LEX

AIM:

To have a detailed analysis of LEX.

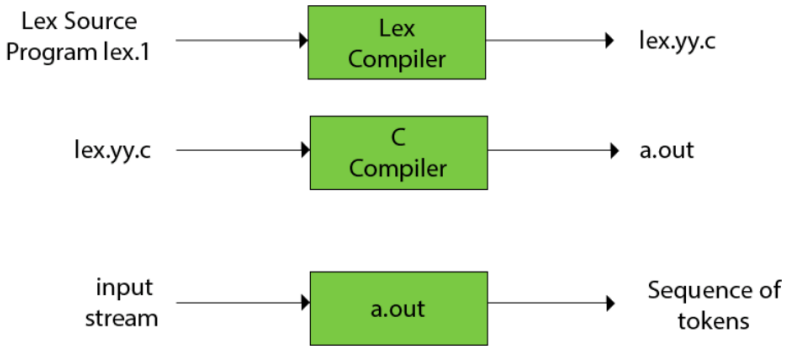
PURPOSE OF THE TOOL:

Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers.

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

WORKING:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



LEX FILE FORMAT

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%

5. { user subroutines }

- Definitions include declarations of constant, variable and regular definitions.
- Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.
- Where pi describes the regular expression and action1 describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
- User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

ADVANTAGES:

- Lex gives you the ability to create lexical analyzers based on patterns. These patterns are regular expression used to define the different tokens to be recognized in the character input sequence. The lexer generates internal tables from all token definitions – the so called deterministic finite automata (DFA's)

DISADVANTAGES:

- Regular expressions are translated by lex to a computer program that mimics an FSA.However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures.
- often produces C programs that are longer and execute more slowly than hand-coded programs that do the same task.

APPLICATIONS:

Using Lex with parser generators

- Lex and parser generators, such as Yacc or Bison, are commonly used together. Parser generators use a formal grammar to parse an input stream, something which Lex cannot do using simple regular expressions, as Lex is limited to simple finite state automata.[clarification needed]
- It is typically preferable to have a parser, one generated by Yacc for instance, accept a stream of tokens (a "token-stream") as input, rather than having to process a stream of characters (a "character-stream") directly. Lex is often used to produce such a token-stream.
- Scannerless parsing refers to parsing the input character-stream directly, without a distinct lexer.

LEX AND MAKE

- make is a utility that can be used to maintain programs involving Lex. Make assumes that a file that has an extension of .l is a Lex source file.
- The make internal macro LFLAGS can be used to specify Lex options to be invoked automatically by make.

FUTURE SCOPE :

Amazon Lex is an AWS solution that allows developers to publish voice or chat bots for use across different mobile, web and chat platforms. It can listen, understand user intent, and respond to context. Powered by deep learning functionalities like automatic speech recognition (ASR) and natural language processing (NLU), Lex is also the technology behind Alexa devices. Available now in the open, it can be easily leveraged by enterprises to build their own digital assistants.

Amazon Lex for Enterprises

For enterprises, Lex-powered applications can become a key competitive advantage, allowing them to optimize processes and enable cost savings. A few key aspects where Amazon Lex can assist are:

Performing User-based Applications

Lex can help build bots capable of providing information, or addressing user requests and queries. It can perform applications like ordering food, booking tickets, and accessing bank account.

Made possible with the help of the ARS and NLU, these capabilities can help create powerful interfaces customer-facing mobile applications. Such a voice or text chat interface on mobile devices can help users perform tasks that involve a series of steps played out in a conversational format. Further, the integration of Lex with Amazon Cognito helps developers control user management, authentication, and sync across all devices.

For example, healthcare enterprises can enable patients to schedule appointments at their facility with Lex powered bots. The patient can send a text request via his mobile application for “an appointment on Monday”.

- Amazon Lex will recognize that an appointment has been requested, and will ask the user for a “preferred time on Monday”.
- The user responds with a text, say, “1 pm”.
- Lex will reserve this appointment time for the user once the account information is retrieved.
- It will further notify the patient that “an appointment time of 1 pm has been finalised on Monday”.

Similarly, tasks like opening bank accounts, ordering food, or finding the right dress at a retail store can all be accomplished via Lex-powered bots.

FLEX

AIM:

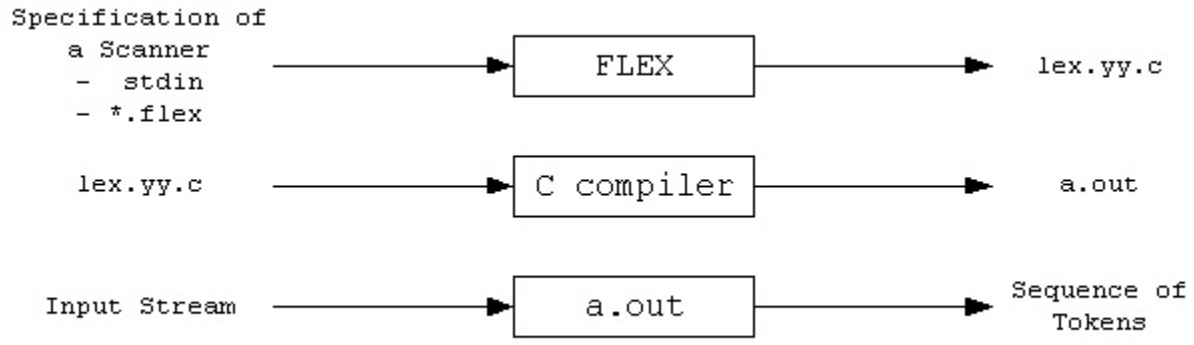
To have a detailed analysis of FLEX.

