The first course in formal language theory.

BISON

(*"If we do not milk the cow fully, it falls sick" - Lalu Prasad Yadav*)

lection №11

Leonov Alexander Georgievich

lead researcher MSU, Faculty Mechanics and Mathematics,

SRISA RAS, APE

head of the department

dr.l@math.msu.su

BISON

YACC - (Yet Another Compiler Compiler) program for a generator of parsers develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

The YACC was originally developed by Stephen C. Johnson at AT&T for the UNIX operating system. Later, analogs of the program were written, such as Berkeley YACC, GNU BISON. BISON was written originally by Robert Corbett. Richard Stallman made it YACC-compatible. Since a parser generated with YACC can use a lexical analyzer, it is FLEX(LEX) in most cases. BISON is upward compatible with YACC.

In order for BISON to parse a language, it must be described by a context-free grammar(CFG). There are various important subclasses of CFG. Although it can handle almost all context-free grammars, BISON is optimized for what are called LR(1) grammars. In brief, in these grammars, it must be possible to tell how to parse any portion of an input string with just a single token of lookahead. For historical reasons, BISON by default is limited by the additional restrictions of LALR(1), which is hard to explain simply. Parsers for LR(1) grammars are deterministic. We'll concentrate on the BISON.

Format of the input file

The BISON input file consists of three sections+"prologue", separated by a line with just *%%* in it:

%{

prologue

%}

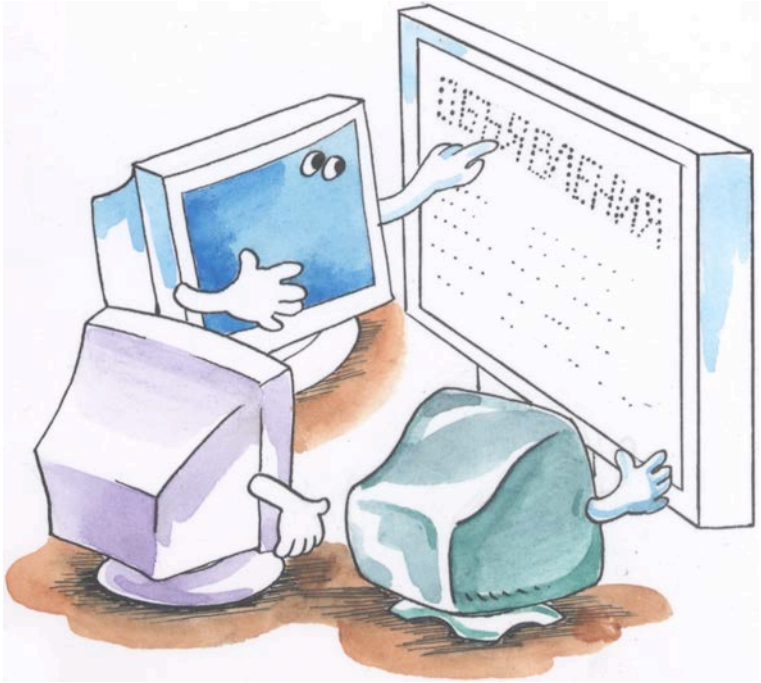
declarations

%%

rules

%%

user code



The *%%*, *%{* and *%}* are punctuation that appears in every BISON grammar file to separate the sections. The prologue may define types and variables used in the actions *(#include* to include header files) that do any of these things. we need to declare the lexical analyzer *yylex* along with any other global identifiers used by the actions in the grammar rules. The BISON declarations declare the names of the terminal and nonterminal symbols.

Format of the input file (2)

The BISON may also describe operator precedence and the data types of semantic values of various symbols. The grammar rules define how to construct each nonterminal symbol from its parts.

The user code can contain any code we want to use. Often the definitions of functions declared in the prologue go here. In a simple program, all the rest of the program can go here.

The description block may be empty and the user code block is also. The minimum BISON specification is:

rules

%%

We give it a BISON grammar file as input. The output is a C source file that implements a parser for the language described by the grammar. This parser is called a *BISON parser*, and this file is called a *BISON parser implementation* file.

The BISON parser is to group tokens into groupings according to the grammar rules (to build identifiers and operators into expressions). As it does this, it runs the actions for the grammar rules it uses.

