# CSC 365: Lab Assignment #2

Anup Adhikari
tutor@anup.pro.np

Compiler Design and Construction July 24, 2022

# 1 BISON

## 1.1 Introduction

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C programming in order to use Bison or to understand this manual.

## 1.2 Who uses bison?

Bison grammars can be used only in programs that are free software. This is in contrast to what happens with the GNU C compiler and the other GNU programming tools.

The reason Bison is special is that the output of the Bison utility–the Bison parser file–contains a verbatim copy of a sizable piece of Bison, which is the code for the yyparse function. (The actions from your grammar are inserted into this function at one point, but the rest of the function is not changed.)

As a result, the Bison parser file is covered by the same copying conditions that cover Bison itself and the rest of the GNU system: any program containing it has to be distributed under the standard GNU copying conditions.

## 1.3 Bison Concepts

### 1.3.1 Languages and Context-free grammar

In order for Bison to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an ‘expression’. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is Backus-Naur Form or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1).

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a symbol. Those which are built by grouping smaller constructs according to grammatical rules are called nonterminal symbols; those which can’t be subdivided are called terminal symbols or token types. We call a piece of input corresponding to a single terminal symbol a token, and a piece corresponding to a single nonterminal symbol a grouping.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators

and punctuation marks. So the terminal symbols of a grammar for C include ‘identifier’, ‘number’, ‘string’, plus one symbol for each keyword, operator or punctuation mark: ‘if’, ‘return’, ‘const’, ‘static’, ‘int’, ‘char’, ‘plus-sign’, ‘open-brace’, ‘close-brace’, ‘comma’ and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

tokens.c

```
int             /* keyword 'int' */
square (x)      /* identifier, open-paren, */
                /* identifier, close-paren */
    int x;      /* keyword 'int', identifier, semicolon */
{               /* open-brace */
  return x * x; /* keyword 'return', identifier, */
                /* asterisk, identifier, semicolon */
}               /* close-brace */
```

### 1.3.2 Bison Grammar Outline

bisonGrammar.y

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

#### 1.3.2.1 The C Declarations Section

The C declarations section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of yyparse. You can use ‘#include’ to get the declarations from a header file. If you don’t need any C declarations, you may omit the ‘%’ and ‘%’ delimiters that bracket this section.

#### 1.3.2.2 The Bison Declarations Section

The Bison declarations section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. In some simple grammars you may not need any declarations. See section Bison Declarations.

#### 1.3.2.3 The Grammar Rules Section

The grammar rules section contains one or more Bison grammar rules, and nothing else. See section Syntax of Grammar Rules.

There must always be at least one grammar rule, and the first ‘%%’ (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

#### 1.3.2.4 The Additional C Code Section

The additional C code section is copied verbatim to the end of the parser file, just as the C declarations section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of yyparse. For example, the definitions of yylex and yyerror often go here.

If the last section is empty, you may omit the '%%' that separates it from the grammar rules.

The Bison parser itself contains many static variables whose names start with 'yy' and many macros whose names start with 'YY'. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

### 1.3.3 Invoking BISON file

Command Line

```
currentdir> bison infile
```

**Info:** Here infile is the grammar file name, which usually ends in '.y'. The parser file's name is made by replacing the '.y' with '.tab.c'. Thus, the 'bison foo.y' filename yields 'foo.tab.c', and the 'bison hack/foo.y' filename yields 'hack/foo.tab.c'.

1. The lexer file (ending in .l)

lexer.l

```
%{
/* definitions */
#include "parser.tab.h"
%}

/* rules */
%%
[0-9]+ { yylval.num=atoi(yytext); return NUMBER; }
"+" {return PLUS;}
\n {return EOL;}
. {}

%%

yywrap(){}
```

Command Line

```
currentdir> flex lexer.l
```

2. The parser file (parser.y)

parser.y

```
%{
  /* definition */
  #include<stdio.h>
  void yyerror(const char* msg) {
      fprintf(stderr, "%s\n", msg);
   }
   int yylex();
%}
/* Union tells that the input can be number or character_symbol */
%union{
  int num;
  char sym;
}

%token EOL
%token<num> NUMBER
%token PLUS
%type<num> exp

/* rules are here */
%%
input:
| line input
;

line:
  exp EOL {printf("%d\n", $1);}
| EOL
;

exp:
  NUMBER {$$=$1;}
| exp PLUS exp {$$ = $1 + $3 ;}
;

%%
int main(){
  yyparse();
}
```

Command Line

```
currentdir> bison -d -t parser.y
```

3. Finally compile the parser using lexer

Command Line

```
currentdir> gcc lex.yy.c parser.tab.c
```

Let's look into simple examples.

ex1.l

```
%{
    #include "ex1.tab.h"
%}

%%

[0-9]   {return (yytext[0]);}
[+*()]  {return (yytext[0]);}
\n      {return (0);}
.       {}

%%
```

ex1.y

```
%{
  #include <stdio.h>
 void yyerror(const char* msg) {
    fprintf(stderr, "%s\n", msg);
 }
 int yylex();
%}

%%

expr : expr '+' term {printf(" + ");}
     | term
     ;

term : term '*' fact {printf(" * ");}
     | fact
     ;

fact : "(" expr ")"
     | '0' {printf("0");}
     | '1' {printf("1");}
     | '2' {printf("2");}
     | '3' {printf("3");}
     | '4' {printf("4");}
     | '5' {printf("5");}
     | '6' {printf("6");}
     | '7' {printf("7");}
     | '8' {printf("8");}
     | '9' {printf("9");}

%%
/* int main(){
  yyparse();
} */
```

**Info:** Remember! GCC may require some libraries to compile. provide extra options with -l
Examples:
For math use -lm
For libl.a use -ll
For libfl.a use -lfl

Command Line

```
gcc hello.c -lm
```