Anjuman-i-Islam’s

M. H. Saboo Siddik College Of Engineering

Computer Engineering Department

|  |
| --- |
| Anjuman-i-Islam’s  M. H. Saboo Siddik College Of Engineering  Computer Engineering Department  **Lab Manual**  **System Programming and Compiler Construction**  (2015 Rev)  Semester VI Computers  Prepared By: Er. Nafisa M. Mapari |

**INDEX**

|  |  |
| --- | --- |
| EXP. NO. | EXPERIMENT NAME |
| 1 | To implement Lexical analyzer tool: Flex |
| 2 | To implement Parser generator tool: Yacc |
| 3 | To Find FIRST(), FOLLOW() set of given grammar. |
| 4 | To removing left recursion using direct as well as indirect method for given the set of production rule. |
| 5 | To Design and implement two pass assembler for X86 machine. |
| 6 | To design and implement two pass Macro Processor. |
| 7 | To implement Syntax Analysis using operator precedence parser. |
| 8 | To generate target code for the code optimized, considering the target machine to be X86. |
| 9 | To implement code optimization. |
| 10 | To create shared library in Linux environment and using it. |

**EXPERIMENT NO-1**

**AIM:** To implement Lexical analyzer tool: Flex

**THEORY:**

**Lexical Analyzer:**

*Lexical analysis* or *scanning* is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. *Tokens* are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

The Lexical Analyzer is the interface between the source program and the compiler. The lexical Analyzer reads the source program one character at a time and can be treated as a single entity. The lexical analyzer (scanner) is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

Source Program

Lexical Analyzer

Parser

Symbol Table

The lexical analyzer is a subroutine of the parser i.e. scanner will operate under the control of the parser. Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters from it until it can identify the next token.

Identifiers, keywords, constants, operators and punctuation symbols such as commas and parenthesis are typical tokens. What is called a token depends upon the language at hand and to some extent, on discretion of the computer designer but in general, each token is a substring of the source program that has to be treated as a single unit.

Regular expressions are used to specify the tokens. The advantage of using regular expressions is that a recognizer for the token, called Finite Automata could be easily constructed.

Other functions performed by the lexical analyzer are:

* Removal of comments
* Case conversion
* Removal of white spaces
* Communication with symbol table i.e. storing information regarding an identifier in the symbol table

**Fast LEX (Flex) :**

Flex is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named, "lex.yy.c", which defines the function yylex(). The file "lex.yy.c" can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

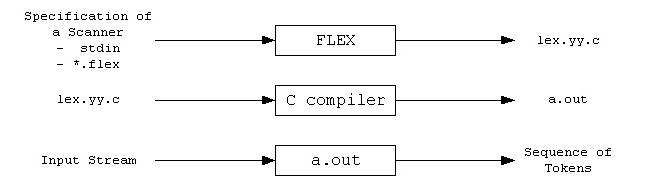
Flex can only generate code for [C](http://en.wikipedia.org/wiki/C_(programming_language)) and [C++](http://en.wikipedia.org/wiki/C%2B%2B). To use the scanner code generated by flex from other languages a [language binding](http://en.wikipedia.org/wiki/Language_binding) tool such as [SWIG](http://en.wikipedia.org/wiki/SWIG) can be used.

A similar lexical scanner for [C++](http://en.wikipedia.org/wiki/C%2B%2B) is **flex++**, which is included as part of the flex package. At the moment, flex supports generating code only for C and C++. The generated code does not depend on any [runtime](http://en.wikipedia.org/wiki/Runtime_library) or external [library](http://en.wikipedia.org/wiki/Library_(computing)) except for a memory allocator ([malloc](http://en.wikipedia.org/wiki/Malloc) or a user-supplied alternative) unless the input also depends on it. This can be useful in[embedded](http://en.wikipedia.org/wiki/Embedded_system) and similar situations where traditional [operating system](http://en.wikipedia.org/wiki/Operating_system) or [C runtime](http://en.wikipedia.org/wiki/C_standard_library) facilities may not be available.

