***Bottom-up Parsing***

-Bottom-up parsing is more general than top down parsing – And just as efficient – Builds on ideas in top-down parsing.

-Bottom-up is the preferred method

* Idea: Split string into two substrings
  + – Right substring is as yet unexamined by parsing (a string of terminals)
  + – Left substring has terminals and non-terminals
* The dividing point is marked by a |
  + – The | is not part of the string
* Initially, all input is unexamined |x1x2 . . . xn
* Bottom-up parsing uses only two kinds of actions:
  + Shift
  + Reduce
* Shift: Move | one place to the right
  + – Shifts a terminal to the left string
    - ABC|xyz ⇒ ABCx|yz
* Apply an inverse production at the right end of the left string
  + – If A → xy is a production,
    - Cbxy|ijk ⇒ CbA|ijk

**The stack:** Left string can be implemented by a stack

* + Top of the stack is the |
* Shift pushes a terminal on the stack
* Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a nonterminal on the stack (production lhs)

**Conflicts:** In a given state, more than one action (shift or reduce) may lead to a valid parse

* If it is legal to shift or reduce, there is a shift reduce conflict
* If it is legal to reduce by two different productions, there is a reduce-reduce conflict

***The parser generator***

***Yacc-bison***

* Yacc and Bison are tools for generating parsers (programs which recognize the grammatical structure of programs.
* Bison is a faster version of Yacc.

**Yacc file structure:**

%{

C declarations

%}

Bison declarations

%%

Grammar rules and actions

%%

C subroutines

**Running commands:**

1. Save yacc file with .y extension (ex: Hello.y).
2. build the parser by: bison –d Hello.y

* Will produce two files:
  + 1. Hello.tab.c
    2. Hello.tab.h

//the command-line argument -d to produce the header file

//this file contains definitions of token codes

1. Compile : gcc Hello.tab.c

To compile and link flex with bison files: gcc lex.yy.c Hello.tab.c

**Bison Input:**

* The non-terminal symbols are given in all lower case, the terminal symbols are given in all caps or as literal symbols and, where the literal symbols conflict with the meta symbols of the EBNF, they are enclosed with single quotes.
  + - stmt: RETURN expr ';'

;

* + - The semicolon in quotes is a literal character [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31), the naked semicolon, and the colon, are Bison punctuation used in every rule.
* The start symbol is program. While the grammar uses upper-case to high-light terminal symbols, they are to be implemented in lower case.
* There are two context sensitive requirements; variables must be declared before they are referenced and a variable may be declared only once.
* each [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) in a Bison grammar has both a [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) type and a **semantic value**
* In a Bison grammar, a grammar rule can have an **action** made up of C statements. Each time the [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) recognizes a match for that rule, the action is executed.

expr: expr '+' expr { $$ = $1 + $3; }

;

* The Bison [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) calls the lexical analyzer each time it wants a new [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31).
* **yyparse** function implements that grammar.
* The C declarations may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use #include to include header files that do any of these things.
* The Bison declarations declare the names of the terminal and nonterminal symbols, and 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 additional C code can contain any C code you want to use. Often the definition of the lexical analyzer yylex goes here, plus subroutines called by the actions in the grammar rules.
* The #define directive defines the macro YYSTYPE, thus specifying the C data type for semantic values of both tokens and groupings .The Bison [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) will use whatever type YYSTYPE is defined as; if you don't define it, int is the default. Because we specify double, each [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) and each expression has an associated value, which is a floating point number.

%{

#define YYSTYPE double

%}

* Alternative rules of non-terminal symbol, joined by the `|' punctuator which is read as "or".
* In each action, the pseudo-variable $$ stands for the semantic value for the [grouping](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS6) (lhs). The semantic values of the components of the rule are referred to as $1, $2, and so on.

exp: NUM { $$ = $1; }

| exp exp '+' { $$ = $1 + $2; }

;

* Empty string(no tokens)

input: /\* empty \*/

;

* The line’s semantic value is uninitialized

line: '\n'

| exp '\n' { printf ("\t%.10g\n", $1); }

* The return value of the lexical analyzer function is a numeric code which represents a [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) type.
* The semantic value of the [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) (if it has one) is stored into the global variable yylval, which is where the Bison [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) will look for it. (The C data type of yylval is YYSTYPE, which was defined at the beginning of the grammar;
* A [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) type code of zero is returned if the end-of-file is encountered. (Bison recognizes any nonpositive value as indicating the end of the input.)
* call yyparse to start the process of parsing

main ()

{

yyparse ();