PURPOSE OF THE TOOL:

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
Bison produces parser from the input file provided by the user. The function yylex() is automatically generated by the flex when it is provided with a .l file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

WORKING:

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. In stead 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. (sample1.lex, sample2.lex)
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner!

ADVANTAGES:

- Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex.
- It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers").
- Run time: Flex also provides faster run time compared to lex. The run time is about two times faster.
- Table compression: The table created by flex is approximately 17 times smaller than that created by lex.
- Compile time: The time it takes to compile the lexer using flex is roughly 3 times faster than the time it takes to compile the lexer using lex.

DISADVANTAGES:

Time complexity

A Flex lexical analyzer usually has time complexity in the length of the input. That is, it performs a constant number of operations for each input symbol. This constant is quite low: GCC generates 12 instructions for the DFA match loop. Note that the constant is independent of the length of the token, the length of the regular expression and the size of the DFA.

However, using the REJECT macro in a scanner with the potential to match extremely long tokens can cause Flex to generate a scanner with non-linear performance. This feature is optional. In this case, the programmer has explicitly told Flex to "go back and try again" after it has already matched some input. This will cause the DFA to backtrack to find other accept states. The REJECT feature is not enabled by default, and because of its performance implications its use is discouraged in the Flex manual.

Reentrancy

By default the scanner generated by Flex is not reentrant. This can cause serious problems for programs that use the generated scanner from different threads. To overcome this issue there are options that Flex provides in order to achieve reentrancy. A detailed description of these options can be found in the Flex manual.

Usage under non-Unix environments

Normally the generated scanner contains references to unistd.h header file which is Unix specific. To avoid generating code that includes unistd.h, %option nounistd should be used. Another issue is the call to isatty (a Unix library function), which can be found in the generated code. The %option never-interactive forces flex to generate code that doesn't use isatty.

Using flex from other languages

Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool such as SWIG can be used.

APPLICATIONS:

One of possible techniques to enable access to internals of Flash applications is including the TestComplete FlexClient helper library in the application to be tested. The topics of this section describe how to compile your Flash or Flex application with the FlexClient library included using different IDEs

JFlex is a lexical analyzer generator (also known as scanner generator) for Java, written in Java.

A lexical analyzer generator takes as input a specification with a set of regular expressions and corresponding actions. It generates a program (a lexer) that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched. Lexers usually are the first front-end step in compilers, matching keywords, comments, operators, etc, and generating an input token stream for parsers. Lexers can also be used for many other purposes.

JFlex lexers are based on deterministic finite automata (DFAs). They are fast, without expensive backtracking.

JFlex is designed to work together with the LALR parser generator CUP by Scott Hudson, and the Java modification of Berkeley Yacc BYacc/J by Bob Jamison. It can also be used together with other parser generators like ANTLR or as a standalone tool.

FUTURE SCOPE:

flex++ is a similar lexical scanner for C++ which is included as part of the flex package. The generated code does not depend on any runtime or external library except for a memory allocator (malloc or a user-supplied alternative) unless the input also depends on it. This can be useful in embedded and similar situations where traditional operating system or C runtime facilities may not be available.

The flex++ generated C++ scanner includes the header file FlexLexer.h, which defines the interfaces of the two C++ generated classes.

YACC:

AIM

To have a detailed analysis of YACC.

PURPOSE OF THE TOOL:

- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

WORKING:

The input to yacc describes the rules of a grammar. yacc uses these rules to produce the source code for a program that parses the grammar. You can then compile this source code to obtain a program that reads input, parses it according to the grammar, and takes action based on the result.

The source code produced by yacc is written in the C programming language. It consists of a number of data tables that represent the grammar, plus a C function named yyparse(). By default, yacc symbol names used begin with yy. This is an historical convention, dating back to yacc's predecessor, UNIX yacc. You can avoid conflicts with yacc names by avoiding symbols that start with yy.

If you want to use a different prefix, indicate this with a line of the form:

```
%prefix prefix
```

at the beginning of the yacc input. For example:

```
%prefix ww
```

asks for a prefix of ww instead of yy. Alternatively, you could specify -p ww on the lex command line. The prefix chosen should be 1 or 2 characters long; longer prefixes lead to name conflicts on systems that truncate external names to 6 characters during the loading process. In addition, at least 1 of the

characters in the prefix should be a lowercase letter (because yacc uses an all-uppercase version of the prefix for some special names, and this has to be different from the specified prefix).

