Assignment - Compiler Construction 1

TITLE	**Lexical analyzer for HLL using LEX**
PROBLEM STATEMENT /DEFINITION	Implement a Lexical Analyzer using LEX for a subset of C. Cross check your output with Stanford LEX.
OBJECTIVE	• Appreciate the role of lexical analysis phase in compilation • Understand the theory behind design of lexical analyzers and lexical analyzer generator • Be able to use LEX to generate lexical analyzers
S/W PACKAGES AND HARDWARE APPARATUS USED	64 bit latest Computer/ min.PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse 64 bit Linux with a support of LEX utility, GCC
REFERENCES	1. A V Aho, R. Sethi, .J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Education, ISBN 81 - 7758 - 590 2. J. R. Levine, T. Mason, D. Brown, "Lex & Yacc", O'Reilly, 2000, ISBN 81-7366 -061-X.– 8 3. K. Louden, "Compiler Construction: Principles and Practice", Thomson Brookes/Cole (ISE), 2003, ISBN 981 - 243 - 694-4 page no.31-90
STEPS	Refer theory, algorithm, test input, test output
INSTRUCTIONS FOR WRITING JOURNAL	Prepare LATEX document including following • Title • Problem Definition • Theory, mathematical model ,test cases • Algorithm • Source Code • Output • Conclusion

Theory:

Regular Expressions in lex

"..." - Any string enclosed in double-quotes shall represent the characters within the double-quotes as themselves, except that backslash escapes.
<state>r, <state1,state2,...>r - The regular expression r shall be matched only when the program is in one of the start conditions indicated by state, state1, and so on.
r/x - The regular expression r shall be matched only if it is followed by an occurrence of regular expression x

* - matches zero or more occurrences of the preceding expressions
[] - a character class which matches any character within the brackets. If the first character is a circumflex '^' it changes the meaning to match any character except the ones within brackets.
^ - matches the beginning of a line as the first characters of a regular expression. Also used for negation within square brackets.
{} - Indicates how many times the previous pattern is allowed to match, when containing one or two numbers.
$ - matches the end of a line as the last character of a regular expressions
\ - used to escape metacharacters and a part of usual c escape sequences e.g '\n' is a newline character
while '*' is a literal asterisk.
+ - matches one more occurrences of the preceding regular expression.
| - matches either the preeceeding regular expression or the following regular expression.
( ) - groups a series of regular expressions together, into a new regular expression.

Examples :

DIGIT [0-9]+
IDENTIFIER [a-zA-Z][a-zA-Z0-9]*

The functions or macros that are accessible to user code:

int yylex(void)

Performs lexical analysis on the input; this is the primary function generated by the lex utility. The function shall return zero when the end of input is reached; otherwise, it shall return non-zero values (tokens) determined by the actions that are selected.

int yymore(void)

When called, indicates that when the next input string is recognized, it is to be appended to the current value of yytext rather than replacing it; the value in yyleng shall be adjusted accordingly.

int yyless(int n)

Retains n initial characters in yytext, NUL-terminated, and treats the remaining characters as if they had not been read;the value in yyleng shall be adjusted accordingly.

int yywrap(void)

Called by yylex() at end-of-file; the default yywrap() shall always return 1. If the application requires yylex() to continue processing with another source of input, then the application can include a function yywrap(), which associates another file with the external variable FILE * yyin and shall return a value of zero.

int main(int argc, char *argv[])

Calls yylex() to perform lexical analysis, then exits. The user code can contain main() to perform application-specific operations, calling yylex() as applicable.

Algorithm:

1. Accept input filename (file is in HLL) as a command line argument.
2. Separate the tokens as identifiers, constants, keywords etc. and fill the generic symbol table.
3. Check for lexical errors and give error messages if needed.

Test Input:

```
#include<stdio.h>
void main()
{
 int a,b;
char bilateral;
a=3;
 c=a+d;
}
```

Test Output:

Lexeme	**Token**
#include<stdio.h>	Directive
void	Reserved
main()	Reserved
{	Punctuation
int	Keyword
a	Identifier
b	Identifier
;	Delimiter
char	Reserved
bilateral	**Lexical error** (as Identifier is more than 8 characters)
;	Delimiter
a	Identifier
=	Operator
3	Constant
;	Delimiter
c	Identifier
=	Operator
a	Identifier
+	Operator
d	Identifier
;	Delimiter
}	Punctuation

Symbol Table:

Symbol name	Symbol Type	Symbol Value
a		
b		
c		
d		

FAQ's

1. What is the data structure used in Lexical phase of compiler
2. What are lexical errors

Practice Problem Statements:

1. Implement lexical analyser for JAVA along-with error handling

Assignment - Compiler Construction 2

TITLE	**Representing Ambiguous grammar using LEX and YACC**
TOPIC for PROBLEM STATEMENT /DEFINITION	Implement a parser for an expression grammar using YACC and LEX for the subset of C. Cross check your output with Stanford LEX and YACC
OBJECTIVE	• To understand basic syntax of YACC specifications, built-in functions and Variables • Be proficient on writing grammars to specify syntax • Understand the theories behind different parsing strategies-their strengths and limitations • Understand how the generation of parser can be automated • Be able to use YACC to generate parsers
S/W PACKAGES AND HARDWARE APPARATUS USED	64 bit latest Computer/ min.PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse 64 bit Linux with a support of LEX and YACC utility, GCC
REFERENCES	1. A V Aho, R. Sethi, .J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Education, ISBN 81 - 7758 - 590 2. J. R. Levine, T. Mason, D. Brown, "Lex & Yacc", O'Reilly, 2000, ISBN 81-7366 -061-X.– 8 3. K. Louden, "Compiler Construction: Principles and Practice", Thomson Brookes/Cole (ISE), 2003, ISBN 981 - 243 - 694-4
STEPS	Refer to theory, algorithm, test input, test output
INSTRUCTIONS FOR WRITING JOURNAL	Prepare LATEX document including following • Title • Problem Definition • Theory, mathematical model ,test cases • Source Code • Output • Conclusion

Theory:

Ambiguity:

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation).
The context free grammar

$$A \rightarrow A + A \mid A - A \mid a$$

is ambiguous since there are two leftmost derivations for the string a + a + a:

$A \rightarrow A + A$	$A \rightarrow A + A$
$\rightarrow a + A$	$\rightarrow A + A + A$
$\rightarrow a + A + A$	$\rightarrow a + A + A$
$\rightarrow a + a + A$	$\rightarrow a + a + A$
$\rightarrow a + a + a$	$\rightarrow a + a + a$

Conflicts may arise because of mistakes in input or logic, or because the grammar rules. This will result in a shift/reduce conflict. When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.
Yacc invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of **precedence levels** for operators, together with information about left or right associativity.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength.

Thus, %left '+' '-'
%left '*' '/'

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators that may not associate with themselves.

Passing Values between Actions

To get values generated by other actions, an action can use the **yacc** parameter keywords that begin with a dollar sign ($1, $2, ...). These keywords refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is:

A : B C D ;

then $1 has the value returned by the rule that recognized B, $2 has the value returned by the rule that recognized C, and $3 the value returned by the rule that recognized D.

To return a value, the action sets the pseudo-variable $$ to some value. For example, the following action returns a value of 1:

{$$ = 1;}

Algorithm:

1. Write lex code to separate out the tokens from arithmetic expression.
2. Write YACC code to check the syntax of the arithmetic expression
3. Accept a input arithmetic expression from the user
4. If input is as per syntax ,evaluate the expression

Test Input:

Arithmetic Expression: 10+10*50

Test Output:

Syntax of Arithmetic Expression is correct

Evaluated Arithmetic Expression: 510

FAQ's

1. Is the grammar to be written naturally always?
2. How the ambiguity in the grammar is resolved?
3. What is verbose option of YACC?

Practice Problem Statements:

1. Write a Yacc program that will take arithmetic expression as input and produce corresponding postfix expression as output and also evaluate that expression.

2. Write a yacc program “desk calculator” which remind, store and clear the result. [Hint: use file to store the value of expression]

3. Write a yacc program “desk calculator” that will evaluate Boolean expressions.

4. Implement text calculator which will handle arithmetic statements using LEX and YACC. Explain the parsing table constructed.

5. Generate the LALR parsing table using LEX and YACC for the following grammar
   S->Aa|bAc|dc|bda
   A->d

6. Implement bottom up parser for
   S->aSa|bSb|c using LEX and YACC

Assignment - Compiler Construction 3

TITLE	Generate and Populate appropriate Symbol Table.
PROBLEM STATEMENT /DEFINITION	Design suitable data structures to generate and Populate appropriate Symbol Table.
OBJECTIVE	- Understand the Symbol table is an important data structure created and maintained by compilers
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Compiler Construction Using Java, JavaCC and Yacc, Anthony J. Dos Reis, Wiley ISBN 978-0-470-94959-7 2. K Muneeswaran, "Compiler Design", Oxford University press, ISBN 0-19-806664-3 3. J R Levin, T Mason, D Brown, "Lex and Yacc", O'Reilly, 2000 ISBN 81-7366-061-X 4. A V Aho, R Sethi, J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Edition, ISBN 81-7758-590-8 5. Dick Grune, Bal, Jacobs, Langendoen, Modern Compiler Design, Wiley, ISBN 81-265-0418-8
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Date • Title • Problem Definition • Learning Objective • Learning Outcome • Theory-Related concept,Architecture,Syntax etc • Class Diagram • Test cases

	• Program Listing • Output • Conclusion

Aim: Generate and Populate appropriate Symbol Table using suitable data structures
Pre-requisite: Basic functionalities of Symbol Table
Learning Objectives:

Understand various phases of compiler

Items stored in Symbol table

Information used by compiler from Symbol table

Learning Outcomes:

The students will be able to

- Generate Symbol Table using suitable data structures
- Study of Symbol Table

Theory:
A *Symbol table* is a data structure used by the compiler, where each identifier in program's source code is stored along with information associated with it relating to its declaration. It stores identifier as well as it's associated attributes like scope, type, line-number of occurrence, etc.
Symbol table can be implemented using various data structures like:

- LinkedList
- Hash Table
- Tree
- Binary Search Tree

A common data structure used to implement a symbol table is HashTable.

