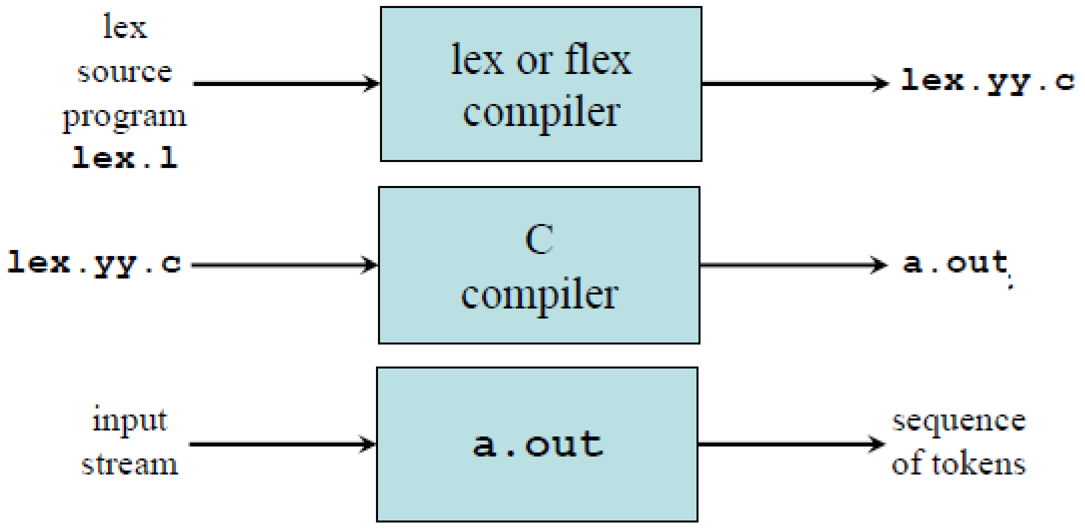
Flex: Language for Lexical Analyzer

Systematically translate regular definitions into C source code for efficient scanning.
Generated code is easy to integrate in C applications



Flex: An introduction

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, ‘lex.yy.c’ by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Flex specification:

A *flex specification* consists of three parts:
Regular definitions, C declarations in **%{ %}**
%%
Translation rules
%%
User-defined auxiliary procedures

The *translation rules* are of the form:

$p1$ {action1}
$p2$ {action2}
……………………..
p_n { $action_n$ }

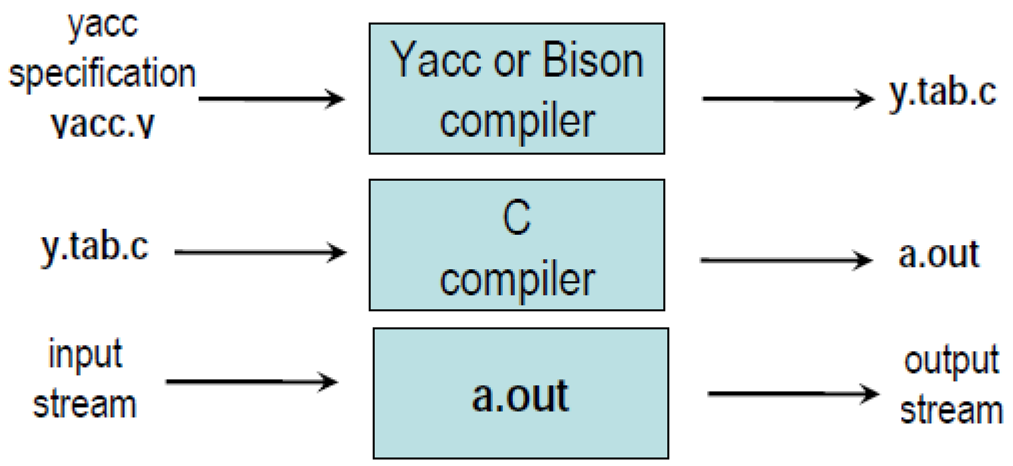
In all parts of the specification comments of the form /* **comment text** */ are permitted.

Parser Generators:

Introduction to Bison

Bison is a general purpose parser generator that converts a description for an *LALR(1) context-free grammar* into a C program file.

- ✓ The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions.
- ✓ The tokens come from a function called the lexical analyzer that must supply in some fashion (such as by writing it in C).
- ✓ The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens.
- ✓ Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this.
- ✓ The Bison parser file is C code which defines a function named *yyparse* which implements that grammar. This function does not make a complete C program: you must supply some additional functions.



Stages in Writing Bison program

1. Formally specify the grammar in a form recognized by Bison
2. Write a lexical analyzer to process input and pass tokens to the parser.
3. Write a controlling function that calls the Bison produced parser.
4. Write error-reporting routines.

Bison Specification

• A *bison specification* consists of four parts:

```
%{
        C declarations
%}
Bison declarations
%%
        Grammar rules
%%
Additional C codes
```

Productions in Bison are of the form

```
Non-terminal:   tokens/non-terminals {action}
                | Tokens/non | terminals {action}
                ……………………………………..
                ;
```

Bison Declaration

Tokens that are single characters can be used directly within productions, e.g. **‘+’, ‘-’**, **‘*’**

Named tokens must be declared first in the declaration part using

%token *Token Name (Upper Case Letter)*

```
        e.g     %token INTEGER IDENTIFIER
                %token NUM 100
```

– *%left, %right* or *%nonassoc* can be used instead for *%token* to specify the precedence & associativity (precedence declaration). All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity.

– *%union* declares the collection data types

– *%type <non-terminal>* declares the type of semantic values of non-terminal

– *%start <non-terminal>* specifies the grammar start symbol (by default the start symbol of grammar)