ADVANTAGES:

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; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

DISADVANTAGES:

Yacc is inflexible in some ways: good error handling is hard (basically, its algorithm is only defined to parse a correct string correctly, otherwise, all bets are off; this is one of the reasons that GCC moved to a hand-written parser)

APPLICATIONS:

- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.

FUTURE SCOPE

Yacc itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted by more recent, largely compatible, programs such as Berkeley Yacc, GNU Bison, MKS Yacc, and Abraxas PCYACC. An updated version of the original AT&T Yacc is included as part of Sun's OpenSolaris project. Each offers slight improvements and additional features over the original Yacc, but the concept and basic syntax have remained the same

BISON:

AIM

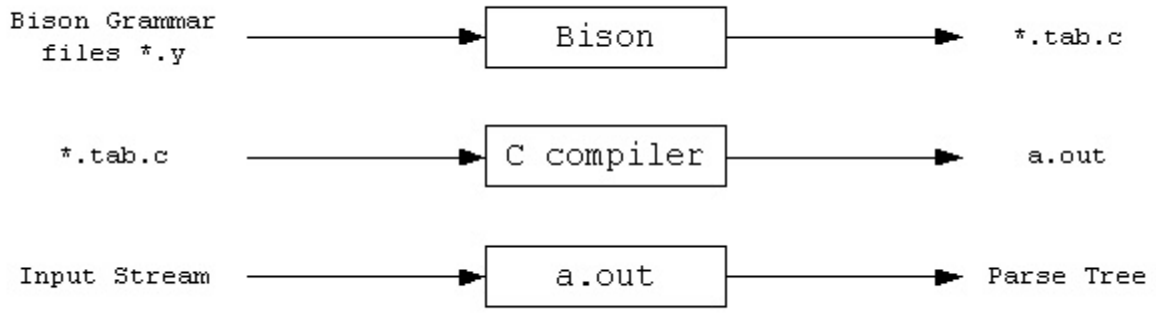
To have a detailed analysis of BISON.

PURPOSE OF THE TOOL:

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR (1) parser tables. As an experimental feature, Bison can also generate IELR (1) or canonical LR(1) parser tables.

WORKING:

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:

  Write a lexical analyzer to process input and pass tokens to the parser (calc.lex).

- Write the grammar specification for bison (calc.y), including grammar rules, yyparse() and yyerror().
- Run Bison on the grammar to produce the parser. (Makefile)
- Compile the code output by Bison, as well as any other source files.
- Link the object files to produce the finished product.

ADVANTAGES:

There are some differences between Lex and Flex, but you have to be abusing Lex to run into the problems with Flex. (I have a program which abuses Lex and doesn't work under Flex, therefore.) This is primarily in the area of input lookahead; in Lex, you can provide your own input code and modify the character stream; Flex won't let you do that.

Yacc and Bison are pretty closely compatible, though Bison has some extra tricks it can do.

You probably can't find legitimate copies of (the original, AT&T versions of) Lex and Yacc to install on Ubuntu. I wouldn't necessarily say it is impossible, but I'm not aware of such. Flex and Bison are readily available and are equivalent for most purposes. You may also find various alternative and approximately equivalent programs from the BSD world.

Lex and Yacc are maintained by the Unix SVRx licencees - companies such as IBM (AIX), HP (HP-UX) and Sun (Solaris) have modified versions of Lex and Yacc at their command. MKS also provides MKS Lex and MKS Yacc; however, the Yacc at least has some non-standard extensions.

Flex and Bison are free. (AT&T) Lex and Yacc are not.

DISADVANTAGES:

- flex and Bison have smaller and fractured communities, but they have good documentation
- flex and Bison are stable and maintained software but there is no active development. C++ support can be of limited quality.
- flex and Bison maintain an old-school design with little support for readability or productivity.
- Bison only supports BNF, which makes grammars more complicated

APPLICATIONS:

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR parser tables. As an experimental feature, Bison can also generate IELR or canonical LR parser tables.

FUTURE SCOPE

Parser generators are nice, but they aren't very user friendly. You typically can't give good error messages, nor can you provide error recovery. Perhaps your language is very weird and parsers reject your grammar or you need more control than the generator gives you.

Hand-written parsers also usually perform better than generated ones, assuming the quality of the parser is high enough. On the other hand, if you don't manage to write a good parser - usually due to lack of experience, knowledge or design-then performance is usually slower. For lexers the opposite is true though: generally generated lexers use table lookups, making them faster than hand-written ones.

Education-wise, writing your own parser will teach you more than using a generator. You have to write more and more complicated code after all, plus you have to understand exactly how you parse a language. On the other hand, if you want to learn how to create your own language either option 1 or option 3 is preferable: if you're developing a language, it will probably change a lot, and option 1 and 3 give you an easier time with that.