SPC LAB MANUAL

Experiment No 1

Aim: Write a program to implement Two Pass Assembler

Theory:

Algorithm for pass 1:

1. Start

2.Intializelocation counter to zero

3.Read opcode field of next instruction.

4.search opcode in pseudo opcode Table(POT)

5.If opcode is found in POT

5.1 If it is ‘DS’ or ‘DC’

Adjust location counter to proper alignment.

Assign length of data field to ‘L’

Go to step 9

5.2 If it is ‘EQU’

Evaluate operand field

Assign values to symbol in label field

Go to step 3

5.3 If it is ‘USING’ or ‘DROP’ Go to step 3

5.4 If it is ‘END’

Assign value to symbol in label field

Go to step 3

6. Search opcode in Machine Opcode Table.

7. Assign its length to ‘L’.

8. Process any literals and enter them into literal Table.

9.If symbol is there in the label field

Assign current value of Location Counter to symbol

10. Location Counter= Location Counter +L.

11.Go to step 3.

12. Stop.

Algorithm for Pass2:

1.Start

2.Intialize location counter to zero.

3. Read opcode field of next instruction.

4.Search opcode in pseudo opcode table.

5.If opcode is found in pseudo opcode Table

5.1 If it is ‘DS’ or ‘DC’

Adjust location counter to proper alignment.

If it is ‘DC’ opcode form constant and insert in assembled program

Assign length of data field to ‘L’

Go to step 6.4

5.2 If it is ‘EQU’ or ‘START’ ignore it. Go to step 3

5.3 If it is ‘USING’

Evaluate operand and enter base reg no. and value into base table

Go to step 3

5.4 If it is ‘DROP’

Indicate base reg no . available in base table . Go to step 3

5.5 If it is ‘END’

Generate literals for entries in Literal Table

Go to step 12

6 Search opcode in MOT

7. Get opcode byte and format code

8. Assign its length to ‘L’.

9. Check type of instruction.

10.If it is type ‘RR’ type

10.1 Evaluate both register expressions and insert into second byte.

10.2 Assemble instruction

10.3 Location Counter= Location Counter +L.

10.4.Go to step 3.

11. If it is ‘RX’ type

11.1 Evaluate register and index expressions and insert into second byte.

11.2 Calculate effective address of operand.

11.3 Determine appropriate displacement and base register

11.4 Put base and displacement into bytes 3 and 4

11.5 Location Counter= Location Counter +L.

11.6 Go to step 11.2

13 Stop.

Conclusion:

Experiment No 2

Aim: Write a program to implement two pass Macro Processor

Theory:

Algorithm for pass1:

1. Set the MDTC (Macro Definition Table Counter) to 1.

2. Set MNTC (Macro Name Table counter) to 1.

3. Read next statement from source program.

4. If this source statement is pseudo-opcode MACRO (start of macro definition)

5. Read next statement from source program (macro name line)

6. Enter Macro name found in step 5 in name field of MNT (macro name table)

7. Increment MNTC by 1.

8. Prepare ALA

9. Enter macro name into MDT at index MDTC

10.Increment MDTC by 1.

11. Read source statement from source program

12. Create and substitute index notation for arguments in the source statement if any.

13. Enter this line into MDT

14. Increment MDTC by 1.

15. Check if currently read source statement is pseudo-opcode MEND. If yes then goto step 3 else goto step 11.

16. Write source program statement as it is in the file

17.Check if pseudo-opcode END is encountered. If yes goto step 18 else goto step 19

18. Goto Pass2

19. Go to step 3

20. End of PASS1.

Algorithm for pass2

1. Read next statement from source program

2. Search in MNT for match with operation code

3. If macro name found then goto step 4 else goto step 11.

4. Retrieve MDT index from MNT and store it in MDTP.

5. Set up argument list array

6. Increment MDTP by one.

7. Retrieve line pointer by MDTP from MDT

8. Substitute index notation by actual parameter from ALA if any.

9. Check if currently retrieved line is pseuodo-opcode MEND, if yes goto step 1 else goto step 10

10. Write statement formed in step 8 to expanded source file and goto step 6

11. Write source statement directly into expanded source file

12. Check if pseudo-opcode END encountered, if yes goto step 13 else goto step 1

13. End of PASS II

Implementation Details

1. Read input file with Macros
2. Display output of Pass1 as the output file, MDT, MNT, and ALA tables.
3. Display output of pass2 as the expanded source file, MDT, MNT and ALA tables.

