Assignment No : A5

1 Title:

Intermediate Code Generation.

2 Problem Definition:

Intermediate code generation for sample language using LEX and YACC.

3 Learning Objectives:

1. To understand Divide and Conquer strategy.
2. To use Divide and Conquer for implementing binary search.

4 S/W and H/W requirements:

1. Open source 64 bit OS.
2. Gedit text editor.

5 Theory

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.

5.1 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.

5.2 Three ways of intermediate representation:

1. Syntax tree
2. Postfix notation
3. 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.

5.3 Three-address code:

Each statements 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.

5.3.1 Advantages of three-address code:

1. Complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
2. The use of names for the intermediate values computed by a program allows three address code to be easily rearranged â unlike post-fix notation.
3. 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.

5.4 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.

5.5 Triples:

- 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.
- If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.
- 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).
- Since three fields are used, this intermediate code format is known as triples.

6 Related Mathematics

Let S be the solution perspective of the given problem.
The set S is defined as:
$S = \{\ s, e, X, Y, F, DD, NDD|_s\}$
Where,
s= Start point
e= End point
F= Set of main functions
DD= set of deterministic data
NDD= set of non deterministic data

X= Input Set.
X = source program code in high level language.

Y = $\{intermediatecodeforthesamplecode\}$

s = sample language ready into read buffer.
e = Intermediate code generated.

F = $\{f_{read}, f_{lex}, f_{parse}, f_{ret}\}$

f_{read} :function to read the sample code.

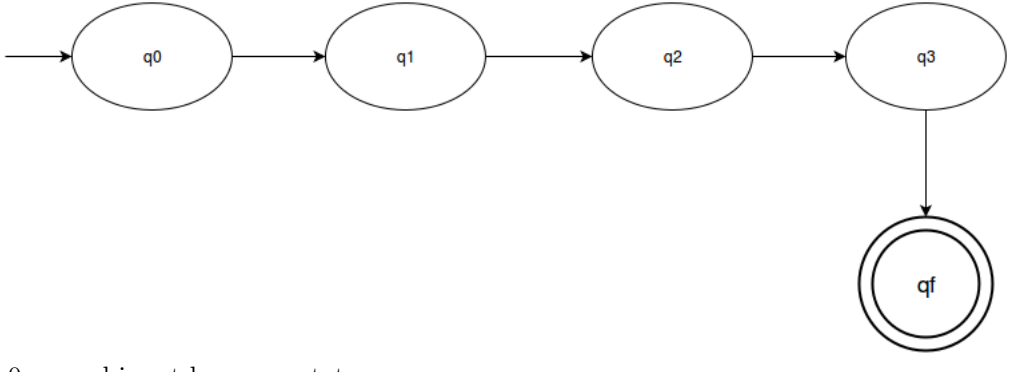
f_{lex} :function to generate tokens for the given codes.

f_{parse} :function to generate parse tree using the generated tokens.

f_{ret} :function generate intermediate code using the syntax tree.

DD = $\{samplecode, tokenmatchingrules, grammarrules\}$
NDD = ϕ

7 State Diagram



q0 = read input language state
q1 = token generation state
q2 = parsing state
q3 = display intermediate code state
qf = final state