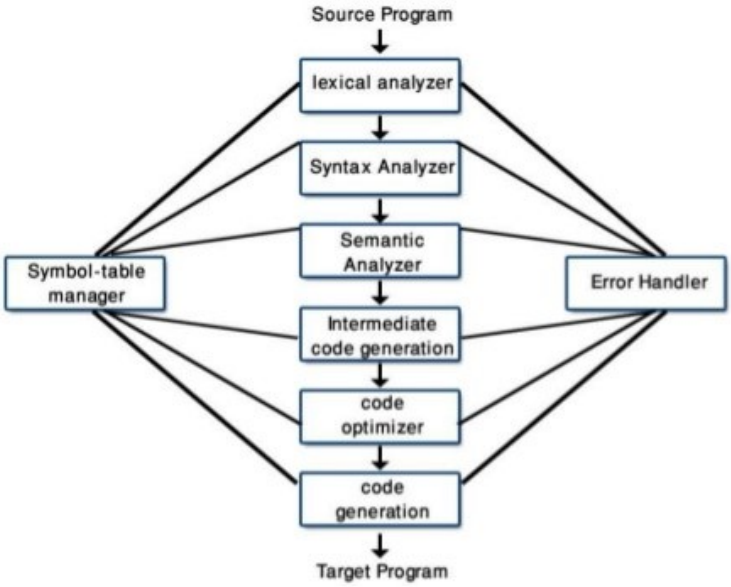
Implementation of Symbol table –
Following are commonly used data structure for implementing symbol table :-

1. **List –**
   - In this method, an array is used to store names and associated information.
   - A pointer **"available"** is maintained at end of all stored records and new names are added in the order as they arrive
   - To search for a name we start from beginning of list till available pointer and if not found we get an error **"use of undeclared name"**

- While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. **"Multiple defined name"**
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- Advantage is that it takes minimum amount of space.

2. **Linked List –**
   - This implementation is using linked list. A link field is added to each record.
   - Searching of names is done in order pointed by link of link field.
   - A pointer **"First"** is maintained to point to first record of symbol table.
   - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
3. **Hash Table –**
   - In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
   - A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.
   - To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.
   - Insertion and lookup can be made very fast – O(1).
   - Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.
4. **Binary Search Tree –**
   - Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
   - All names are created as child of root node that always follow the property of binary search tree.
   - Insertion and lookup are $O(\log_2 n)$ on average.

1.3 The Phases of a Compiler:



40

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

FAQs:

Which data structure is used in symbol table?

What is the role of symbol table in compiler design?

What is the purpose of symbol?

Oral/Review Questions:

Define Symbol Table.

What Is A Symbol Table?

List The Different Storage Allocation Strategies

Assignment - Compiler Construction 4

ASSIGNMENT NUMBER: Elective-III Compiler 4

TITLE	Implementation of Semantic Analysis Operations
PROBLEM STATEMENT /DEFINITION	Implement Semantic Analysis Operations (like type checking, verification of function parameters, variable declarations and coercions) possibly using an Attributed Translation Grammar.
OBJECTIVE	- **Semantic analysis** is the task of ensuring that the declarations and statements of a program are semantically correct, i.e, that their **meaning** is clear and consistent with the way in which control structures and data types are supposed to be used.
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Compiler Construction Using Java, JavaCC and Yacc, Anthony J. Dos Reis, Wiley ISBN 978-0-470-94959-7 2. K Muneeswaran, "Compiler Design", Oxford University press,

	ISBN 0-19-806664-3 3. J R Levin, T Mason, D Brown, "Lex and Yacc", O'Reilly, 2000 ISBN 81-7366-061-X 4. A V Aho, R Sethi, J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Edition, ISBN 81-7758-590-8 5. Dick Grune, Bal, Jacobs, Langendoen, Modern Compiler Design, Wiley, ISBN 81-265-0418-8
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Date • Title • Problem Definition • Learning Objective • Learning Outcome • Theory-Related concept,Architecture,Syntax etc • Class Diagram • Test cases • Program Listing • Output • Conclusion

Aim: Implementation of Semantic Analysis Operations
Pre-requisite: familiarity with the phases of compiler like lexical analysis i.e. parser
Learning Objectives:

- Study of Semantics analysis phase of compiler how it helps interpret symbols, their types, and their relations with each other.

- Study of Grammar used by Semantics analysis to interpret meaning of programming and natural language sentences.

Learning Outcomes:

The students will be able to

- Construct meaningful sentence in programming and natural language

- Able to understand how semantic analysis phase of compiler behaves.

Theory:

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

```
CFG + semantic rules = Syntax Directed Definitions
```

For example:

```
int a = "value";
```

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking
-

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

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

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

```
S → ABC
```

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

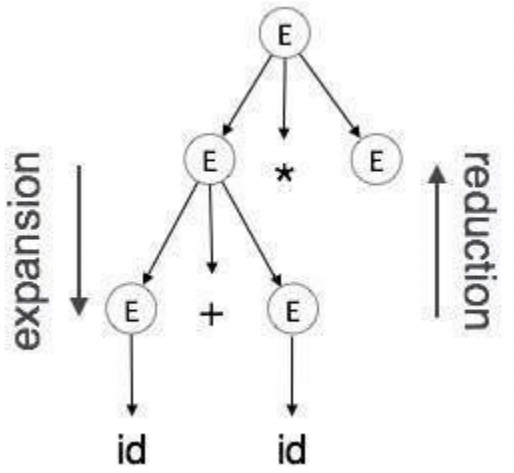
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → ABC
```

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

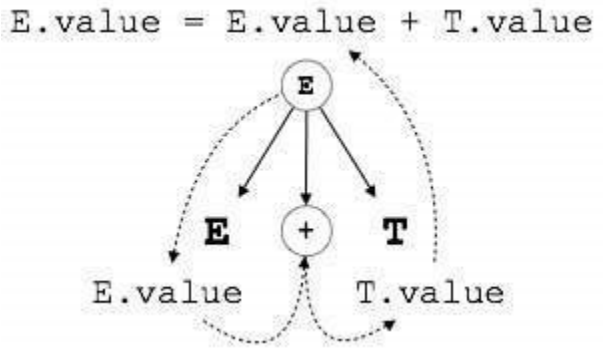
For example:

```
int value  = 5;

<type, “integer”>

<presentvalue, “5”>
```

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

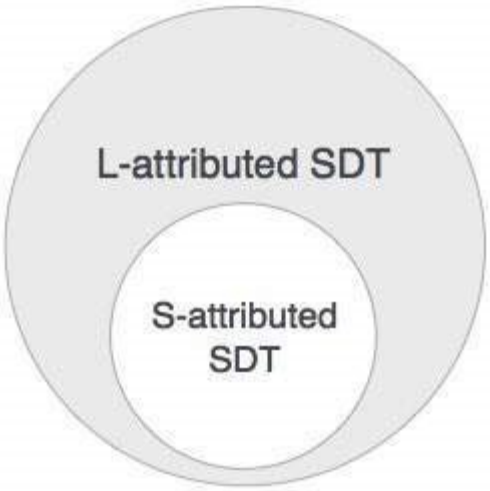
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

FAQ:

What is semantic analysis in compiler?

What is semantic analysis in natural language processing?

What is semantic text analysis?

What is syntactic and semantic analysis?

What is semantic analysis in system programming?

What is semantic error in compiler design?

Assignment - Compiler Construction 5

TITLE	Implement the front end of a compiler that generates the three address code for a simple language.
PROBLEM STATEMENT /DEFINITION	1. Write a LEX and YACC program to generate Intermediate Code for arithmetic expression 2. Write a LEX and YACC program to generate Intermediate Code for subset of C (If,else,while)

OBJECTIVE	• To understand the fourth phase of a compiler: Intermediate code generation (ICG) • To learn and use compiler writing tools. • Understand and learn how to write three address code for given statement.
S/W PACKAGES AND HARDWARE APPARATUS USED	Linux OS (Fedora 20), PC with the configuration as 64-bit Fedora or equivalent OS with 64-bit Intel-i5/ i7 or latest higher processor computers, FOSS tools, LEX, YACC,
REFERENCES	1. A V Aho, R. Sethi, .J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Education, ISBN 81 - 7758 - 590 2. J. R. Levine, T. Mason, D. Brown, "Lex & Yacc", O'Reilly, 2000, ISBN 81-7366 -061-X.– 8 3. K. Louden, "Compiler Construction: Principles and Practice", Thomson Brookes/Cole (ISE), 2003, ISBN 981 - 243 - 694-4 page no.31-90
STEPS	Refer theory, algorithm, test input, test output
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Theory and mathematical model • State transition diagram • Algorithm • Source Code • Output • Conclusion

THEORY:

Introduction

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of

the target machine to the back end. The front end translates a source program into an intermediate representation from which the back end generates target code. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: m x n compilers can be built by writing just m front ends and n back ends.

Benefits of using a machine-independent intermediate form are:

1. Compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Three ways of intermediate representation:

- Syntax tree
- Postfix notation
4. Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating post-fix notation.

Three-address code:

Each statement generally consists of 3 addresses, 2 for operands and 1 for result.
X:=Y op Z where X,Y,Z are variables, constants or compiler generated variables.

Advantages of three-address code:

Complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
The use of names for the intermediate values computed by a program allows three address codes to be easily rearranged – unlike post-fix notation.
Three-address code is a liberalized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. Copy statements of the form x : = y where the value of y is assigned to x.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as if x rel op y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation rel op to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.

6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. For example,

```
 param x1
param x2
.
 param xn call p,n
```

generated as part of a call of the procedure p(x1, x2, …. ,xn).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

9. Implementation of Three-Address Statements: A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are: Quadruples, Triples, Indirect triples.

A. Quadruples:

- A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result.

- The op field contains an internal code for the operator. The 3 address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result.

- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

- Fig a) shows quadruples for the assignment a : b * - c + b *- c

B. Triples:

1. To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

2. If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.

3. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).

4. Since three fields are used, this intermediate code format is known as triples.

	op	arg1	arg2	Result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Fig(a)

Mathematical Model :

Let S be the solution for the system.

S= {St , E , I , O , F, DD , NDD}

Where, St is an initial state such that St={Fl,Fy | Fl is lex file and Fy is a yacc file}. At initial state, system consists of two files, each with three sections.

E is a End state.

E={Sc, Fc | Sc = success case which generate intermediate three-address code. and Fc = failure case which mentions that unable to generate three-address code}.

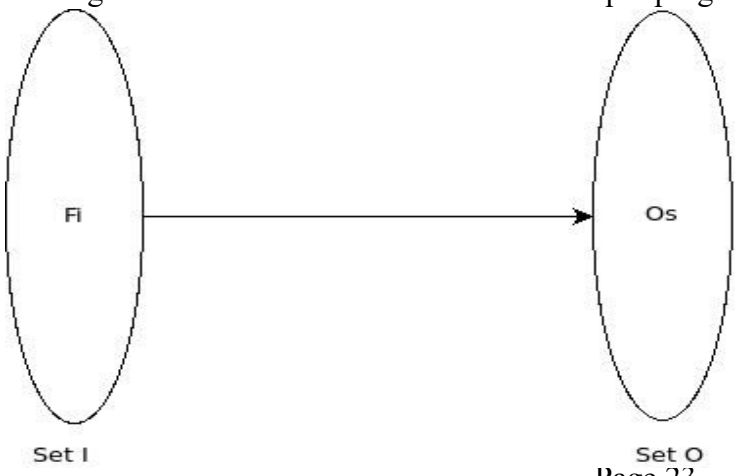
I= set of inputs

I={Fi | Fi is an input file which consists of High Level Language code}.

O={Os, Oc | Os is symbol table, Oc is console output which mention that three-address code of input program.}.

F=A set of Functions.

F={f1 | f1:I → O, function f1 generates the three-address code from input program}



Function f1: I → O

DD=Deterministic Data

DD={x | x is HLL's syntax which includes brackets, keywords, variables, functions definitions etc.}

NDD= Non-Deterministic Data

NDD={y | y is Semantic of the statements of Language}

ALGORITHM:

Write a LEX and YACC program to generate Intermediate Code for arithmetic expression

LEX program

1. Declaration of header files specially y.tab.h which contains declaration for Letter, Digit, expr.
2. End declaration section by %%
3. Match regular expression.
4. If match found then convert it into char and store it in yylval.p where p is pointer declared in YACC
5. Return token
6. If input contains new line character (\n) then return 0
7. If input contains „.“ then return yytext[0]
8. End rule-action section by %%
10. Declare main function
    a. open file given at command line
    b.if any error occurs then print error and exit
    c. assign file pointer fp to yyin
    d.call function yylex until file ends
11. End.

YACC program:

1. Declaration of header files
2. Declare structure for three address code representation having fields of argument1, argument2, operator, result.
3. Declare pointer of char type in union.
4. Declare token expr of type pointer p.
5. Give precedence to „*“,“/“.
6. Give precedence to „+“,“-“.
7. End of declaration section by %%.
8. If final expression evaluates then add it to the table of three address code.
9. If input type is expression of the form.
    a. exp“+“exp then add to table the argument1, argument2, operator.
    b. exp“-“exp then add to table the argument1, argument2, operator.
    c. exp“*“exp then add to table the argument1, argument2, operator.
    d. exp“/“exp then add to table the argument1, argument2, operator.
    e. „(„exp“)“ then assign $2 to $$.
    f. Digit OR Letter then assigns $1 to $$.

10. End the section by %%.
11. Declare file *yyin externally.
12. Declare main function and call yyparse function untill yyin ends
13. Declare yyerror for if any error occurs.
14. Declare char pointer s to print error.
15. Print error message.
16. End of the program.

Test Input:

```
main(){
        int x,y,z,i;
        x=5;
        i=6;
        y=8;
        z=1;
        if(x<=7){
                i=i+1+y*z;
                if(x==1){
                        y=3;
                        if(x<=2){
                                y=2;
                        }
                }
                else{
                                x=5;
                }
                }
        else{
                i=i-1-y*z;
        }
        while(x>0){
                        x=x-1;
        }
        if(x>=7){
                i=i+1+y*z;
        }
        else{
                i=i-1-y*z;
        }
        while(x>0){
                        x=x-1;
        }
}
```

Steps to execute the program:

$ lex filename.l (eg: comp.l)

$ yacc -d filename.y (eg: comp.y)
$gcc lex.yy.c y.tab.c
$./a .out

Output:

The three-address code:

0	=	5	x	
1	=	6	i	
2	=	8	y	
3	=	1	z	
4	<=	x	7	t0
5	IF	t0	0	20
6	+	i	1	t1
7	*	y	z	t2
8	+	t1	t2	t3
9	=	t3	i	
10	==	x	1	t4
11	IF	t4	0	17
12	=	3		y
13	<=	x	2	t5
14	IF	t5	0	16
15	=	2		y
16	GOTO			19
17	ELSE			
18	=	5		x
19	GOTO			25
20	ELSE			
21	-	i	1	t6
22	*	y	z	t7
23	-	t6	t7	t8
24	=	t8		i
25	>	x	0	t9
26	WHILE	t9	0	30
27	-	x	1	t10
28	=	t10		x
29	GOTO			26
30	>=	x	7	t11

Similarly …….

Symbol Table:

i int
z int
y int
x int

FAQ's

1. What are the different forms of ICG?

2. What are the difference between syntax tree and DAG?
3. What are advantages of 3-address code?
4. Which representation of 3-address code is better than other and why? Justify.
5. What is role of Intermediate code in compiler?

Practice Problem Statements:

1. Write a program to generate intermediate three-address code statements for 'for statement' construct in C language.
2. Write a program to generate intermediate three-address code statements for 'switch statement' construct in C language.
3. Write a program to implement the syntax directed definition for translating booleans to three addresses code.

Assignment - Compiler Construction 6

TITLE	**Code generation using Register Allocation**
PROBLEM STATEMENT /DEFINITION	Write a program for Register Allocation algorithm that translates the given code into one with a fixed number of registers.
OBJECTIVE	• To understand code generation phase of compilation. • To understand register allocation algorithm
S/W PACKAGES AND HARDWARE APPARATUS USED	Linux OS (Fedora 20), PC with the configuration as Core 2 Duo 2.2 GHz processor. 1 GB RAM, 160 G.B HDD, 17''Color Monitor, Keyboard, Mouse
REFERENCES	1. A V Aho, R. Sethi, .J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Education, ISBN 81 - 7758 - 590 2. K. Louden, "Compiler Construction: Principles and Practice", Thomson Brookes/Cole (ISE), 2003, ISBN 981 - 243 - 694-4 page no.31-90
STEPS	Refer theory, algorithm, test input, test output
INSTRUCTIONS FOR	* Title * Problem Definition

WRITING JOURNAL	* Theory and mathematical model * State transition diagram * Algorithm * Source Code * Output * Conclusion

Theory:
Code Generation

Goal: Takes Intermediate code representation of source program (optimized three address code statements) and produce equivalent target program as output.

Requirement of code genrator:

- Correct target code
- High quality code
- Effective use of resources of target machines
- Quick

Code generation is the last phase of compilation.

Simple Code Generator

- Simple code generation involves generating code for each three-address statements which are divided into different basic blocks. For that it makes use of registers to store operands and leaves result of computation in the register as long as possible.
- At the end of a basic block the results of the computation are stored only if the register is needed for another computation or just before procedure call, jump or labeled statements. Because after leaving this basic block flow enters into another different basic block.
- When generating code by above mentioned strategy we need :

1. Register Descriptor
2. Address Descriptor

1. Register descriptor

- To keep track of what is currently in each register, register descriptor is maintained.
- Register descriptor is pointer to a list that contains information about what is currently in each of the register. Initially all registers are empty.

2. Address descriptor

- To keep track of the locations where current value of the name can be found at run time, address descriptor is maintained. The locations can be a register, stack location or memory addresses.
- This particular information is stored in the symbol table.

Code Generation Algorithm

- Input to the code generation algorithm is a sequence of 3-address statement divided into blocks.
- For each three address statement of the form a := b op c in the basic block it performs following steps :
  1. Call getreg( ) to obtain the location L in which the computation a := b op c is performed. For that we need to pass three-address statement a := b op c as parameter to the getreg( ) function. Typically location L can be a register or it can be a memory location.
  2. Obtain the current location of the operand b from its address descriptor. If the value of y is in memory location and also in register then prefer register rather than memory location. If the value of b is not in location L then generate the instruction MOV b, L to store copy of b in L.
  3. The instruction OP x L is generated and the address descriptor of x is updated to indicate that x is now available in L. If L is register then update its descriptor to indicate that it will contain the run time value of x.
  4. If the current value of b and c are in the register then we can say that b and c are not further used. That means value of b and c are not live at the end of the basic block. So alter the register descriptor to indicate that after the execution of the three-address statement a : = b op c, those register will no longer contain b and / or c.

The Function getreg( )

- To perform computation specified by each of the three-address statement, we need a location. For that function getreg( ) is used.
- When a function getreg( ) is called, it returns a location for the computation performed by a three-address statement. For example if a := b op c is to be performed then getreg( ) returns a location L where the computation b op c should be performed ; and if possible it returns a register.
- It performs the following steps to return location L.