The prologue

The Prologue 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*. We can use *#include* to get the declarations from a header file. If we don't need any C declarations, we may omit the *%{* and *%}* delimiters that bracket this section. The Prologue section is terminated by the the first occurrence of *%}* that is outside a comment, a string literal, or a character constant.

We may have more than one Prologue section, intermixed with the BISON declarations.

Example.

```
%{
  #include <stdio.h>
  int yylex (void);
%}
```

BISON Declarations.

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

All token type names (but not single-character literal tokens such as *+* and * must be declared. Nonterminal symbols must be declared if we need to specify which data type to use for the semantic value.

The basic way to declare a token type name (terminal symbol) is as follows:

%token name

BISON will convert this into a *#define* directive in the parser, so that the function *yylex* (if it is in this file) can use the name name to stand for this token type's code. we can explicitly specify the numeric code for a token type by appending a decimal or hexadecimal integer value in the field immediately following the token name:

%token NUM 1

It is generally best to let BISON choose the numeric codes for all token types.

Operator Precedence

Use the *%left, %right and %precedence*, declaration to declare a token and specify its precedence and associativity, all at once.

%left symbols...

The associativity of an operator op determines how repeated uses of the operator nest: whether *x op y op z* is parsed by grouping *x* with *y* first or by grouping *y* with *z* first. *%left* specifies left-associativity (grouping *x* with *y* first) and *%right* specifies right-associativity (grouping *y* with *z* first). The Last alternative, *% precedence*, allows to define only precedence and no associativity at all.

The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first. *Example:*

%left '<'

%left '-'

Explanation of input

Consider the definition of input:

input: %empty | input line;

This definition reads as follows: “A complete input is either an empty string, or a complete input followed by an input line”. Notice that “complete input” is defined in terms of itself. This definition is said to be left recursive since input appears always as the leftmost symbol in the sequence. The first alternative is empty because there are no symbols between the colon and the first ‘|’; this means that input can match an empty string of input (no tokens).

The second alternate rule (input line) handles all nontrivial input. It means, “After reading any number of lines, read one more line if possible.” The left recursion makes this rule into a loop. Since the first alternative matches empty input, the loop can be executed zero or more times.

The parser function *yyparse* continues to process input until a grammatical error is seen or the lexical analyzer says there are no more input tokens; we will arrange for the latter to happen at end-of-input.

Explanation of line. Function. Variable. Value. Directive.

Consider the definition of input:

```
line:   '\n' | S '\n'     { printf ("%d\n", $1); };
```

The first alternative is a token which is a newline character; this means that the parser accepts a blank line (and ignores it, since there is no action). The second alternative is an expression followed by a newline. The semantic value of the exp grouping is the value of *$1*. The action prints this value, which is the result of the computation the user asked:

- Function: *yylex* - user-supplied lexical analyzer function, called with no arguments to get the next token.
- Function: *yyparse* - the parser function produced by BISON; call this function to start parsing.
- Variable: *yylval* - external variable in which *yylex* should place the semantic value associated with a token. (In a pure parser, it is a local variable within *yyparse*, and its address is passed to *yylex*.)
- Value: *YYEOF* - the token kind denoting is the end of the input stream.
- Directive: *%start* - BISON declaration to specify the start symbol.

Grammar rules. Example.

A block of rules consists of one or more grammar rules, of the form:

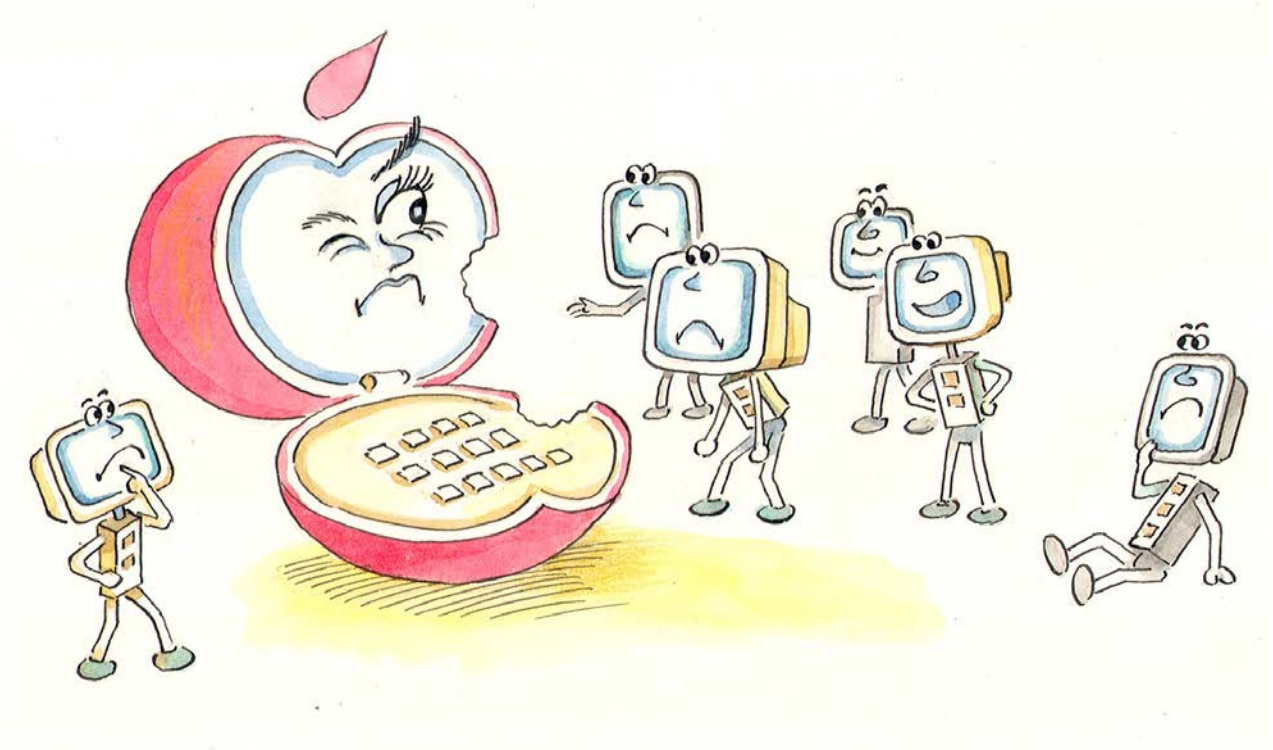
<name>: <body>;

<name> is a nonterminal. <body> is a sequence of zero or more names and literals (characters enclosed in single quotes).

If there are multiple grammar rules with the same left side, the pipe | can be used.

Example.

```
%token name
%start e
%%
e : e '+' m | e '-' m | m ;
m : m '*' t | m '/' t | t ;
t : name | '(' e ')' ;
%%
```



Semantic actions

Each grammar rule can be assigned actions that are performed each time the rule is recognized in the input. These actions can return values and receive values returned by previous actions. Moreover, the lexical analyzer can return token values if needed.

The actions are the C code that appears inside braces.

The groupings of the parser defined in this section are the expression (given the name exp). Each of these nonterminal symbols has several alternate rules, joined by the vertical bar '|' which is read as "or". The semantics of the language is determined by the actions taken when a grouping is recognized.

Example:

```
exp:
NUM
| exp '+' exp { $$ = $1 + $3; }
| exp '-' exp { $$ = $1 - $3; }
;
```

Semantic actions (2)

In each action, the pseudo-variable $$ stands for the semantic value for the grouping that the rule is going

to construct (the left character of the rule). Assigning a value to $$ is the main job of most actions. The semantic values of the components of the rule are referred to as $1, $2, and so on.

Note. In addition to the usual stack, the parser built by BISON contains a semantic stack containing these character values. Values are of type *YYSTYPE*, which is defined as *integer* by default and can be reassigned:

%define api.value.type {double}.

Because we specify *{double}*, each token and each expression has an associated value, which is a floating point number. C code can use *YYSTYPE* to refer to the value *api.value.type*.

Action can be used in the formula interpreter, in which the value of the nonterminal "expression" is its calculated value. If no action is specified for the rule, or the action does not contain a mention of the *$$* pseudo-variable, then the value of the left side of the rule becomes equal to the value of the first character of the right side, i.e. the action is implicitly performed *{ $$ = $ 1; }*. The value of the next token is copied from the *yylval* variable, into which the scanner usually sets it.