}

* When yyparse detects a syntax error, it calls the error reporting function yyerror to print an error message

int yyerror(char const \*msg) {

printf("Error: %s\n", msg);

return 0;

}

* The declarations %left and %right (right associativity) take the place of %[token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) which is used to declare a [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) type name without associativity.
* Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration, the higher the precedence

Error recovery

line: '\n'

| exp '\n' { printf ("\t%.10g\n", $1); }

| error '\n' { yyerrok; }

;

* + how to continue parsing after the [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) detects a syntax error.
  + The Bison language itself includes the reserved word error, which may be included in the grammar rules.
  + If an expression that cannot be evaluated is read, the error will be recognized by the third rule for line, and parsing will continue. (The yyerror function is still called upon to print its message as well.) The action executes the statement yyerrok, a macro defined automatically by Bison; its meaning is that error recovery is complete.
  + There are other kinds of errors; for example, division by zero, which raises an exception signal that is normally fatal. A real calculator program must handle this signal and use longjmp to return to main and resume parsing input lines; it would also have to discard the rest of the current line of input. We won't discuss this issue further because it is not specific to Bison programs.

**Others:**

* + The %prec simply instructs Bison that the rule `| '-' exp' has the same precedence as NEG-

/\* BISON Declarations \*/

%[token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) NUM

%left '-' '+'

%left '\*' '/'

%left NEG /\* negation--unary minus \*/

%right '^' /\* exponentiation \*/

Exp: |'-' exp %prec NEG { $$ = -$2; }

* ***Union***

%{

#include <math.h> /\*For math functions, cos(), sin(),etc. \*/

#include "calc.h" /\* Contains definition of `symrec' \*/

%}

%union {

double val; /\* For returning numbers. \*/

symrec \*tptr; /\* For returning symbol-table pointers \*/

}

%[token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) <val> NUM /\* Simple double precision number \*/

%[token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) <tptr> VAR FNCT /\* Variable and Function \*/

%type <val> exp

%right '='

%left '-' '+'

%left '\*' '/'

%left NEG /\* Negation--unary minus \*/

%right '^' /\* Exponentiation \*/

* + The %union declaration specifies the entire list of possible types; this is instead of defining YYSTYPE. The allowable types are now double-floats (for exp and NUM) and pointers to entries in the symbol table.
  + The Bison construct %type is used for declaring nonterminal symbols, just as %[token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) is used for declaring [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) types.

***Associativity:***

* The associativity of an operator *op* determines how repeated uses of the operator nest: whether `*x* *op* *y* *op* *z*' is parsed by [grouping](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS6) *x* with *y* first or by [grouping](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS6) *y* with *z* first. %left specifies left-associativity ([grouping](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS6) *x* with *y* first) and %right specifies right-associativity ([grouping](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS6) *y* with *z* first). %nonassoc specifies no associativity, which means that `*x* *op* *y* *op* *z*' is considered a syntax error.

**[The Start-Symbol](https://www.math.utah.edu/docs/info/bison_toc.html" \l "SEC55)**

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the %start declaration as follows:

%start *symbol*

### [Suppressing Conflict Warnings](https://www.math.utah.edu/docs/info/bison_toc.html" \l "SEC54)

The declaration looks like this:

%expect *n*

Here *n* is a decimal integer. The declaration says there should be no warning if there are *n* [shift](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS27)/reduce conflicts and no reduce/reduce conflicts. The usual warning is given if there are either more or fewer conflicts, or if there are any reduce/reduce conflicts.

* Use the `-v' option to get a list of where the conflicts occur. Bison will also print the number of conflicts.
* Check each of the conflicts to make sure that Bison's default resolution is what you really want. If not, rewrite the grammar and go back to the beginning

## [Look-Ahead Tokens](https://www.math.utah.edu/docs/info/bison_toc.html" \l "SEC69)

* + when a[reduction](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS22) is possible, the [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) sometimes "looks ahead" at the next [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) in order to decide what to do, Also for shifting.
  + The current look-ahead [token](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS31) is stored in the variable yychar.

## [Stack Overflow, and How to Avoid It](https://www.math.utah.edu/docs/info/bison_toc.html#SEC80)

* The Bison [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) stack can overflow if too many tokens are shifted and not reduced. When this happens, the [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) function yyparse returns a nonzero value, pausing only to call yyerror to report the overflow.

By defining the macro YYMAXDEPTH, you can control how deep the [parser](https://www.math.utah.edu/docs/info/bison_14.html#GLOSS20) stack.

* The default value of YYMAXDEPTH, if you do not define it, is 10000.

Reference:

<https://www.math.utah.edu/docs/info/bison_1.html>