  1. First it searches for a register that already contains a value of b. If such register is available and if value of b has no further use after execution of a := b op c and if b is

not live at the end of basic block and hold the value of no other name then it returns that register for L.

2. Otherwise getreg() function searches for the empty register. If empty register is available then it returns that for L.
3. If empty register is not available and if a has further use in the block or op is an indexing operator that requires register then getreg ( ) finds a suitable occupied register. This occupied register is emptied by storing its value in proper memory location M and by updating address descriptor that register is returned to L. The least recently used strategy can be used to find suitable occupied register to be emptied.
4. If a has no further use in the block or no suitable occupied register can be found then getreg( ) selects a memory location and returns it for L.

Example

Consider the following expression x = (a + b) – ((c + d) – e)

The three-address code for this is generated as

$t_1 = a + b$

$t_2 = c + d$

$t_3 = t_2 – e$

$x = t_1 – t_3$

By applying above algorithm following code will be generated.

Table 1 shows the instruction generated and the contents of register descriptor and address descriptor :

Table 1

Statements	L	Instructions Generated	Register Descriptor	Address Descriptor
			All registers empty	
$t_1 = a + b$	R_o	MOV a, R_o ADD b, R_o	R_o will hold t_1	t_1 is in R_o
$t_2 = c + d$	R_1	MOV c, R_1 ADD d, R_1	R_1 will hold t_2	t_2 is in R_1
$t_3 = t_2 – e$	R_1	SUB e, R_1	R_1 will hold t_3	t_3 is in R_1
$x = t_1 – t_3$	R_o	SUB R_1, R_0	R_0 will hold x	x is in R_0
		MOV R_0, x		x is in R_0 and memory

Next-use information

- The next use information of each name is used to decide the register allocation.
- The next-use information is computed if :

1. A statement I assign a value to name x in a block.
2. A statement J uses x as an operand in the same block.
3. The path from the statement I to the statement J does not change the assignment to name x.

- For each three-address statement we have to compute the next use information for the names and this requires the backward scanning of the basic block.
- For each statement i of the form a := b op c perform following steps :
  1. The information about the next uses of a, b and c is attached to statement i.
  2. The information for a is set to no next-use.
  3. Set the information for b and c to be next-use in statement j.
- Consider the above three address statement :

$$t_1 = a + b$$
$$t_2 = c + d$$
$$t_3 = t_2 - e$$
$$x = t_1 - t_3$$

- When the above code generation algorithm is applied by considering only two registers R_0 and R_1 are available then the generated code is as shown in Table 2.

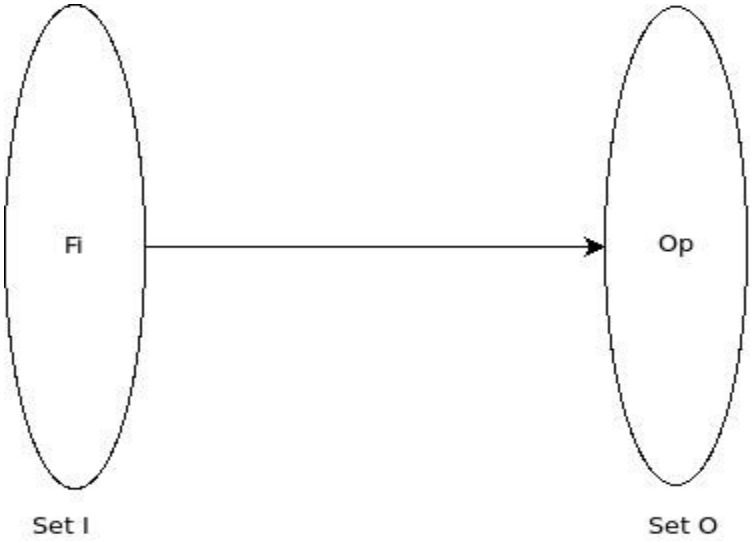
Table 2

Statements	L	Instructions Generated	Cost (in words)	Register Descriptor	Address Descriptor
				All registers empty	
$t_1 = a + b$	R_o	MOV a, R_o ADD b, R_o	2 2	R_o will hold t_1	t_1 is in R_o
$t_2 = c + d$	R_1	MOV c, R_1 ADD d, R_1	2 2	R_1 will hold t_2	t_2 is in R_1
$t_3 = t_2 - e$		MOV R_0 , t_1 Generated by getreg( )	2		t_1 is in R_o
	R_0	MOV e, R_0 SUB R_1, R_0	2 1	R_0 will hold t_3 R_1 will be empty because t_2 has no next use	t_3 is in R_o
$x = t_1 - t_3$	R_1	MOV t_1, R_1 SUB R_0, R_1	2 1	R_1 will hold x R_0 will be empty because t_3 has no next use	x is in R_1

Mathematical Model :

Let S be the solution for the system.

S= { St , E , I , O , F, DD , NDD}
Where,
St is an initial state such that St={Fl | Fl is IC}. At initial state, system has IC.
E is a End state.
E={Sc, Fc | Sc = success case which gives assembly code and Fc = failure case which gives the lexical errors}.
I= set of inputs
I={Fi | Fi is an input file which consists of IC}.
O= set of outputs
O={Op | Op is the assembly code generated}.
F=A set of Functions.
F={f | f1:I → O, function f implements simple code generator and generates the target code}



Function f: I → O

Assignment - Compiler Construction 7

Assignment - Compiler Construction 8

TITLE	Implement local and global code optimizations such as common sub expression elimination,copy propagation,dead code elimination,loop and basic block optimization(optional)

PROBLEM STATEMENT /DEFINITION	Write a Java program (using OOP features) to implement different code optimization techniques such as common sub expression elimination copy propagation,dead code elimination,loop and basic block optimization
OBJECTIVE	To learn and understand • Different techniques of code optimization • Implementation of code optimization techniques
S/W PACKAGES AND HARDWARE APPARATUS USED	C/C++ editors and compilers for Linux OS Linux OS/Fedora/Ubuntu PC with the configuration as Pentium IV 2.4 GHz. 4 GB RAM, 500 G.B HDD, 15''Color Monitor, Keyboard, Mouse
REFERENCES	"Compilers", Aho , Ulman, Sethi .
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	1. **Handwritten write-up as follows :** 2. Title 3. Objectives 4. Problem statement 5. Outcomes 6. Software and hardware requirements 7. Date of completion 8. Theory – Concept in brief 9. Algorithm 10. Flowchart 11. Design 12. Test cases

Aim: Implement local and global code optimizations such as common sub expression elimination,copy propagation,dead code elimination,loop and basic block optimization

Prerequisites:

Basics of code optimization

Learning Objectives:

To learn and understand

- Different techniques of code optimization
- Implementation of code optimization techniques

Learning Outcomes:

The student will be able to

1. Understand code optimization in detail
2. Implement local and global code optimization techniques

THEORY:

Code Optimization: Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (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:

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

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.

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.

Code optimization techniques

1.Common sub expression elimination

2.Copy propagation

3.Dead code elimination

4.Loop and basic block optimization

Common sub expression elimination

Common sub expression elimination is a compiler optimization technique of finding redundant expression evaluations, and replacing them with a single computation. This saves the time overhead resulted by evaluating the expression for more than once .

Example:

In the following code:

```
a = b * c + g;
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

Copy propagation

copy propagation is the process of replacing the occurrences of targets of direct assignments with their values.A direct assignment is an instruction of the form `x = y`, which simply assigns the value of `y` to `x`.

From the following code:

```
y = x
z = 3 + y
```

Copy propagation would yield:

```
z = 3 + x
```

Algorithm:

```
1)Generate the flow graph from list of instructions
do{

2)Perform reaching copy analysis

3)for each node in flow graph

 for each use

 if use is reached by the copy where it is the target

change the use to the source in the move statement tuple

While(changes)
4)Generate the list of instruction from modified flow graph
```

Dead code elimination

Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated
example
In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f ()
{
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

Below is the code fragment after dead code elimination.

```
int global;
void f ()
{
  global = 2;
  return;
}
```

Algorithm:

```
1)Generate the flow graph from list of instructions
do{
2)perform liveness on graph
3)for each node in flow graph
 ifthe define s set contains only one memory
   if the temporary being defined is not in the live out set
      Remove the node from the flow graph
While(changes)
4)Generate the list of instruction from modified flow graph
```

Loop and basic block optimization

loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops; as such, many compiler optimization techniques have been developed to make them faster.

Steps to do /algorithm:

1. Create s menu to select various code optimization techniques.
2. Take code as a input
3. Perform different optimization techniques on code
4. Analyse the time taken by code with and without optimization.

FAQs

- what is local and global code optimization?
- How different code optimization techniques works?

Oral/Review Questions:

5. What is code optimization in compiler?
6. How do you optimize code?
7. Why code optimization is called optional phase?
4. What are the different techniques of code optimization?