Conclusion:

Experiment No 3

Aim : Write a program to implement simple parser using Lex and Yacc

Theory:

Parser for a grammar is a program which takes in the language string as its input and produces either a corresponding parse tree or a error.

Syntax of a Language

The rules which tells whether a string is a valid program or not are called the syntax

Semantic s of Language

The rules which give meaning to programs are called the semantic of a language

Tokens

When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the language.

Lexical Analysis

The function of lexical Analyzer is to read the input stream representing the source program , one character at a time and translate into valid tokens.

Implementation Details

1: Create a lex file

The general format for lex file consists of three sections:

1. Definitions
2. Rules
3. User subroutine Section

Definitions consists of any external ‘C’ definitions used in the lex actions or subroutines. The other types of definitions are definitions are lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations. The rules is the basic part which specifies the regular expressions and their corresponding actions. The user Subroutines are the functions that are used in the Lex actions.

2 : Yacc is the Utility which generates the function ‘yyparse’ which is indeed the Parser. Yacc describes a context free, LALR(1) grammer and supports both bottom up and top-down parsing. The general format for the yacc file is very similar to that of the lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In declarations apart from the legal ‘C’ declarations here are few Yacc specific declarations which begins with a % sign.

1. % union It defines the Stack type for the Parser.

   It is union of various datas/structures/objects.

2. % token These are the terminals returned by the yylex function to the yacc. A token cal also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as % token <stack member> tokenName.

3. %type The type of non-terminal symbol in the grammar rule can be specified with this. The format is %type <stack member> non termainal.

4. % noassoc Specifies that there is no associativity of a terminal symbol.

5. % left Specifies the left associativity of atermainal symbol.

6. % rightSpecifies the right associativity of a termainal symbol.

7. % start specifies the L.H.S. non-terminal symbol of a production rule which specifies starting point of grammar rules.

8. % prac changes the precedence level associated with a particular rule to that of the following token name or literal.

The Grammar rules are specified as follows:

Context free grammar production-

p-> AbC

Yacc Rule-

P: A b C { /* ‘C’ actions*/}

The general style foe coding the rules is to have all Terminals in upper –case ans all non-termainals in lower –case.

To facilitate a proper syntax directed translation the Yacc has something calles pseudo-variables which forms a bridge between the values of termainals/non-termainals and the

actions. These pseudo variables are $$, $1, $2, $3,……The $$ is the L.H.S value of the rule whereas $1 is the first R. H. S value of the rule ans so is the $2 etc. The default type for pseudo variables is integer unless they are specified by % type.

%token <type> etc.

Conclusion

Experiment No 4

Aim : Generate a target code for the optimized code.

Theory:

The final phase in compiler model is the the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

Code Generation Algorithm:

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 we perform following function:

1. Invoke a function getreg to determine the location L where the result of computation y op z should be stored. (L cab be a register or memory location .

2. Consult the address descriptor for y to determine y, the current locations of y. Prefer the register for y if the value of y is currently both in memory and register. If value of y is not alreadt in L , generate the instruction MOV y, L to place a copy of y in L.

3. Generate instruction po z, L where z is a current location of z. Again address descriptor of x to indicate that x is in location L. if L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptor.

4. If the current values of y and z have no next uses , are not live on exit from the block , and are in registers alter the register descriptor to indicate that , after execution of x=y op z, those register no longer will contain y and z, resply.

The function getreg:

The function getreg returns the location L to hold the values of x for the assignment x= y op z.

1.If the name y is in a reg that holds the value of no other names, and y is not live and has no next use after execution of x= y op z ,then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there os one.

3. Failing (2) , if X has a next use in the block, or op is an operator , such as indexing, that requires a register find an occupied register R. Store the values of R into a memory location (MOV R ,M) if it is not already in the proper memory location M, update the address

descriptor for M , and return R. if R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable register might be one whose data is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If x is not used in the block , or no suitable occupied register can found, select the memory location of x as L.