Context-Dependent Precedence

A minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator. The BISON precedence declarations, *%left*, *%right*, can only be used once for a given token; so a token has only one precedence declared in this way. For context dependent precedence, we need to use an additional mechanism: the *%prec* modifier for rules.

The *%prec* modifier declares the precedence of a particular rule by specifying a terminal symbol whose precedence should be used for that rule. It's not necessary for that symbol to appear otherwise in the rule. The modifier's syntax is:

%left '+' '-'

%left UMINUS

Now the precedence of UMINUS

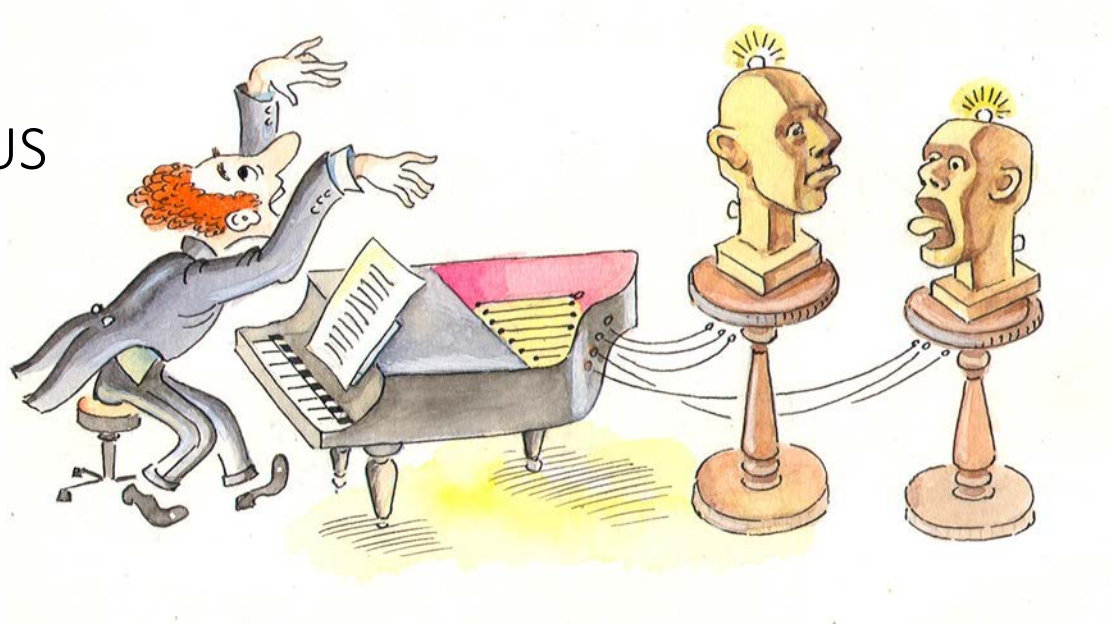
can be used in specific rules:

exp: ...

| exp '-' exp

...

| '-' exp %prec UMINUS



BISON & FLEX

The file generated by BISON during its work contains the analyzer tables and the C-text of the *int yyparse (void)* function, which implements the table interpreter and semantic actions. For starting the parser, it is enough to call this function.

If parsing is successful, it returns 0, on error - 1.

To get the next token, the parser calls the *int yylex (void)* function. It should return a token code and place its value in the *YYSTYPE yylval* variable.

The token code is a positive integer. The tokens, specified as character constants, correspond to their code in the character set (usually ASCII), in the range 0..255. Lexemes, those with symbolic names are assigned codes starting with 258.

The output file contains *#define* statements that define token names as codes. If the names of tokens are required in other files (keep these definitions in the *y.tab.h* file). The output file for the FLEX is usually named lex.yy.c. So it's a good idea to keep a source file names for FLEX & BISON as a standard.

The current version of BISON (the GNU parser generator) is 3.8.2 (25 September 2021). The distribution terms for BISON-generated parsers permit using the parsers in nonfree programs. Earlier, BISON-generated parsers could be used only in programs that were free software. The main output of the BISON utility contains a verbatim copy of a sizable piece of BISON, which is the code for the parser's implementation. Most of the codes of the implementation is not changed due to change the grammar rules. Limiting BISON's use to free software was doing little to encourage people to make other software free. The practical conditions for using BISON match the practical conditions for using the other GNU tools now.

But Software should be FREE.