The flex++ classes and code require a C++ compiler to create lexical and pattern-matching programs. The flex++ generated C++ scanner includes the header file FlexLexer.h, which defines the interfaces of the two C++ generated classes.

**How to use flex:**

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the *vocabulary* of a certain language (e.g. Simple), write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



First, FLEX reads a specification of a scanner either from an input file \*.lex, or from standard input, and it generates as output a C source file *lex.yy.c*. Then, *lex.yy.c* is compiled and linked with the "-lfl" library to produce an executable *a.out*. Finally, *a.out* analyzes its input stream and transforms it into a sequence of tokens.

* **\*.lex** is in the form of pairs of regular expressions and C code.
* **lex.yy.c** defines a routine *yylex()* that uses the specification to recognize tokens.
* **a.out** is actually the scanner

**How to input file:**

1. Format:  
       definitions  
       %%  
       rules  
       %%  
       user code
2. The *definitions* section: "name definition"  
   The *rules* section: "pattern action"  
   The *user code* section: "yylex() routine"

**How to execute with flex:**

1. Try sample\*.lex
2. Command Sequence:  
       flex sample\*.lex  
       gcc lex.yy.c -lfl  
       ./a.out

**sample1.lex**

/\*

\* Sample Scanner1:

\* Description: Replace the string "username" from standard input

\* with the user's login name (e.g. lgao)

\* Usage: (1) $ flex sample1.lex

\* (2) $ gcc lex.yy.c -lfl

\* (3) $ ./a.out

\* stdin> username

\* stdin> Ctrl-D

\* Question: What is the purpose of '%{' and '%}'?

\* What else could be included in this section?

\*/

%{

/\* need this for the call to getlogin() below \*/

#include <unistd.h>

%}

%%

username printf("%s\n", getlogin());

%%

main()

{

yylex();

}

**sample2.lex**

/\*

\* Sample Scanner2:

\* Description: Count the number of characters and the number of lines

\* from standard input

\* Usage: (1) $ flex sample2.lex

\* (2) $ gcc lex.yy.c -lfl

\* (3) $ ./a.out

\* stdin> whatever you like

\* stdin> Ctrl-D

\* Questions: Is it ok if we do not indent the first line?

\* What will happen if we remove the second rule?

\*/

int num\_lines = 0, num\_chars = 0;

%%

\n ++num\_lines; ++num\_chars;

. ++num\_chars;

%%

main()

{

yylex();

printf("# of lines = %d, # of chars = %d\n", num\_lines, num\_chars);

}

**CONCLUSION:-**

Thus we have successfully implemented lex file using Flex tool.

**EXPERIMENT NO-2**

**AIM:** To implement Parser generator tool: YACC

**THEORY:**

**PARSER**

Parser is a program that takes as input a sentence , and if the grammar can derive the sentence, it generates a parse tree, els e it generates an error.

Sentence

Parser

Parse Tree Error

The following diagram shows the communication between parser and scanner.

get next token

Parser

Scanner

return token

Symbol

Table

**Yet Another Compiler Compiler (YACC) :**

Yacc is officially known as a "parser". It's job is to analyze 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. 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. Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.

For example, a C program may contain something like:

{

int int;

int = 33;

printf("int: %d\n",int);

}

In this case, the lexical analyser would have broken the input stream into a series of "tokens", like this:

{

int

int

;

int

=

33

;

printf

(

"int: %d\n"

,

int

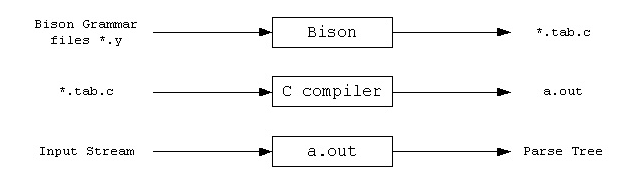
)

;

}

**How to use Bison:**

*Bison* is a general-purpose *parser generator* that converts a grammar description (Bison Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

****

**Steps to use Bison:**  
  
http://alumni.cs.ucr.edu/~lgao/Img/dot.gif Write a lexical analyzer to process input and pass tokens to the parser.  
http://alumni.cs.ucr.edu/~lgao/Img/dot.gif Write the grammar specification for bison , including grammar rules, yyparse() and yyerror().  
http://alumni.cs.ucr.edu/~lgao/Img/dot.gif Run Bison on the grammar to produce the parser.   
http://alumni.cs.ucr.edu/~lgao/Img/dot.gif Compile the code output by Bison, as well as any other source files.  
http://alumni.cs.ucr.edu/~lgao/Img/dot.gif Link the object files to produce the finished product.

**How to input file:**

1. Format:  
       %{  
       C Declarations  
       %}  
       Bison Declarations  
       %%  
       Grammar Rules  
       %%  
       Additional C Code
2. Useful Bison definitions:  
       %token, %union, %type, %left, %right, %nonassoc, ...  
   Format of the grammar rules section:  
       ***result: components ...  
           ;***  
   Important data structure and functions:  
       yylval, YYSTYPE, yyerror(), yyparse()

**How to execute with Bison:**

1. Create a Directory: "mkdir calc"
2. Save the five files ([calc.lex](http://alumni.cs.ucr.edu/~lgao/teaching/calc/calc.lex), [calc.y](http://alumni.cs.ucr.edu/~lgao/teaching/calc/calc.y), [Makefile](http://alumni.cs.ucr.edu/~lgao/teaching/calc/Makefile), [main.cc](http://alumni.cs.ucr.edu/~lgao/teaching/calc/main.cc), and [heading.h](http://alumni.cs.ucr.edu/~lgao/teaching/calc/heading.h)) to directory "calc"
3. Command Sequence: "make"; "./calc"
4. Use input programs (or stdin) which contain expressions with integer constants and operators + and \*, then press Ctrl-D to see the result

**cacl.lex**

/\* Mini Calculator \*/

/\* calc.lex \*/

%{

#include "heading.h"

#include "tok.h"

int yyerror(char \*s);

int yylineno = 1;

%}

digit [0-9]

int\_const {digit}+

%%

{int\_const} { yylval.int\_val = atoi(yytext); return INTEGER\_LITERAL; }

"+" { yylval.op\_val = new std::string(yytext); return PLUS; }

"\*" { yylval.op\_val = new std::string(yytext); return MULT; }

[ \t]\* {}

[\n] { yylineno++; }

. { std::cerr << "SCANNER "; yyerror(""); exit(1); }

**calc.y**

/\* Mini Calculator \*/

/\* calc.y \*/

%{

#include "heading.h"

int yyerror(char \*s);

int yylex(void);

%}

%union{

int int\_val;

string\* op\_val;

}

%start input

%token <int\_val> INTEGER\_LITERAL

%type <int\_val> exp

%left PLUS

%left MULT

%%

input: /\* empty \*/

| exp { cout << "Result: " << $1 << endl; }

;

exp: INTEGER\_LITERAL { $$ = $1; }

| exp PLUS exp { $$ = $1 + $3; }

| exp MULT exp { $$ = $1 \* $3; }

;

%%

int yyerror(string s)

{

extern int yylineno; // defined and maintained in lex.c

extern char \*yytext; // defined and maintained in lex.c

cerr << "ERROR: " << s << " at symbol \"" << yytext;

cerr << "\" on line " << yylineno << endl;

exit(1);

}

int yyerror(char \*s)

{

return yyerror(string(s));

}

**Makefile:**

# Makefile

OBJS = bison.o lex.o main.o

CC = g++

CFLAGS = -g -Wall -ansi -pedantic

calc: $(OBJS)

$(CC) $(CFLAGS) $(OBJS) -o calc -lfl

lex.o: lex.c

$(CC) $(CFLAGS) -c lex.c -o lex.o

lex.c: calc.lex

flex calc.lex

cp lex.yy.c lex.c

bison.o: bison.c

$(CC) $(CFLAGS) -c bison.c -o bison.o

bison.c: calc.y

bison -d -v calc.y

cp calc.tab.c bison.c

cmp -s calc.tab.h tok.h || cp calc.tab.h tok.h

main.o: main.cc

$(CC) $(CFLAGS) -c main.cc -o main.o

lex.o yac.o main.o : heading.h

lex.o main.o : tok.h

clean:

rm -f \*.o \*~ lex.c lex.yy.c bison.c tok.h calc.tab.c calc.tab.h calc.output calc

**CONCLUSION:-**

Thus we have successfully implemented parser program using BISON tool.

**EXPERIMENT NO-3**

**AIM:** To Find FIRST(), FOLLOW() set of given grammar.

**THEORY:**

**GRAMMAR**

For programming languages, grammar is synonymous with context-free grammar (CFG). The CFG is not only used to define the syntax of a programming language, it is also used as input to compiler writing tools (such as yacc). The alternatives for grammars for programming languages are in the [Chomsky hierarchy](http://en.wikipedia.org/wiki/Chomsky_hierarchy)of rewriting systems. In addition to the Type 2 (context-free) grammars, Type 3 (regular grammars) are frequently used, especially in describing parsing algorithms.

**FIRST():**

If a is any string of grammar symbols, let FIRST(a) be the set of terminals that begin the strings derived from a. If a Þ e then e is also in FIRST(a).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set:

1. If X is a terminal **then** First(X) is just X!
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production X → Y1Y2..Yk**then** add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is **either**
   1. First(Y1) (if First(Y1) doesn't contain ε)
   2. **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)
   3. If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

Now, we can compute FIRST for any string X1X2 . . . Xn as follows. Add to FIRST(X1X2 ... Xn) all the none symbols of FIRST(X1). Also add the non-e symbols of FIRST(X2) if e is in FIRST(X1), the non-e symbols of FIRST(X3) if e is in both FIRST(X1) and FIRST(X2), and so on. Finally, add e to FIRST(X1X2 ... Xn) if, for all i, FIRST(Xi ) contains e.

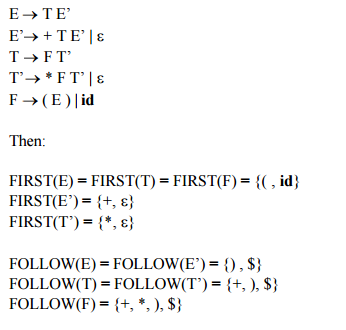
**FOLLOW ()**

FOLLOW(A), for non-terminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form SÞaAab for some a and b. Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived e and disappeared. If A can be the rightmost symbol in some sentential form, then $, representing the input right endmarker, is in FOLLOW(A).

To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. 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).
3. If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

**EXAMPLE:**

Consider the expression grammar given below:

**CONCLUSION:-**

Thus we have successfully implemented program to calculate FIRST and FOLLOW.

**EXPERIMENT NO-4**

**AIM:** To removing left recursion using direct as well as indirect method for given the set of production rule.

**THEORY:**

Left recursion is a special case of recursion. In terms of context-free grammar, a non-terminal r is left-recursive if the left-most symbol in any of r’s productions (‘alternatives’) either immediately (direct/immediate left-recursive) or through some other non-terminal definitions (indirect/hidden left-recursive) rewrites to r again.

A grammar is left-recursive if there exists a non-terminal symbol A that can derive to a sentential form with itself as the left-most symbol. Symbolically,

 A \Rightarrow^* A\alpha,

where \alpha is any sequence of terminal and non-terminal symbols.

### Direct left recursion

Immediate left recursion occurs in rules of the form

A \to A\alpha \mid \beta

where \alpha and \beta are sequences of nonterminals and terminals, and \beta doesn't start with A. For example, the rule

\mathit{Expr} \to \mathit{Expr} + \mathit{Term}

is immediately left-recursive. The [*recursive descent parser*](http://en.wikipedia.org/wiki/Recursive_descent_parser) for this rule might look like:

function Expr()

{

Expr(); match('+'); Term();

}

and a recursive descent parser would fall into infinite recursion when trying to parse a grammar which contains this rule.

### Indirect left recursion

Indirect left recursion in its simplest form could be defined as:

A \to B\alpha \mid C

B \to A\beta \mid D,

possibly giving the derivation A \Rightarrow B\alpha \Rightarrow A\beta\alpha \Rightarrow \ldots 

More generally, for the nonterminals A_0, A_1, \ldots, A_n, indirect left recursion can be defined as being of the form:

A_0 \to A_1\alpha_1 \mid \ldots

A_1 \to A_2\alpha_2 \mid \ldots

\cdots

A_n \to A_0\alpha_{n+1} \mid \ldots

where \alpha_1, \alpha_2, \ldots, \alpha_n are sequences of nonterminals and terminals.

**CONCLUSION:-**

Thus we have successfully implemented program to remove left recursion using direct and indirect method.

**EXPERIMENT NO.5**

**AIM:** To Design and implement two pass assembler for X86 machine.

**THEORY:**

An assembler is a language translator that takes as input assembly language program & generates its machine language equivalent along with the information required by the loader.

ASSEMBLER

ALP Machine language equivalent + information required by loader

DATABASE

The rules in ALP state that the symbol should be defined somewhere in the course of the program .Hence there may be some cases in which the reference to the symbol is made prior to its definition & such a reference is called as Forward Reference . Due to Forward Reference the assembler cannot assemble the instructions & such a problem is called as Forward Reference Problem.

To resolve Forward Reference Problem we make two passes over the input program.

The purpose of Pass1 is to define the symbols & literals encountered in the program.

The purpose of Pass2 is to assemble the instructions & assemble the data.

The database consists of the following tables:

**MOT [Machine Opcode Table]**

|  |  |  |  |
| --- | --- | --- | --- |
| Mnemonic opcode | Binary opcode | Instruction length | Instruction format |
|  |  |  |  |
|  |  |  |  |

MOT is a fixed length table i.e. no entries are made in it in either of the passes.

In Pass1 it is consulted to obtain instruction length [update LC].

In Pass2 it is consulted to obtain binary opcode, instruction length & instruction format.

**POT [Pseudo Opcode table]**

|  |  |
| --- | --- |
| Pseudo opcode | Address of routine to process pseudo opcode |
|  |  |
|  |  |

POT is a fixed length table ie no entries are made in it in either of the passes.

In both the passes it is consulted to process pseudo opcodes like START, USING, DROP, DC, DS, etc

**ST [Symbol Table]**

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Value | Length | Relocation [R/A] |
|  |  |  |  |
|  |  |  |  |

ST is used for keeping track of symbols that are defined in the program.

In Pass1 whenever a symbol is defined an entry is made in the symbol table.

In Pass2 ST is used for generating the address of the symbols.

**LT [Literal Table]**

|  |  |  |  |
| --- | --- | --- | --- |
| Literal | Value | Length | Relocation [R/A] |
|  |  |  |  |
|  |  |  |  |

LT is used for keeping track of symbols that are defined in the program.

In Pass1 whenever a literal is defined an entry is made in the LT.

In Pass2 LT is used for generating the address of the literals.

**BT [Base Table]**

|  |  |  |
| --- | --- | --- |
|  | Availability indicator | Contents of base register |
| 1 | ‘n’ |  |
| : |  |  |
| 15 | ‘n’ |  |

BT is used to keep track on the contents of the base register.

In Pass1 BT is not required.

In Pass2 BT is used for processing USING & DROP statements.

**CONCLUSION:-**

Thus we have successfully implemented Assembler program.**EXPERIMENT NO-6**

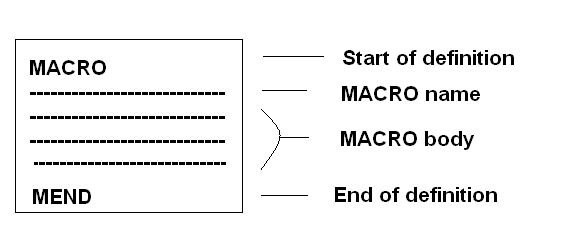
**AIM:** To design and implement Macro Processor.

**THEORY:**

The assembly language programmer generally finds that certain instructions are repeated in the programs.

Instead of writing these groups of instructions again and again the programmer can take the advantage of macro facility where macro is defined to be single line abbreviation for group of instructions

The format for defining the macro is as follows

****

Suppose the requirements of the programmer is as shown

**|**

**|**

**|**

**A1,data**

**A2,data**

**A3,data**

**|**

**|**

**|**

**A1,data**

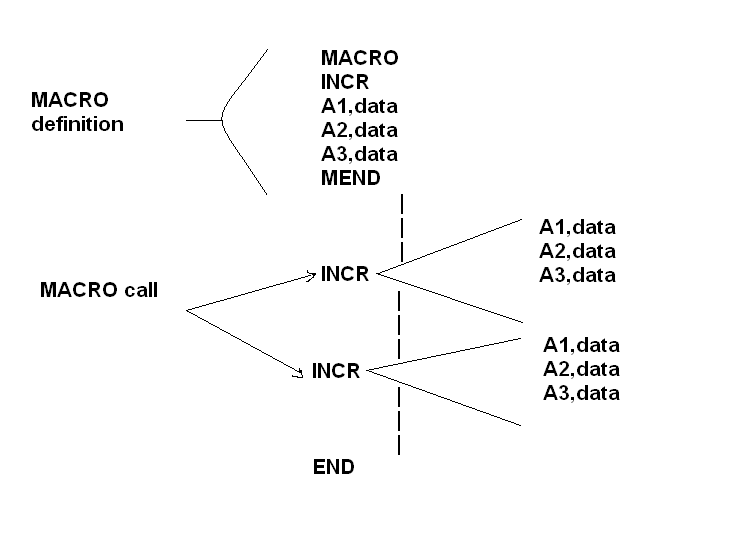
**A2,data**

**A3,data**

**|**

**|**

The above requirement can be achieved by using the macro facility as follows



**DEFINITION:**

It is a program which is responsible for the processing of the macro.

**FUNCTION:**

* + 1. Recognize the macro definition.
    2. Store the macro definition.
    3. Recognize the macro call.
    4. Perform macro expansion.

**Design of two pass MACRO PROCESSOR:-**

Since the macro call can appear before its macro definition,we have to make two passes over the input program.

**Purpose of pass 1:-**

The purpose of pass 1 is to Recognize the macro definition and Store the macro definition.

**Purpose of pass 2:-**

The purpose of pass 2 is to Recognize the macro call and Perform macro expansion.

**Format of database:-**

1. **MACRO NAME TABLE:-**

|  |  |  |
| --- | --- | --- |
| **INDEX** | **MACRO NAME** | **MDT INDEX** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| . |  |  |
| . |  |  |
| . |  |  |
| . |  |  |

Used in pass 1 for storing the macro name along with MDT index.

Used in pass 2 for recognizing the macro call.

2.**MACRO DEFINITION TABLE:-**

|  |  |
| --- | --- |
| **INDEX** | **MACRO DEFINITION** |
| 1 |  |
| 2 |  |
| 3 |  |
| . |  |
| . |  |
| . |  |

Used in pass 1 for storing the macro definition.

Used in pass 2 for performing the macro expansion.

3.**ARGUMENT LIST ARRAY**:-

|  |  |
| --- | --- |
| **INDEX** | **ARGUMENT** |
| 1 |  |
| 2 |  |
| 3 |  |
| . |  |
| . |  |
| . |  |

Used in pass 1 for replacing the formal parameter by index notation.

Used in pass 2 for replacing the index notations by actual parameter.

**FLOWCHART:**

Write in copy file

MDTC=MDTC+1

Enter MACRO name card in MDT

Prepare ALA

MNTC=MNTC+1

Enter the MACRO name along with current value of MDTC in MNT entry number MNTC

Read next card

Goto PASS 2

END pseudo op?

MACRO pseudo op?

PASS 1

MNTC=1 MDTC=1

MDTC=1

Read next card

MEND pseudo op?

MDTC=MDTC+1

Enter line in MDT

Substitute index notation for argument

Read next card

Pass 2

Read from copy file

Write in ESC

Supply ESC to assembler for further processing

Write in ESC

Substitute argument from MACRO call

Get line from MDT

MDTP=MDTP+1

Setup ALA

MDTP=MDT index from MNT

MACRO name found?

END pseudo op?

MEND pseudo op?

Search MNT for match with operation code

**CONCLUSION:-**

Thus we have successfully implemented MacroProcessor program.

**EXPERIMENT NO-7**

**AIM:** To implement Syntax Analysis using operator precedence parser

**THEORY:**

**SYNTAX ANALYSIS**

Syntax Analysis is alternatively known as parsing. It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning).

The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc.

The parser takes the tokens produced during the lexical analysis stage, and attempts to build some kind of in-memory structure to represent that input. Frequently, that structure is an 'abstract syntax tree' (AST).

The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a *grammar*. A grammar is a set of rules (or *productions*) that specifies the syntax of the language (i.e. what is a valid sentence in the language).

There can be more than one grammar for a given language. Furthermore, it is easier to build parsers for some grammars than for others.

**Operator Precedence:**

Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars.

Operator Grammars have the property that no production right side is empty or has two adjacent non-terminals.

**Operator Precedence Parsing Algorithm**

Initialize: Set ip to point to the first symbol of the input string w$

Repeat: Let b be the top stack symbol, a the input symbol pointed to by ip

if (a is $ and b is $)

return

else

if a ·> b or a =· b then

push a onto the stack

advance ip to the next input symbol

else if a <- b then

repeat

c <- pop the stack

until (c-> stack-top)

else error

end

**Consider:**

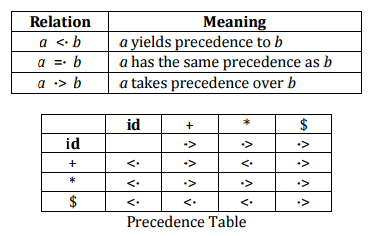
E-> E op E | id

op -> + | \*

Not an operator grammar but:

E-> E + E | E \* E | id

This parser relies on the following three precedence relations:



**Example**: The input string:

id1 + id2 \* id3

After inserting precedence relations becomes:

$ <- id1 -> + <- id2 -> \* <- id3 -> $

**CONCLUSION:-**

Thus we have successfully implemented operator precedence parser program.

**EXPERIMENT NO. 8**

**AIM**: To generate target code for the code optimized, considering the

target machine to be X86

**THEORY:**

Code generation is the process by which a compilers code generator converts some internal representation of source code into a form [e.g. machine code] that can be readily executed by machine [often a computer].

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multistage process is used because many algorithm for code optimization are easy to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages[the backend] needs to change from target to target.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into linear sequence of instructions, usually in an intermediate language such as three address code. Further stage of compilation may or may not be referred to as “code generation” depending on whether they involve a significant change in the representation of the program.

**Major Task in Code Generation**

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way. The generator may try to use faster instructions, use fewer instructions, exploit available registers and avoid redundant computations.

Tasks which are typically part of compilers “code generation” phase includes:

* **Instruction Selection**: which instructions to use
* **Instruction Scheduling**: in which order to put those instructions. Scheduling is a speed optimization that can have critical effect to pipelined machines.
* **Register Allocation**: the allocation of variables to processor registers.

In a compiler that uses an intermediate language, there may be two language selection stage- one to convert the parse tree into intermediate code and a second phase much later to convert the intermediate code into the ISA of the target machine.

The second phase does not require a tree traversal; it can be done linearly and typically involves a simple replacement of intermediate language operations with their corresponding opcodes.

**EXAMPLE:**

Expression: a=b+c\*60

Syntax tree:

=

(id1,1)

+

\*

(id3,3)

60

(id2,2)

Three Address Code

t1= int to float(60)

t2=id3 \* t1

t3=id2 + t2

id1=t3

t1=id3 \* int to float(60)

id1=id2 + t1

Code Generated

LDF r1 60.0

LDF r2, id3

MULF r2, r1

LDF r3, id2

ADDF r3, r2

STF id1,r3

**CONCLUSION:-**

Thus we have successfully implemented code generation program.

**EXPERIMENT NO-9**

**AIM:** To implement Code Optimization.

**THEORY:**

It attempts to improve the intermediate representation so that a faster running machine must be generated 

**MACHINE DEPENDENT OPTIMISATION**

Machine instructions that use registers as operands must be selected since such instructions are faster than the instructions that refer to location in memory

1. consecutive instructions that involve different functional units could be executed at same time 

**FUNCTIONAL PRESENTING TRANSFORMATION**

There are 2 classes within this

Common sub expression elimination C is called common sub expression if C was previously computed

& the values of variables in E have not changed since previous computation

We can avoid this calculation again if we use the previously computed value

Before optimization:

E1:=u\*i

x:=a[+1]

t2:=u\*i

y:=b[+2]

If there are images in this attachment, they will not be displayed.  [Download the original attachment](http://mail.google.com/mail/?view=att&th=1284aa4f01152c94&attid=0.1&disp=attd&zw)

after optimization:

t1:=u\*i

x:=a[+1]

y:=b[+2]

**LOOP OPTIMISATION:**

Running time of program can be improved if we decrease the number of instructions in an inner loop

Even if we increase the amount of code outside that loop eg. Frequency reduction

This transformation takes an error that yield the same result independently of the no. of times a loop

Is executed & places that expression before a loop

BEFORE OPTIMISATION:

While (i<=limit-2)

AFTER OPTIMISATION:

t=limit-2

While ( t<=2)

**CONCLUSION:-**

Thus we have successfully implemented code optimization program.

**EXPERIMENT NO-10**

**AIM:** To create shared library in Linux environment and using it.

**THEORY:**

A library is a file containing compiled code from various object files stuffed into a single file. It may contain a group of functions that are used in a particular context. For example, the ‘pthread’ library is used when thread related functions are to be used in the program.  
  
Broadly, a library (or Program Library) can be of two types :

1. Shared Library
2. Static Library

There are mainly three standard locations in the filesystem where a library can be placed.

* /lib
* /usr/lib
* /usr/local/lib

### Example (How to Create a Shared Library)

Let’s take a simple practical example to see how we can create and use shared libraries. The following is the piece of code (shared.c) that we want to put in a shared library :

#include "shared.h"

unsigned int add(unsigned int a, unsigned int b)

{

printf("\n Inside add()\n");

return (a+b);

}

shared.h looks like :

#include<stdio.h>

extern unsigned int add(unsigned int a, unsigned int b);

How to make shared.c as a shared library.

1. Run the following two commands to create a shared library :

gcc -c -Wall -Werror -fPIC shared.c

gcc -shared -o libshared.so shared.o

The first command compiles the code shared.c into position independent code which is required for a shared library.  
The second command actually creates a shared library with name ‘libshared.so’.

2. Here is the code of the program that uses the shared library function ‘add()’

#include<stdio.h>

#include"shared.h"

int main(void)

{

unsigned int a = 1;

unsigned int b = 2;

unsigned int result = 0;

result = add(a,b);

printf("\n The result is [%u]\n",result);

return 0;

}

3. Next, run the following command :

gcc -L/home/himanshu/practice/ -Wall main.c -o main -lshared

This command compiles the main.c code and tells gcc to link the code with shared library libshared.so (by using flag -l) and also tells the location of shared file(by using flag -L).

4. Now, export the path where the newly created shared library is kept by using the following command :

export LD\_LIBRARY\_PATH=/home/himanshu/practice:$LD\_LIBRARY\_PATH

The above command exports the path to the environment variable ‘LD\_LIBRARY\_PATH’.

5. Now run the executable ‘main’ :

# ./main

Inside add()

The result is [3]

So we see that shared library was loaded and the add function inside it was executed.

**CONCLUSION:-**

Thus we have created shared library in Linux and used it in other program.