COMPILER DESIGNING

Introduction :-

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called *machine language*. Since this is a tedious and error prone process most programming is, instead, done using a high-level *programming language*. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the *compiler* comes in.

A compiler translates (or *compiles*) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

* + Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
  + The compiler can spot some obvious programming mistakes.
  + Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well structured programs.

## The phases of a compiler: -

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-defined interfaces. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate module. Some of these modules are written by hand, while others may be generated from specifications. Often, some of the modules can be shared between several compilers.

A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

Lexical analysis This is the initial part of reading and analysing the program text: The text is read and divided into *tokens*, each of which corresponds to a symbol in the programming language, *e.g.*, a variable name, keyword or number.

Syntax analysis This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the *syntax tree*) that reflects the structure of the program. This phase is often called *parsing*.

Type checking This phase analyses the syntax tree to determine if the program violates certain consistency requirements, *e.g.*, if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable, such as trying to use a Boolean value as a function pointer.

Intermediate code generation The program is translated to a simple machine independent intermediate language.

Register allocation The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

## Lexical Analysis: -

## Introduction

The word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, words are objects like variable names, numbers, keywords *etc*. Such words are traditionally called *tokens*.

A *lexical analyser*, or *lexer* for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called *white-space*), *i.e.*, lay-out characters (spaces, newlines *etc.*) and comments.

The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

* Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non-linear factor involved which may make a separated system smaller than a combined system.
* Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
* Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

It is usually not terribly difficult to write a lexer by hand: You first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a

number, a variable or whatnot. However, this is not a very good way of handling the problem: You may read the same part of the input repeatedly while testing each possible token and in some cases it may not be clear where the next token ends. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by *lexer generators*, which transform human-readable specifications of tokens and white-space into efficient programs.

We will see the same general strategy in the chapter about syntax analysis: Specifications in a well-defined human-readable notation are transformed into efficient programs.

For lexical analysis, specifications are traditionally written using *regular expressions*: An algebraic notation for describing sets of strings. The generated lexers are in a class of extremely simple programs called *finite automata*.

This chapter will describe regular expressions and finite automata, their properties and how regular expressions can be converted to finite automata. Finally, we discuss some practical aspects of lexer generators.

LEX Description :-

LEX is a program generator designed for lexical processing of character input streams. It helps write programs whose control flow is directed by instances of regular expressions in the input stream, and is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

LEX source is a table of regular expressions and corresponding program fragments. The general format of LEX source is:

{definitions}

%%

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The rules represent the user's control decisions, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. The generated program is named yylex. Figure 5 is an overview of LEX.

Source ~ yylex

Input ~ Output

The recognition of the expressions is performed by a deterministic finite automaton generated by LEX. The program fragments written by the user are executed in the order

in which the corresponding regular expressions occur in the input stream. The lexical analysis programs written with LEX accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look ahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it. For the details of LEX source, regular expressions, actions, ambiguous source rules, source definition, usage, etc.

## Syntax Analysis: -

## Introduction

Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as *parsing*) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This “something” is typically a data structure called the *syntax tree* of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting *syntax errors*.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called *parser generation*.

The notation we use for human manipulation is *context-free grammars*, which is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can in some cases be translated almost directly into recursive programs, but it is often more convenient to generate *stack automata*. These are similar to the finite automata used for lexical analysis but they can additionally use a stack, which allows counting and non-local matching of symbols. We shall see two ways of generating such automata. The first of these, LL(1), is relatively simple, but works only for a somewhat restricted class of grammars. The SLR construction, which we present later, is more complex but accepts a wider class of grammars. Sadly, neither of these work for all context-free grammars. Tools that handle all context-free grammars exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars.

YACC Description: -

YACC provides a general tool for imposing structure on the input to a computer program. The YACC user prepares a specification of the input process, including rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine (the lexical

analyzer, LEX here} to do the basic input. The class of specification accepted is the LALR(l) grammars with disambiguating rules. The basic specification consists of three sections: the declarations, (grammar} rules, and programs.

A full specification file looks like

declarations

%%

rules

%%

programs

where the declarations and programs section may be empty. The rules section is made up of one or more grammar rules. With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. A rule has the form:

NT : BODY { ACTIONS }

NT represents a nonterminal name, and BODY represents a sequence of zero or more names and literals.

YACC then generates a function to control the input process. This function, called yyparse, is a parser which calls the lexical analyzer (LEX here} to pick up the basic items (tokens) from the input stream. These tokens are organized according to grammar rules. For the details of how the parser works, how it deals with ambiguity and conflicts, precedence, error handling, etc.

The Combination of LEX and YACC :-

LEX programs recognize only regular expressions; YACC writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of LEX and YACC is often appropriate. When used as a preprocessor for a later parser generator, LEX is used to partition the input stream, and the parser generator assigns structure to the resulting pieces.

lexical grammar

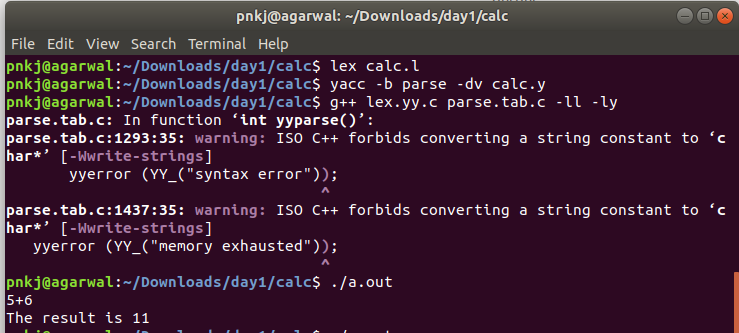
rules rules

Normally, the default main program on the LEX library calls yylex() routine. But if YACC is loaded, and its main program is used, YACC will call yylex(). IN this case each LEX rule should end with return(token): where the appropriate token values is returned. Supposing the YACC source file to be yfile and the LEX source file to be lfile the UNIX command sequence can be:

yacc yfile lex lfile

cc y.tab.c -ly -lS

The YACC library (-ly) should be loaded before the LEX library (-11) to obtain a main program which invokes the YACC parser. The generations of LEX and YACC programs can be done in either order.



PROJECT OBJECTIVE :-

The objective of this project is to build an compiler for an small language known as Little Quilt.

Little Quilt: -

Little Quilt is a very small language for designing quilts. It is based on ML syntax. The language has two types of quilt designs denoted by ‘a’ and ‘b’.

There are two major operations in Little Quilt, Turn and Sew.

Turn: - The work of this function is to turn the Quilt by 90 degrees clockwise. The design can either be ‘a’ or ‘b’.

The turn function has only one parameter.

Syntax: -Turn(expr)

Sew: -The work of this function is to switch two quilts designs sideways.

The Sew function has two parameters.

Syntax: - Sew (expr1, expr2)

Implementation Details: -

Assignment A0:

The task in this assignment is to scan and parse a valid quilt expression. The expression is scanned through command line and only the valid tokens were accepted and a corresponding value for each token is returned to the yacc file (which is being later used for parsing). The scanning is done through an LEX (lexical Analysis Generator) file.

In the Lex file, all the valid token is being specified that are to be accepted. The extension of Lex file is ‘.l’ and the command to compile Lex file is:

lex filename.l

In the yacc file, the valid tokens that were accepted by Lex file are received. Now in the yacc file the parsing in done and it is the part where the grammar is written and it is checked whether the expression is valid or not. The extension of YACC (Yet Another Compiler Compile) file is ’.y’ and the command to compile yacc file is:

yacc –b parse –dv filename.y

Assignment A1:

Our task in this assignment is to extend the assignment A0 and create an AST (Abstract Syntax Tree) of it.

In computer science, an **abstract syntax tree** (**AST**), or just **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

Assignment A2:

The task in the assignment that is to be performed is to print the quilt that is being generated in the matrix form.

Assignment A3:

The task in this assignment is to compile the AST produced in the previous assignment and generate c++ code to compute the resulting quilt in the matrix form.



If you have to include same header files more than one in different files then we should use

#ifndef name

#define name

…

…

…

…

…

…

...

#endif

In some cases when we used above code the header files are not included even once so to overcome this we can do the prototype declaration of that class that we have wanted to use.

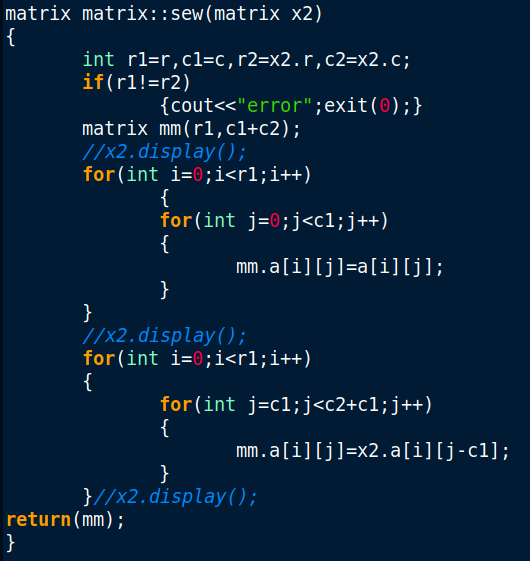
As you can see here, the value of matrix x2 is getting updated in the first loop where the matrix is not even referenced.

This error occurred because we were declaring the same pointer again.

Segmentation Fault :

The segmentation fault is an common condition that causes program to crash. They are often associated with a file named core. Segmentation faults are caused by a program trying to read or write an illegal memory location.

It occurs when a reference of an variable fall outside the segment where that variable resides,or when an write is attempted to a location that is in a read-only segment. In practice,segfaults are almost always due to trying to read or write a non-existent array element, not properly defining a pointer before using it, or accidentally using a variable’s value as an address.



class matrix{

public : int r,c,\*\*a; *//already declared int \*\*a here*

matrix(int r,int c){

this->r = r;

this->c = c;

*//int \*\*a = new int\* [r]; ← wrong statement*

a = new int\* [r];

for(int i=0;i<r;i++)

a[i]=new int [c];

}

};