Conclusion:

Experiment No 5

Aim : Write a program to implement Lexical analyzer

Theory :

Lexical analytical is the process of converting a sequence of characters into a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called lexical analyzers or lexers. The specification of a programming language will include a set of rules, often expressed syntactically, specifying the set of possible character sequences that can form a token or lexeme. The whitespaces characters are often ignored during lexical analysis.

A token is a categorized block of text. The block of text corresponding to the token is known as a lexeme. A lexical analyzer processes lexemes to categorize them according to function , giving them meaning. This assignment of meaning is known as tokenization.

A token can look like anything ; it just needs to be useful part of the structured text. Consider this expression in the C programming language:

Tokens are as follows:

Token	Token Type
Sum	IDENT
=	OPERATOR
3	NUMBER
+	OPERATOR
2	NUMBER
;	SEMICOLON

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzerreads in a stream of characters, indetifiers the lexemes in the stream and categorized them into tokens. This is called tokenizing. If lexer finds an invalid token , it will report an error.

Conclusion:

Experiment No 6

Aim : Design recursive descent parser

Theory :

A **recursive descent parser** is a kind of top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

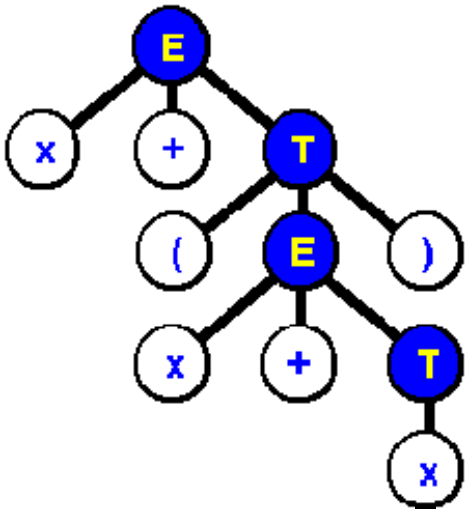
This parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input. Our recursive descent parsers will look ahead one character and advance the input stream reading pointer when proper matches occur.

What a recursive descent parser actually does is to perform a depth-first search of the derivation tree for the string being parsed. This provides the 'descent' portion of the name. The 'recursive' portion comes from the parser's form, a collection of recursive procedures.

As our first example, consider the simple grammar

E ® x+T
T ® (E)
T ® x

and the derivation tree in figure 2 for the expression x+(x+x)



Derivation Tree for x+(x+x)

A recursive descent parser traverses the tree by first calling a procedure to recognize an E. This procedure reads an 'x' and a '+' and then calls a procedure to recognize a T. This would look like the following routine.

```
Procedure E()
Begin
If (input_symbol='x') then
next();
If (input_symbol='+') then
Next();
T();
Else
Errorhandler();
END
```

Procedure for E

Note that the 'next' looks ahead and always provides the next character that will be read from the input stream. This feature is essential if we wish our parsers to be able to predict what is due to arrive as input.

Note that 'errorhandler' is a procedure that notifies the user that a syntax error has been made and then possibly terminates execution.

In order to recognize a T, the parser must figure out which of the productions to execute. This is not difficult and is done in the procedure that appears below.

```
Procedure T()
Begin
Begin
If (input_symbol='(') then
next();
E();
If (input_symbol=')') then
next();
end
else If (input_symbol='x') then
next();
else
Errorhandler();
END
```

Procedure for T

In the above routine, the parser determines whether T had the form (E) or x. If not then the error routine was called, otherwise the appropriate terminals and nonterminals were recognized.

Algorithm:

1. Make grammar suitable for parsing i.e. remove left recursion(if required).

2. Write a function for each production with error handler.

3. Given input is said to be valid if input is scanned completely and no error function is called.

Conclusion:

Experiment No 7

Aim: Study of different debugger tools

Theory:

In this experiment, students are required to study typical debugging process , Break points, Break point conditions, Inspection Tools, Watches , Local Variable Window , Call stack, stepping Controls , Dynamic recompilation, profiler.

Conclusion: