy of source program with error messages marked in it. An important notation used to specify patterns is a regular expression. Each pattern matches a set of strings. Regular expression will serve as a name for set of strings. Lexical analyzer scans source program and breaks it up into small meaningful units called tokens. Lexical analysis identifies the lexical unit in a source statement. A token contains two fields – class code and number in class.

The three task of the lexical analyzer are:-

1. To parse the source program into the basic element or tokens of the language.
2. To build the literal and an identifier table
3. To build a uniform symbol table

Other functions performed by lexical analyzer are:-

1. Removal of comments.
2. Case conversion.
3. Removal of white space.

6. Algorithm:

```
1. Start
2. Open input source file in read mode to read C language programming statements
   (fopen).
3. Write regular expression for recognizing keywords, brackets, and identifiers, comments,
   operators and numbers.
4. Create linked list to represent symbol table, which stores identifiers in the given
   program.
5. Parse the input character string into token
6. If the word is keyword, bracket, comment, operator or number then
    6.1 Display the appropriate message.
7. If recognized word is identifier then
    7.1 If word is not present in symbol table
        7.1.1    Add word in symbol table.
8  Stop
```

7. Conclusion:

Thus we have designed and implemented lexical analyzer for the given statements in C language and understand the symbol table management for lexical analysis phase.

8. Viva Questions:

- What is the function of lexical analyzer?
- What are the Tokens in Compiler?
- What are the possible error recovery actions in lexical analysis?

9. References:

- https://en.wikipedia.org/wiki/Lexical_analysis
- www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/Cours/.../l3-0708-LexA.pdf
- www.iith.ac.in/~ramakrishna/Compilers-Aug15/slides/02-lexical-analysis-part-1.pdf
- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003

System Software Lab

Experiment No. : 3

Parser generator tool: YACC

Experiment No. 3

1. **Aim:** Write a program to recognize valid arithmetic expression that uses operators + , -,* and / using YACC.

2. **Objectives:** From this experiment, the student will be able to
   - Learn the overview of YACC generator tool
   - Learn basic YACC generator tool
   - Learn to design effective use of YACC generator tool.

3. **Outcomes:** The learner will be able to
   - Explore various tools like LEX and YACC.

4. **Hardware / Software Required :** Ubuntu, YACC

5. **Theory:**

YACC provides a general tool for imposing structure on the input to a computer program. YACC user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. YACC then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is officially known as a "parser". Its job is to analyse the structure of the input stream, and operate of the "big picture”. In the course of its normal work, the parser also verifies that the input is syntactically sound.

Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a (or a = There should be exactly one } for each { in the program.

YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers. The YACC input file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
%{
                /* C declarations and includes */
%}
                /* Yacc token and type declarations */
%%
    /* Yacc Specification  in the form of grammer rules like this:    */
  symbol    :    symbols tokens
                 { $$ = my_c_code($1); }
               ;
%%
      /* C language program (the rest)
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Attribute return values :

To access the value returned by the ith symbol in a rule body, use $i an action occurring in a rule body counts as a symbol. E.g:

```
decl : type { tval = $1 } id_list { symtbl_install($3, tval); }
```

To set the value to be returned by a rule, assign to $$ by default, the value of a rule is the value of its first symbol, i.e., $1. $$ stands for the semantic value of the grouping created under the respective rule. The storage space for our $$ is just another value attached to a token, and this is handled automatically by yacc.

A number of advanced features of YACC include-

1. Simulating Error and Accept in Actions
2. Accessing Values in Enclosing Rules.

Yacc Grammar Rules:

Simple Rules

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

Grammar production	yacc rule
$A \rightarrow B_1\ B_2\ \ldots\ B_m$	$A \rightarrow B_1\ B_2\ \ldots\ B_m$
$A \rightarrow C_1\ C_2\ \ldots\ C_n$	$\mid\ C_1\ C_2\ \ldots\ C_n$
$A \rightarrow D_1\ D_2\ \ldots\ D_k$	$\mid\ D_1\ D_2\ \ldots\ D_k$
	; /* ‘;’ optional, but advised */

Binary operators: **%left**, **%right**, **%nonassoc** :

```
%left  '+'  '-'        /*operators in the same group have same precedence */
%left  '*'  '/'
%right '^'             /* Across groups, precedence increases going down */
```

Yacc rules define what is a legal sequence of tokens in our specification language. In our case, let’s look at the rule for a simple, executable menu-command:

```
menu_item        :        LABEL  EXEC;
```

This rule defines a non-terminal symbol, menu_item in terms of the two tokens LABEL and EXEC. Tokens are also known as "terminal symbols", because the parser does not need to expand them any further. Conversely, menu_item is a "non-terminal symbol" because it can be expanded into LABEL and EXEC.

You may notice that I'm using UPPER CASE for terminal symbols (tokens), and lower-case for non-terminal symbols. This is not a strict requirement of yacc, but just a convention that has been established.

Alternate Rules

Any given menu-item may also have the keyword DEFAULT appear between the label and the executable command. Yacc allows us to have, multiple alternate definitions ofmenu_item, like this:

```
menu_item        :        LABEL  EXEC
          |      :        LABEL  DEFAULT  EXEC
```

Note that the colon (:) semi-colon (;) and or-symbol (|) are part of the yacc syntax - they are not part of our menu-file definition. All yacc rules follow the basic syntax shown above and must end in a semi-colon. We've put the semi-colon on the next line for clarity, so that it does not get confused with our syntax-definitions. This is not a strict requirement, either, but another convention of style that we will adhere to.

Note also that the word DEFAULT appears litterally, not because it is a keyword in our input-language, but because we have defined a %token called DEFAULT, and the lexer returns this token when it finds a certain piece of text.

Literal Characters in a Rule

There *is* a way to include litteral text within a rule, but it requires that the lexer pass the characters to the parser one-by-one, as tokens.

For example, remember that menu-items may have an icon-file instead of a label, like this:

```
</usr/X11R6/icons/mini.netscape.xpm> exec netscape
```

When our lexer encounters a < or > it returns the *character as a token*

We can include litteral characters in a grammar rule, like this:

```
menu_item :       LABEL  EXEC  '\n'
          |       '<' LABEL  '>'  EXEC  '\n'
          ;
```

Where the second form of the menu_item is a used when specifying an icon-file instead of a text-label. This explains why yacc allocates token-numbers starting at >255. Because the values 1-255 are reserved for litteral characters (remember 0 is reserved for end-of-file indication).

Recursive Rules

So far, we've defined a single menu-item, whereas our menu-file may contain any number of such menu-items. Yacc handles this allowing **recursive rules**, like this:

```
menu_items        :       menu_item
          |       menu_items menu_item
          ;
```

By defining menu_items in terms of itself, we now have a rule which means "one or more menu items". Note that we could also have written our recursive definition the other way round, as:

```
          |       menu_item menu_items
```

but, due to the internals of yacc, this builds a less memory-efficient parser. Refer to the section "Recursion" in the Yacc/Bison documentation for the reasons behind this.

Empty Rules

Referring back to the rule for a single menu_item, there is another way we could accommodate the optional DEFAULT keyword; by defining an empty rule, like this:

```
  menu_item       :       LABEL  default  EXEC  '\n'
          ;
```

```
default        :        /* empty */
               |        DEFAULT
```

6. Algorithm:-

1. Start
2. A YACC source program has three parts as follows:

   Declarations %% translation rules %% supporting C routines

3. Write Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
4. Write Main()- The required main program that calls the yyparse subroutine to start the program.
5. To parse the input character string into token write LEX program .
6. Those token which satisfy the lexical rules for forming identifiers are and number classified as possible identifiers and terminals.
7. Stop

7. Conclusion :

Yacc provides a general tool written in portable C language for describing the input to a computer program. Yacc turns user specification into a subroutine that handles the input process. We studied YACC tool and different commands required for compilation of Parser generator.

8. Viva Questions:

- What is the full form of YACC?
- Define YACC?
- What are the basic commands that are used for YACC tool.

9. References:

- dinosaur.compilertools.net/yacc/
- https://en.wikipedia.org/wiki/Yacc
- www.dailymotion.com/video/x3mp6bh
- www.powershow.com/.../Getting_Started_with_YACC_powerpoint.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003

System Software Lab

Experiment No. : 4

Scientific Calculator using YACC

Experiment No. 4

1. Aim: Write a LEX-YACC specification program for scientific calculator

2. Objectives: From this experiment, the student will be able to:

- Learn YACC generator tool
- Learn to compute any expression with the use of YACC and LEX tool.

3. Outcomes: The learner will be able to**:**

- Parse the given input string by constructing Top down /Bottom up parser.

4. Hardware / Software Required: ubuntu, YACC.

5. Theory:

Yacc (yet another compiler compiler) is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C programming language. Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

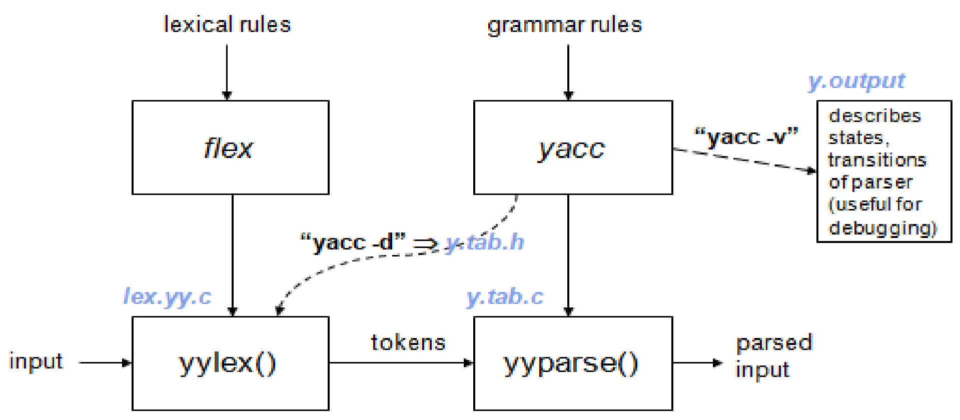


Figure 4.1 : Lex with YACC

In the YACC file, you write your own main() function, which calls yyparse() at one point. The function yyparse() is created for you by YACC, and ends up in y.tab.c.

yyparse() reads a stream of token/value pairs from yylex(), which needs to be supplied. You can code this function yourself, or have Lex do it for you. In our examples, we've chosen to leave this task to Lex. The yylex() as written by Lex reads characters from a FILE * file pointer called yyin. If you do not set yyin, it defaults to standard input. It outputs to yyout, which if unset defaults to stdout. You can also modify yyin in the yywrap() function which is called at the end of a file. It allows you to open another file, and continue parsing.

If this is the case, have it return 0. If you want to end parsing at this file, let it return 1. Each call to yylex() returns an integer value which represents a token type. This tells YACC what kind of token it has read. The token may optionally have a value, which should be placed in the variable yylval.

By default yylval is of type int, but you can override that from the YACC file by re#defining YYSTYPE. The Lexer needs to be able to access yylval. In order to do so, it must be declared in the scope of the lexer as an extern variable. The original YACC neglects to do this for you, so you should add the following to your lexter, just beneath #include <y.tab.h>:

```
extern YYSTYPE yylval;
```

Bison, which most people are using these days, does this for you automatically.

Token Values:

As mentioned before, yylex() needs to return what kind of token it encountered, and put its value in yylval. When these tokens are defined with the %token command, they are assigned numerical id's, starting from 256.Because of that fact, it is possible to have all ascii characters as a token. Let's say you are writing a calculator, up till now we would have written the lexer like this:

```
 [0-9]+         yylval=atoi(yytext); return NUMBER;
[ \n]+         /* eat whitespace */;
-             return MINUS;
\*             return MULT;
\+             return PLUS;
...
```

Our YACC grammer would then contain:

```
exp:    NUMBER
      |     exp PLUS exp
      |     exp MINUS exp
      |     exp MULT exp
```

This is needlessly complicated. By using characters as shorthands for numerical token id's, we can rewrite our lexer like this:

```
[0-9]+        yylval=atoi(yytext); return NUMBER;

[ \n]+        /* eat whitespace */;

.             return (int) yytext[0];
```

This last dot matches all single otherwise unmatched characters.

The yywrap() function

Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex . Some implementations of lex include copies of main and yywrap in a library thus eliminating the need to code them explicitly.

The yyerror() function

The yyerror() function is called when yacc encounters an invalid syntax. The yyerror() is passed a single string (char*) argument. Unfortunately, this string usually just says "parse error", so on its own, it's pretty useless. Error recovery is an important topic which we will cover in more detail later on. For now, we just want a basic yyerror() function like this:

```
yyerror(char *err) {

        fprintf(stderr, "%s\n",err);   }
```

7. Conclusion:

We studied LEX -YACC and implemented scientific Calculator with it. Also studied different functions like yywrap() and yyerror() used in YACC.

8. Viva Questions:

- What is the full form of YACC?
- Define YACC?
- What are the basic commands that are used for YACC tool.

9. References:

- dinosaur.compilertools.net/yacc/
- https://en.wikipedia.org/wiki/Yacc
- www.powershow.com/.../Getting_Started_with_YACC_powerpoint.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003

0

System Software Lab

Experiment No. : 5

LL (1) Parser

Experiment No. 5

1. Aim: Design and implement syntax analysis using LL (1) approach

2. Objectives: From this experiment, the student will be able to:

- Study and implementation of LL (1) parser.
- Learn & use the new tools and technologies used for designing a compiler.

3. Outcomes: The learner will be able to**:**

- Parse the given input string by constructing Top down /Bottom up parser.

4. Hardware / Software Required: Windows Operating System, Java

5. Theory:

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery. First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

Rules for First Sets

a. If X is a terminal then First(X) is just X!
b. If there is a Production X → ε then add ε to first(X)
c. If there is a Production X → Y1Y2..Yk then add first(Y1Y2..Yk) to first(X)
d. First(Y1Y2..Yk) is either
e. First(Y1) (if First(Y1) doesn't contain ε)
f. OR (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
g. If First(Y1) First(Y2)..First(Yk) all contain ε then add ε to First(Y1Y2..Yk) as well.

Rules for Follow Sets

a. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
b. If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).

c. If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)

d. If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Design of LL(1) parser:

An LL parser is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing leftmost derivation of the sentence. An LL parser is called an LL(k) parser if it uses k tokens of look ahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(k) grammar. LL parsers can only parse languages that have LL(k) grammars without ε-rules. FIRST is applied to the right hand side of a production rule, and tells us all the terminal symbols that can start sentences derived from that right hand side. FOLLOW is used only if the current non-terminal can derive; then we're interested in what could have followed it in a sentential form. (NB: A string can derive if and only if is in its FIRST set.)

The parser consists of

- An input buffer, holding the input string (built from the grammar)
- A stack on which to store the terminals and non-terminals.
- A parsing table.
- First and follow sets

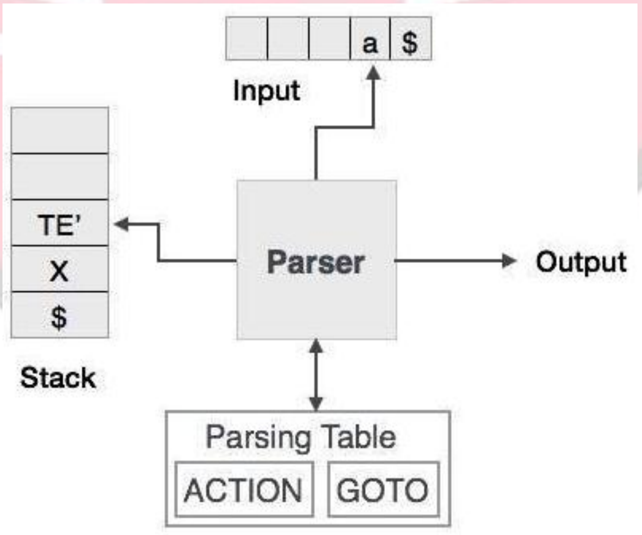


Figure 5.1: Predictive Parser

Constructing LL(1) parsing tables is relatively easy (compared to constructing the tables for lexical analysis). The table is constructed using the following algorithm:

LL(1) Table Generation

For every production A → α in the grammar:

1. If α can derive a string starting with *a* (i.e., for all *a* in FIRST(α) ,
   Table [A, *a*] = A → α
2. If α can derive the empty string, ϵ , then, for all *b* that can follow a string derived from A (i.e., for all *b* in FOLLOW (A) ,
   Table [A,*b*] = A → α

6. Algorithm:

Step 1: Compute first and follow

Algorithm for First ( ):

1. Start
2. Initialize a character array of string containing all the productions.
3. First character of every array element is a Non terminal
4. Make the appropriate entries in the Non terminal table
5. First(X) = First (Y1Y2Y3…Yk), where X is the non Terminal on the left of every production (first character of every array element)
6. It checks all the tokens by first comparing them with the entries in the Non terminal table.
7. If match found it is classified as a non terminal symbol and if match is not found it is a terminal.
8. If Y1 is a terminal then First(X)= Y1
9. Else First(X) = First (Y1)
10. Repeat 5, 6 and 7 for all the productions.
11. Stop

Algorithm for Follow ( ):

1. Start
2. First put $ (the end of input marker) in Follow (Start symbol).
3. If there is a production A → aBb, (where a can be a whole string) then everything in FIRST (b) except for ε is placed in FOLLOW(B).
4. If there is a production A → aB, then everything in FOLLOW (A) is in FOLLOW (B)
5. If there is a production A → aBb, where FIRST (b) contains ε, then everything in FOLLOW (A) is in FOLLOW (B)
6. Stop

Step 2: Construct LL(1) parsing table as with fields non terminals as rows, and terminal as columns with a end marker to show end of productions.

Step 3: Those who have a epsilon production, make its entry for the follow of the same in the parsing table.

Step 4: For a given input string with stack, input and productions applied check whether the input string satisfies the productions if you get $ $ in stack and input at bottom.

7. Conclusion:

Thus we have implemented the predictive parsing table to parse the given input using stack. If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

8. Viva Questions:

- Define LL(1) Parser.
- Explain Recursive decent parsing.
- What is Top down parsing?

9. References:

- https://www.youtube.com/watch?v=yWIDSqxIaG0
- https://www.scribd.com/.../Predictive-Parsing-and-LL-1-Compiler-Design-Dr-D-P-Sha.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003

System Software Lab

Experiment No. : 6

3-address Code Generation

Experiment No. 6

1. Aim: Write a LEX-YACC specification program for 3-address intermediate code generation (ICG) and check the output with gray box probing.

2. Objectives: From this experiment, the student will be able to:

- Learn Intermediate code Generation
- Design and Implementation of 3 address code.

3. Outcomes: The learner will be able to**:**

- Implement synthesis phase of compiler.

4. Hardware / Software Required: ubuntu, YACC

5. Theory:

The three-address code is a sequence of the statements of the form:

X:=Y op Z

Where X, Y and Z are operands and **op** is an operator.

- The operands can be constants, names or compiler generated temporaries.
- The operator is any arithmetic operator or a logical operator.

The term "Three-address code" is used because each statement usually contains three addresses, two for operands and one for result.

As only one operator is present in three address code, every arithmetic expression needs to be converted into three-address code.
For example an arithmetic expression a*b+c is converted into three address code as given below

```
t1 := a * b
t2 := t1 + c
```

where t1 and t2 are compiler generated temporaries.

Implementation of three address code
The abstract form of an intermediate code is three-address statement. These statements can be implemented in compiler, as records of fields for operators and operand. The three such implementations are: quadruples, triples and indirect triples.

1. **Quadruples**

A quadruple is a record structure with four fields, namely **op**, **arg1**, **arg2** and **result**. The **op** field contains code for operators. The **arg1**, **arg2** and **result** fields store the address of the identifier for which entry is created in symbol table.
Example

Consider the three-address cod for assignment statement a:=b*c+b*c

	op	arg1	arg2	result
(0)	*	B	c	t1
(1)	*	B	c	t2
(2)	+	t1	t2	t3
(3)	:=	t3		A

Figure 6.1 : Quadruple representation of three-address statement

2. Triples

In quadruples, we have to create entry for temporary variables in symbol table. To avoid this entry we might refer to the temporary value by the position of the statement it computes. So the three-address statement is represented by three field's **op**, **arg1** and **arg2**. Since only three fields are used, the intermediate code format is known as triples.

	Op	arg1	arg2
(0)	*	B	C
(1)	*	B	C
(2)	+	(0)	(1)
(3)	Assign	A	(2)

Figure 6.2: Triple representation of three-address statement

The information needed to interpret the different entries in **arg1** and **arg2** fields can be encoded into op field. The statement **a:=t3** is encoded in the triple representation by placing **a** in **arg1** and using the operator **assign**.

3. Indirect Triples

Another representation of three-address code is that listing pointers to triples, rather than listing the triples themselves. This representation is called indirect triples.

Let us use array named **statement** to list pointers to triples in the desired order. The triples in figure 4 might be represented as in figure 6.3.

	statement
(0)	(101)
(1)	(102)
(2)	(103)
(3)	(103)

	op	arg1	arg2
(101)	*	b	C
(102)	*	b	C
(103)	+	101	102
(104)	assign	a	103

Figure 6.3: Indirect Triple representations of three-address statement

GIMPLE

GIMPLE is a family of intermediate representations (IR) based on the tree data structure. At present, there are only two kinds of GIMPLE:

- High level GIMPLE is what the middle-end produces when it lowers the GENERIC language that is targeted by all the language front ends.
- Low Level GIMPLE is obtained by linearizing all the high-level control flow structures of high level GIMPLE, including nested functions, exception handling, and loops.
- SSA GIMPLE is low level GIMPLE rewritten in SSA form.

Almost all tree optimizer passes work on SSA GIMPLE, while some work on normal low level GIMPLE. It is very likely that at one point, GCC will lower low level GIMPLE even further, in order to simplify the GIMPLE to RTL expand pass.

The gcc (or cc1) argument options *-fdump-tree-all -fdump-tree-ssa -fdump-tree-optimized* etc... might be used to have some GIMPLE intermediate representations dumped (in a simpler textual form).

6. Algorithm:

1. Start.
2. Read the input arithmetic expression.
3. Apply the triple representation of three address code.
4. Generate the three address code for each expression.
5. Stop.

7. Conclusion:

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. The intermediate code generator will try to divide expression into sub-expressions and then generate the corresponding three-address code which has at most three address locations to calculate the expression.

8. Viva Questions:

- What is the purpose of TAC?
- Explain Quadruples, triples.

9. References:

- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003
- https://en.wikipedia.org/wiki/Code_generation_(compiler).
- https://nptel.ac.in/courses/106108113/module5/Lecture17.pdf

System Software Lab

Experiment No. : 7

Code Optimization Techniques

Experiment No. 7

1. Aim: To implement any Code Optimization Technique.

2. Objectives:

- To analyze the need for compiler analysis and optimization techniques.
- To learn the function of Code Optimization Techniques.
- To learn the efficient tools to implement Code Optimization Techniques

3. Outcomes: The learner will be able to**:**

- Implement synthesis phase of compiler.

4. Hardware / Software Required: Windows Operating System

5. Theory:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
    item = 10;
    value = value + item;
  } while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
  Item = 10;
  do
  {
    value = value + item;
  } while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

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 memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

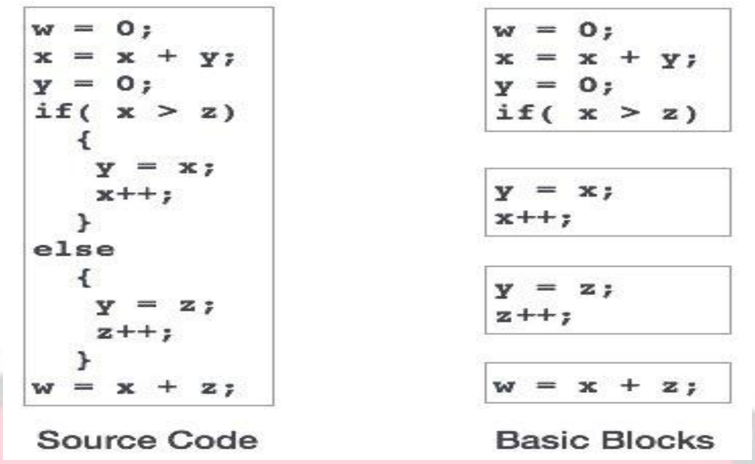
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
    y = x;
    x++;
  }
else
  {
    y = z;
    z++;
  }
w = x + z;
```

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

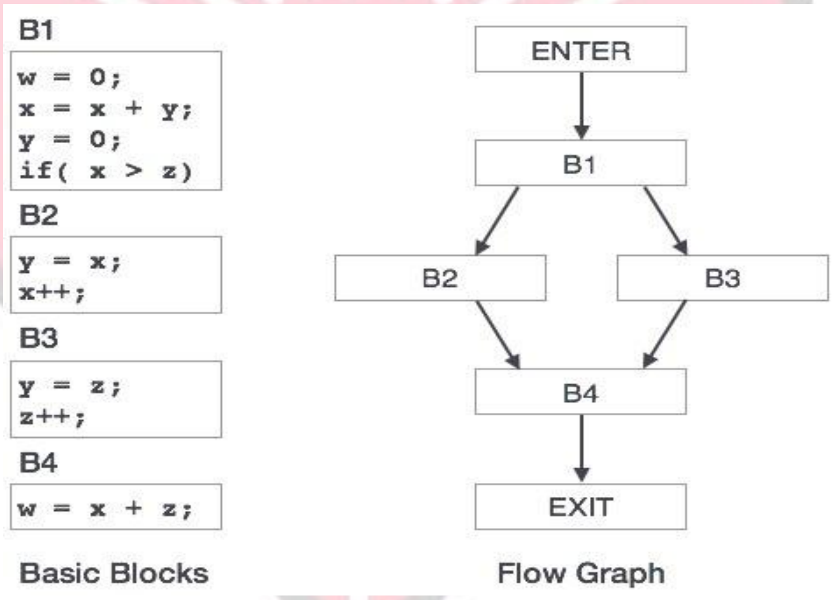
```
y = z;
z++;
```

```
w = x + z;
```

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



B1
```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

B2
```
y = x;
x++;
```

B3
```
y = z;
z++;
```

B4
```
w = x + z;
```

$$\begin{tikzcd}
& \text{ENTER} \arrow[d] & \\
& \text{B1} \arrow[dl] \arrow[dr] & \\
\text{B2} \arrow[dr] & & \text{B3} \arrow[dl] \\
& \text{B4} \arrow[d] & \\
& \text{EXIT} &
\end{tikzcd}$$

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code**: A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.

- **Induction analysis**: A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction**: There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

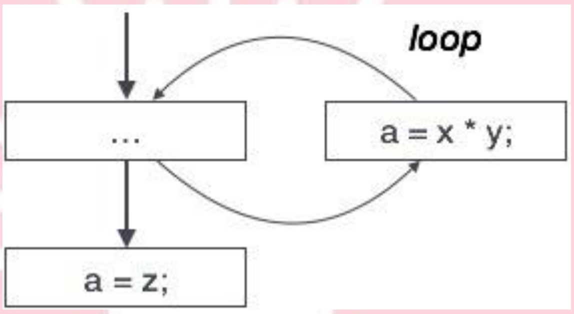
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements who have computed values are used only under certain s, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop. Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.

6. Conclusion:

Code optimization is a process, to improve the code in terms of execution time or memory space, without changing meaning of program. It transforms the code in such way that, the code will consume less resource and deliver high speed.

7. Viva Questions:

- What is meant by code optimization?
- What is the function of Dead code elimination?
- What are different machine dependent code optimization technoiques?

8. References:

- https://en.wikipedia.org/wiki/Optimizing_compiler
- www.unf.edu/public/cop4620/ree/Notes/optimization.ppt
- https://www.youtube.com/watch?v=8hNrXo88LCc
- www.isical.ac.in/~mandar/compilers/optimization.pdf

System Software Lab

Experiment No. : 8

Code Generation

Experiment No. 8

1. Aim: Write a program to implement code generation algorithm for given input statement.

2. Objectives: The learner will be able:

- To learn the Code generation Algorithm.
- To implementation the flow graph for the

3. Outcomes: The learner will be able to**:**

- Implement synthesis phase of compiler.

4. Hardware / Software Required: Windows Operating System, Java

5. Theory:

Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

- Generates the shortest sequence of instructions
- Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)

1. Machine model
   - All computations are carried out in registers
   - Instructions are of the form op R,R or op M,R
2. Always computes the left sub tree into a register and reuses it immediately
3. Two phases
   - Labelling phase
   - Code generation phase

1. The Labelling Algorithm

Labels each node of the tree with an integer:

- fewest no. of registers required to evaluate the tree with no intermediate stores to memory
- Consider binary trees

For leaf nodes

- if n is the leftmost child of its parent then
label(n) := 1 else label(n) := 0

For internal nodes

- label(n) = max (l1, l2), if l1<> l2
  = l1 + 1, if l1 = l2

6. Algorithm:

Procedure GENCODE(n)

1. RSTACK – stack of registers, R0,...,R(r-1)
2. TSTACK – stack of temporaries, T0,T1,...
3. A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into
4. The register top(RSTACK) ,and the rest of RSTACK remains in the same state as the one before the call
5. A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

The Code Generation Algorithm (1)

```
Procedure gencode(n);
{ /* case 0 */
if
        n is a leaf representing operand N and is the leftmost child of its parent
then
        print(LOAD N, top(RSTACK))
```

The Code Generation Algorithm (2)

```
/* case 1 */

else if

        n is an interior node with operator OP, left child n1, and right child n2

then

        if label(n2) == 0 then {

                let N be the operand for n2;

                gencode(n1);

        print(OP N, top(RSTACK));

}
```

The Code Generation Algorithm (3)

```
/* case 2 */

else if ((1 < label(n1) < label(n2)) and( label(n1) < r))

then {

        swap(RSTACK); gencode(n2);

        R := pop(RSTACK); gencode(n1);

        /* R holds the result of n2 */

        print(OP R, top(RSTACK));
```

```
                push (RSTACK,R);
                swap(RSTACK);
        }
```

The Code Generation Algorithm (4)

```
        /* case 3 */
        else if ((1 < label(n2) < label(n1)) and( label(n2) < r))
        then {
                gencode(n1);
                R := pop(RSTACK); gencode(n2);
                /* R holds the result of n1 */
                print(OP top(RSTACK), R);
                push (RSTACK,R);
        }
```

The Code Generation Algorithm (5)

```
        /* case 4, both labels are > r */
        else {
                gencode(n2); T:= pop(TSTACK);
                print(LOAD top(RSTACK), T);
                gencode(n1);
                print(OP T, top(RSTACK));
                push(TSTACK, T);
                }}
```

7. Conclusion:

The code generation algorithm takes as input a sequence of three address statements constituting a basic block. For each three address statement of the form x=y op z, a target machine code is generated.

8. Viva Questions:

- Major task in code generation.
- List the characteristics of peephole optimization.

9. References:

- Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman, Compilers principles, techniques and tools, IE, PHI 2003
- https://en.wikipedia.org/wiki/Code_generation_(compiler).
- cse.iitkgp.ac.in/~bivasm/notes/scribe/11CS30038.ppt
- www.personal.kent.edu/~rmuhamma/Compilers/MyCompiler/codeGen.htm

System Software Lab

Experiment No. : 9

Two Pass Assembler- Pass1

Experiment No. 9

1. **Aim:** Design and implement PASS1 of Two Pass Assembler for given machine.

2. **Objectives:** From this experiment, the student will be able to

- To understand the role of MOT and POT
- Symbol table generation.
- Literal Table generation
- To generate intermediate code after pass 1

3. **Outcomes:** The learner will be able to

- Generate machine code by using various databases generated in two pass assembler.

4. **Hardware / Software Required:** Windows Operating System, Java

5. **Theory:**

The assembler is a program which accepts an assembly language program as input and generates its equivalent machine code. It also produces some information to be used by loader. This process is depicted in following figure.

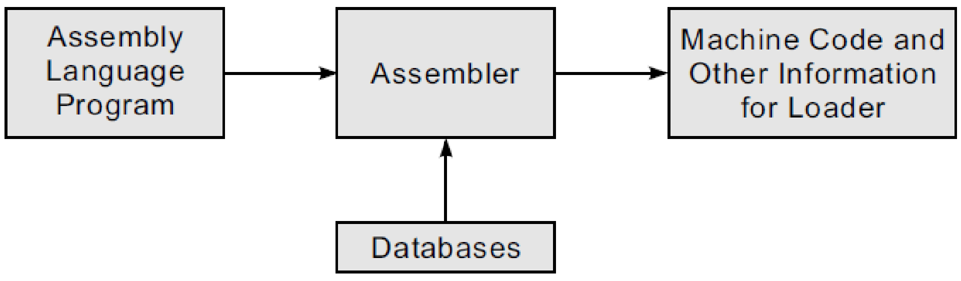

Assembly Language Program
Assembler
Machine Code and Other Information for Loader
Databases

Figure 1: Function of Assembler.

An assembler performs the following functions

1. Generate instructions
   i. Evaluate the mnemonic in the operator field to produce its machine code.
   ii. Evaluate subfields- find value of each symbol, process literals & assign address.
2. Process pseudo ops.

Pass 1: Purpose - To define symbols & literals

1. Determine length of machine instruction (MOTGET)
2. Keep tack of location counter (LC)
3. Remember values of symbols until pass2 (STSTO)
4. Process some pseudo ops. EQU
5. Remember literals (LITSTO)

Pass 1: Database

1. Source program
2. Location counter(LC) which stores location of each instruction
3. Machine Operation Table (MOT). This table indicates the symbolic mnemonic for each instructions and its length.
4. Pseudo Operation Table (POT). This table indicates the symbolic mnemonic and action taken for each pseudo-op in pass1.
5. Symbol Table (ST) which stores each label along with its value.
6. Literal Table(LT) which stores each literal and its corresponding address
7. A copy of input which will be used by pass2.

Format of databases

The Machine Operation Table (MOT) and Pseudo Operation Table (POT) are examples of fixed tables. During the assembly process the contents of this table are not filled in or altered

1. **Machine-op Table (MOT)**

6- bytes per entry

Mnemonic -codes (4 bytes) characters	Binary op-codes (1 byte) hexadecimal	Instruction length (2-bits) binary	Instruction format (3-bits) binary	Not used in this design (3 bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	
………	…………..	………….	…………….	

b : bank space

2. **Pseudo-op Table (POT)**

8-BYTES PER ENTRY

Pseudo-op (5-bytes) characters	Address of routine to process pseudo-op (3 bytes= 24 bits address)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

Let us consider following source code and find the contents of symbol table and literal table.

Stmt no	Symbol	Op-code	Operands
1	SAMPLE	START	0
2		USING	*,15
3		A	1,FOUR
4		A	2,FIVE
5	TEMP	EQU	10
6		A	3,=F'3'
7		USING	TEMP,15
8	FOUR	DC	F'4'
9	FIVE	DC	F'5'
10		END	

3. **Symbol Table:**

Symbol	Value	Length	Relocation
"SAMPLEbbb"	0	1	"R"
"TEMPbbbb"	10	4	"A"
"FOURbbbb"	12	4	"R"
"FIVEbbbb"	16	4	"R"

4. **Literal Table**

Literal	Value	Length	Relocation
F'3'	20	4	"R"

5. **Code after pass1:**

Stmt no	Relative address	Statement		
1		SAMPLE	START	0
2			USING	*,15
3	0		A	1, _ (0,15)
4	4		A	2, _ (0,15)
5		-		
6	8		A	3, _ (0,15)
7		-		
8	12	FOUR	4	
9	16	FIVE	5	
10		-		

6. Algorithm: - Pass1

1. Initially location counter is set to relative address 0 i.e. LC=0
2. Read the statement from source program
3. Examine the op-code field: If match found in MOT then
    - 3.1 From the MOT entry determine the length field i.e. L=length.
    - 3.2 Examine the operand field to check whether literal is present or not. If any new literal is found then corresponding entry is done in LT.
    - 3.3 Examine the label field for the presence of symbol. If label is present then it is entered in ST and current value of location counter is assigned to symbol.
    - 3.4 The current value of location counter is incremented by length of instruction(L)
4. If match found in POT then
    - 4.1 If it is USING or DROP pseudo-op then first pass do nothing. It just writes a copy of these cards for pass 2.
    - 4.2 If it is EQU pseudo-op then evaluate expression in operand field and assign value to the symbol present in label field.
    - 4.3 If it is DS or DC pseudo-op then by examining the operand field find out number of bytes of storage required. Adjust the location counter for proper alignment.
    - 4.4 If it is END pseudo-op then pass1 is terminated and control is passed to pass2. Before transferring the control it assigns location to literals.
5. A copy of source card is saved for pass 2.
6. Go to step 2.

7. Conclusion:

Thus, the functionality of first pass of a two pass assembler is implemented. The MOT and POT are traversed to find match for the instruction. The symbol table, literal table and intermediate code are generated for the given program.

8. Viva Questions:

- What is symbol Table?
- Design a symbol table format for the language.
- What is POT and MOT?
- Define Labels.

9. References:

- www.cse.aucegypt.edu/~rafea/csce447/slides/table.pdf
- https://en.wikipedia.org/wiki/Symbol_table.
- https://www.youtube.com/watch?v=j3SCUBsZm4A.
- J. J Donovan: Systems Programming Tata McGraw Hill Publishing Company

System Software Lab

Experiment No. : 10

Two Pass Assembler- Pass2

Experiment No. 10

1. **Aim:** Design and implement PASS2 of Two Pass Assembler for given machine.
2. **Objectives:** From this experiment, the student will be able to
    1. To generate Base table
    2. To generate machine code
3. **Outcomes:** The learner will be able to

- Generate machine code by using various databases generated in two pass assembler.

4. **Hardware / Software Required:** Windows Operating System, Java

5. Theory:

Pass 2: Purpose - To generate object program

1) Look up value of symbols (STGET)
2) Generate instruction (MOTGET2)
3) Generate data (for DC, DS)
4) Process pseudo ops (POT, GET2)

Pass 2: Database

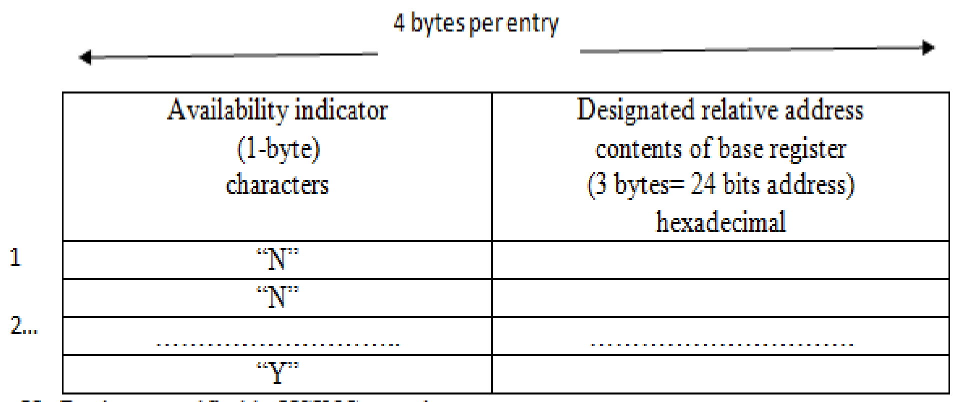
1) Copy of source program from Pass1
2) Location counter
3) MOT which gives the length, mnemonic format opcode
4) POT which gives mnemonic & action to be taken
5) Symbol table from Pass1
6) Base table which indicates the register to be used or base register
7) A work space INST to hold the instruction & its parts
8) A work space PRINT LINE, to produce printed listing
9) A work space PUNCH CARD for converting instruction into format needed by loader
10) An output deck of assembled instructions needed by loader.

Format of database:

Base Table:

Assembler uses this table to generate proper base register reference in machine instructions and to compute offset. Then the offset is calculated as:

offset= value of symbol from ST - contents of base register


4 bytes per entry

	Availability indicator (1-byte) characters	Designated relative address contents of base register (3 bytes= 24 bits address) hexadecimal
1	"N"	
	"N"	
2...	…………………………..	……………………………
	"Y"	

Y : Register specified in USING pseudo-op

N: Register never specified in USING pseudo-op or made unavailable by DROP pseudo-op.

Let us consider the same example as experiment no. 10 and the base table after statement 2:

Base register	**Contents**
15	**0**

After statement 7:

Base register	**Contents**
15	**10**

Code after pass2:

stmt	**Relative address**	**Statement**	
3	0	A	1, 12 (0,15)
4	4	A	2, 16(0,15)
6	8	A	3, 20(0,15)
8	12	4	
9	16	5	
10		-	

6. Algorithm:

1. Initialize the location counter as: LC=0
2. Read the statement from source program
3. Examine the op-code field: If match found in MOT then
   a. From the MOT entry determine the length field i.e. L=length, binary op-code and format of the instruction.
      Different instruction format requires different processing as described below:
   1. **RR Instruction : (Register to Register)**
      Both of the register specification fields are evaluated and placed into second byte of RR instruction
   2. **RX Instruction : (Register to Index)**
      Both of the register and index fields are evaluated and processed similar to RR instruction. The storage address operand is evaluated to generate effective address (EA). The BT is examined to find the base register. Then the displacement is determined as:
         D=EA- Contents of base register.

      The other instruction formats are processed in similar manner to RR and RX.

   b. Finally the base register and displacement specification are assembled in third and fourth bytes of instruction.
   c. The current value of location counter is incremented by length of instruction.

4. If match found in POT then
   a. If it is EQU pseudo-op then EQU card is printed in the listings.
   b. If it is USING pseudo-op then the corresponding BT entry is marked as available.
   c. If it is DROP pseudo-op then the corresponding BT entry is marked as unavailable.
   d. If it is DS or DC pseudo-op then various conversions are done depending on the data type and symbols are evaluated. Location counter is updated by length of data.
   e. END pseudo-op indicates end of source program and then pass2 is terminated. Before that if any literals are remaining then the code is generated for them.
5. After assembling the instruction it is put in the format required by loader.
6. Finally a listing is printed which consist of copy of source card, its storage location and hexadecimal representation.
7. Go to step 2.

7. Conclusion:

Thus, the functionality of second pass of a two pass assembler is implemented. The base table and machine code are generated for the given program. The program is assembled, check the assembled code and see whether forward reference problem is solved.

8. Viva Questions:

- Define the Base Table.
- What is the difference between one pass and two pass assembler?

- Why do we need a two pass assembler?

9. References:

- codearea.in/program-for-implementing-pass-1-of-assembler/
- https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf
- users.cis.fiu.edu/~downeyt/cop3402/two-pass.htm
- https://www.youtube.com/watch?v=1j9dE2nQ-Yo
- J. J Donovan: Systems Programming Tata McGraw Hill Publishing Company

System Software Lab

Experiment No. : 11

Macro processor

Experiment No. 11

1. Aim: Design and implement single Pass macro processor.

2. Objectives: From this experiment, the student will be able to:

- To understand the functionality of Macro Processor.
- Study and implement the Macro Processor.
- Learn scope and significance of Macro Processor.

3. Outcomes: The learner will be able to**:**

- Construct different databases and implement single pass macro processor.

4. Hardware / Software Required: Windows Operating System, Java

5. Theory:

One-Pass Macro Processor:

- A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition.
- Restriction The definition of a macro must appear in the source program before any statements that invoke that macro. This restriction does not create any real inconvenience. The design considered is for one-pass assembler. The data structures required are:

- **DEFTAB (Definition Table)**
  - ➢ Stores the macro definition including macro prototype and macro body
  - ➢ Comment lines are omitted.
  - ➢ References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
- **NAMTAB (Name Table)**
  - ➢ Stores macro names
  - ➢ Serves as an index to DEFTAB
- Pointers to the beginning and the end of the macro definition (DEFTAB)
- **ARGTAB (Argument Table)**
  - ➢ Stores the arguments according to their positions in the argument list.
  - ➢ As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.

6. Algorithm:

```
begin {macro processor}
        EXPANDING : = FALSE
            while OPCODE ≠ 'END' do
              begin
                 GETLINE
                 PROCESSLINE
                 end  {while}
            end {macro processor}
        procedure PROCESSLINE
            begin
              search NAMTAB for OPCODE
              if found then
                 EXPAND
             else if OPCODE = 'MACRO' then
                 DEFINE
             else write source line to expanded file
             end {PROCESSLINE}


procedure EXPAND
    begin
      EXPANDING : = TRUE
      get first line of macro definition {prototype} from DEFTAB
      set up arguments from macro invocation in ARGTAB
      write macro invocation to expanded file as a comment
      while not end of macro definition do
         begin
           GETLINE
           PROCESSLINE
           end {while}
         EXPANDING : = FALSE
    end  {EXPAND}
```

```
procedure GETLINE
    begin
        if EXPANDING then
            begin                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
    else
        read next line from input file
end {GETLINE}
```

7. Conclusion:

One pass macro processor is implemented by creating DEFTAB, NAMTAB, ARGTAB tables.

8. Viva Questions:

- Explain what is meant by pass of macro-processor.
- Database used in one pass macro processor .
- Define the term macro

9. References:

- https://en.wikipedia.org/wiki/General-purpose_macro_processor
- http://solomon.ipv6.club.tw/~solomon/Course/SP/sp4-1.pdf
- https://www.computer.org/csdl/trans/ts/1976/02/01702350.pdf
- J. J Donovan: Systems Programming Tata McGraw Hill Publishing Company

System Software Lab

Experiment No. : 12

JavaCC

Experiment No. 12

1. Aim: Case Study on JavaCC.

2. Objectives: From this experiment, the student will be able to:

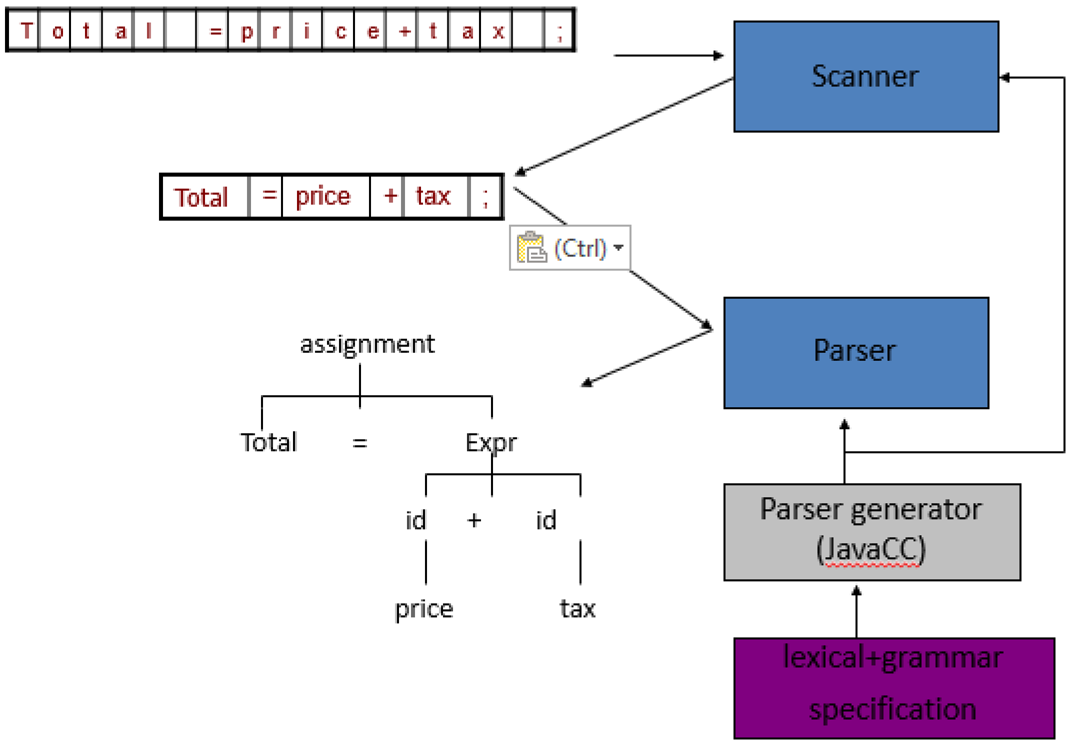
- To study JavaCC.
- To familiarize and encourage the students to use various software tools for Compilation.

3. Outcomes: The learner will be able to**:**

- Implement synthesis phase of compiler.

4. Hardware / Software Required: Windows Operating System, ubuntu.

5. Theory: Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions and debugging.



T o t a l = p r i c e + t a x ;
Scanner
Total = price + tax ;
(Ctrl)
assignment
Total
=
Expr
id
+
id
price
tax
Parser
Parser generator (JavaCC)
lexical+grammar specification

Figure 12.1: JavaCC Compiler System

Features of JavaCC

- TopDown LL(K) parser generator
- Lexical and grammar specifications in one file
- Tree Building preprocessor
  - with JJTree
- Extreme Customizable
  - many different options selectable
- Document Generation
  - by using JJDoc
- Internationalized
  - can handle full unicode
- Syntactic and Semantic lookahead
- Permits extneded BNF specifications
  - can use | * ? + () at RHS.
- Lexical states and lexical actions
- Case-sensitive/insensitive lexical analysis
- Extensive debugging capability
- Special tokens
- Very good error reporting

JavaCC Installation

- Download the file javacc-4.X.zip from https://javacc.dev.java.net/
- unzip javacc-4.X.zip to a directory %JCC_HOME%
- add %JCC_HOME%\bin directory to your %path%.
  - javacc, jjtree, jjdoc are now invokable directly from the command line.

Steps to use JavaCC

- Write a javaCC specification (.jj file)
  - Defines the grammar and actions in a file (say, calc.jj)
- Run javaCC to generate a scanner and a parser
  - javacc calc.jj
  - Will generate parser, scanner, token,… java sources
- Write your program that uses the parser
  - For example, UseParser.java
- Compile and run your program
  - javac -classpath . *.java
  - java   -cp . mainpackage.MainClass

6. Conclusion:

From this experiment we understood the working of JavaCC compiler which converts high level language to machine level language.

7. Viva Questions:

- What does JavaCC mean?
- Explain working of LLVM.

8. References:

https://javacc.github.io/javacc/