BISON

- BISON takes an input grammar file and produces a C program that parses the language described by that grammar.
- Tokens are read from the lexical analyser function `yylex()` which can be coded manually or generated automatically using FLEX.
- The BISON output file (C program) defines a function called `yyparse()` – the implementation of the grammar.
- The parser generated by BISON expects a user-implemented error reporting function `yyerror()`. And the `main()` C function.

BISON Grammar Files

- A BISON grammar file contains four sections separated by delimiters:

```
%{
C Declaration
%}

Bison Declarations

%%
Grammar Rules

%%
Additional C Code
```

The C Declarations Section

- This section contains global definitions, constants, variables, #include's, #define's and functions that will be used in the actions of the grammar rules.
- The contents of this section are copied to the very beginning of the output parser file so that they precede the `yyparse()` function.
- If no C declarations are used, the `%{` and `%}` delimiters may be omitted.
- By now you should have noticed that both BISON and FLEX have a lot of variable and function definitions starting with yy. It is a good idea NOT to name any variables or functions of your own starting with yy too.

Other Sections

- Bison Declarations
  - This section contains declarations of terminal and non-terminal symbols used in the language being described, as well as definitions of operator precedence and the data types of semantic values of various symbols.
- Grammar Rules Section
  - This section contains the grammars production rules, which define how a non-terminal is constructed from its parts. There must always be at least one grammar rule in a BISON file.
- Additional C Code
  - Like the C Declarations section, the contents of this section contains C code that is copied exactly to the output. This section is copied to the END of the output file. It is a convenient way to put anything required AFTER the `yyparse()` function such as `main()`.

Symbols – Terminal and Non-Terminal (1)

- A terminal symbol represents lexical analyser tokens. These tokens are represented by numeric constants, any `yylex()` returns a token type code to indicate what token type has been read.
- There are 2 ways of writing terminal symbols in the grammar:
  - Single Characters: A single character token type such as + or * does not need to be declared. It can be used directly in the rules section by enclosing it in single quotes.
  - Multi-Character Tokens: These are represented by a declared name using a `%token` declaration. By convention names are written in upper case. For example the words “Begin” and “End” might be declared as:

```
%token  BEGIN
%token  END
```

-or-

```
%token  BEGIN END
```


Symbols – Terminal and Non-Terminal (2)

- Internally, each token is represented by an integer, starting from 257. 0 to 255 are used to represent ASCII characters and 256 is used to represent the error token. When using BISON the programmer is not generally concerned about these values but these token values must be known when implementing these tokens in FLEX. Using the ‘–d’ option when running BISON will automatically generate an include file containing the definitions for these tokens:

```
...
#define BEGIN 257
#define END 258
...
```

Symbols – Terminal and Non-Terminal (3)

- Non-terminals are declared in exactly the same way, but their names are in lowercase by convention.

BISON Grammar Rules (1)

- A BISON grammar rule has the form:

```
result    : components...
          ;
```

- Where result is the non-terminal symbol that the rule describes (LHS) and the components are the various terminal and non-terminal symbols that put together this rule (RHS).
- For example:

```
exp             : exp '+' exp
                ;
if_statement    : IF exp THEN
                ;
```

BISON Grammar Rules (2)

- Multiple rules for the same result can be written separately:

```
exp     : exp '+' exp;
exp     : exp '-' exp;
```

- Or together, separated by the vertical bar:

```
exp     : exp '+' exp
        | exp '-' exp
        ;
```

- If the components section is left empty, it means that result can match the empty string.

BISON Grammar Rules (3)

- Here is how to define a comma separated sequence of zero or more `exp` groupings:

```
expseq          : /* empty */
                | expseq1
                ;
expseq1         : exp
                | expseq1 ',' exp
                ;
```

- It is convention to write a comment `/* empty */` in each rule with no component.
- Within components actions consisting of C statements may be included:

```
exp : exp '+' exp { printf("Addition Expression");}
    | exp '-' exp { printf("Subtraction Expression");}
    ;
```

Recursive Rules (1)

- A rule is called recursive when its result also appears also on the right-hand side. Nearly all BISON grammars need to use recursion, because it is the only way to define a sequence (zero-or-more, one-or-more) of 'somethings'.
- Consider the left and right recursive definitions of a comma-separated sequence of one or more expressions:

```
expseqleft: exp
                | expseqleft ',' exp
                ;
expseqright     : exp
                | exp ',' expseqright
                ;
```


Recursive Rules (2)

- Any kind of sequence may be defined using either left or right recursion, but one should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space.
- Indirect or mutual recursion occurs when the result of the rule does not appear directly on the right hand side, but does appear in rules for other non-terminals which do appear on its right hand side. For example:

```
expr            : primary
                | primary '+' primary
                ;
primary         : constant
                | '(' expr ')'
                ;
```

Defining Semantics (1)

- The grammar rules for a language determine only its syntax. The semantics are determined by the semantic ‘meaning’ associated with various tokens and the actions taken when these tokens are recognised.
- A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol `integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `x+4' is grammatical then `x+1' or `x+3989' is equally grammatical.

Defining Semantics (2)

- Semantic values have all the rest of the information about the meaning of a token, such as the value of an integer, or the name of an identifier. (A token such as ',' which is just punctuation doesn't need to have any semantic value.)
- For example, an input token might be classified as token type INTEGER and have the semantic value 4. Another input token might have the same token type INTEGER but value 3989. When a grammar rule says that INTEGER is allowed, either of these tokens is acceptable because each is an INTEGER. When the parser accepts the token, it keeps track of the token's semantic value.
- Each grouping can also have a semantic value as well as its non-terminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

Defining Semantics (3)

- Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the sub-expressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.
- The C code in an action can refer to the semantic values of the components matched by the rule with the construct `$`*n*, which stands for the value of the *n*th component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)
- Here is a typical example:

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


Defining Semantics (4)

- If you don't specify an action for a rule, Bison supplies a default: `$$ = $1`.
- Every terminal and non-terminal defined in the grammar is given a type. Bison's default is to use the `int` type for all semantic values. Clearly, this can be overridden.

BISON Declarations

- This section defines all the symbols used in formulating the grammar and the data types of semantic values.
- All token types except for single character tokens (such as +, which are enclosed in single quotes) must be declared.
- Non-terminal symbols must be declared if you need to specify which data type to use for the semantic value.

The Sentence Symbol

- The sentence symbol of the grammar is, by default, the first non-terminal defined at the start of the rules section. An alternative start symbol may be specified using the `%start` statement. For example, if the starting symbol in your language is `program`, you may specify this as:

```
%start program
```

Token Types

- The basic way to specify a token is using the `%token` statement:

```
%token begin
%token end
```

- One can explicitly specify a numeric code to each token type:

```
%token begin      300
%token end        301
```

- But, in general, it is better to let BISON choose the numeric code itself.

Types of Semantic Values (1)

- The BISON `%union` declaration is used to specify the collection of all possible data types for all the semantic values:

```
%union
{
  double val;
  char * str;
}
```

- This means that we defined two types – val (based on the double type) and str (a C string).

Types of Semantic Values (2)

- In certain cases, token declarations (`%token …`) should be assigned a type. For example, if the token `NUM` must be associated to the semantic type `double`, then the token declaration should be modified as:

```
%token <val> NUM
```

- We previously mentioned that in some cases, non-terminals could be associated with a semantic type. In this case the non-terminal declaration is mandatory. Suppose the EXPR and PRIMARY non-terminals are associated a double type:

```
%type <val> EXPR PRIMARY
```


Associativity

- If one wishes to declare a token and specify its associativity the `%left`, `%right` and `%nonassoc` statements are used.
- If '+' is declared to be left associative:

```
%left '+'
```

- The same reasoning goes for right associativity. A token may be non-associative. Say, '+' should be declared with no associative information. We get:

```
%nonassoc '+'
```

- But keep in mind that statements like `'a+b+c'` will be considered as a syntax error (we have more than one operator but don't have associativity information)

Precedence

- All tokens declared together have the same precedence.
- When tokens are declared separately, the one declared later has the highest precedence.

```
%left '+' '-'
%left '*'
```