Bison
Parser Generator

Scanning/parsing tools

- **lex** - original UNIX lexics generator (Lesk, 1975).
  - create a C function that will parse input according to a set of regular expressions.
- **yacc** - "*yet another compiler compiler*" UNIX parser (Johnson, 1975).
  - generate a C program for a parser from BNF rules.
- **bison** and **flex** ("fast lex") - more powerful, free versions of yacc and lex, from GNU Software Fnd'n.
- **Jflex** - generates Java code for a scanner.
- **CUP** - generates Java code for a parser.

Bison Overview

- *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.
- Interfaces with scanner generated by Flex.
  - Scanner called as a subroutine when parser needs the next token.

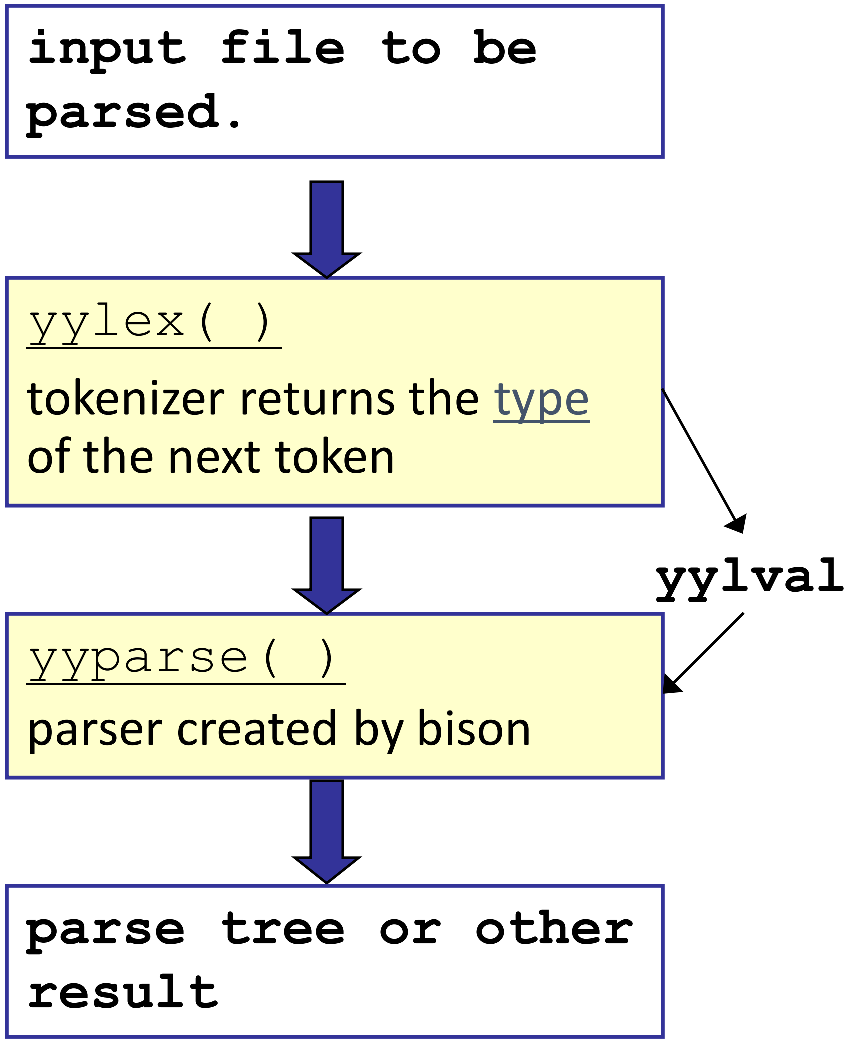
Bison Overview

- **Purpose:** automatically write a parser program for a grammar written in BNF.
- **Usage:** you write a bison source file containing rules that look like BNF. Bison creates a C program that parses according to the rules

```
term     : term '*' factor { $$ = $1 * $3; }
         | term '/' factor { $$ = $1 / $3; }
         | factor          { $$ = $1; }
         ;
factor   : ID              { $$ = valueof($1); }
         | NUMBER          { $$ = $1; }
        ;
```

Bison Overview

- In operation:
- your main program calls yyparse( ).
- yyparse( ) calls yylex when it wants a token.
- yylex returns the **type** of the token.
- yylex puts the **value** of the token in a global variable named yylval



Bison input file format

- The input file consists of three sections, separated by a line with just **`%%'** on it:

```
%{
   C declarations (types, variables, functions, preprocessor commands)
%}

/*  Bison declarations (grammar symbols, operator precedence decl., attribute data type)  */

%%
/* grammar rules go here */

%%
/* additional C code goes here */
```

C Declarations

- This section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules.
- You can use “#include” to get the declarations from a header file.
- If you don’t any C declarations, you may omit the “%{ and %}” delimiters that bracket this section.

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

Bison Declarations

- Define terminal and nonterminal symbols.
- Define attributes and their associations with terminal and nonterminal symbols.
- Specify precedence and associativity.

```
%union {
  int val;
  char *varname;
  }
  %type <val> exp
  %token <varname> NAME
  %right =
  %left + -
  %left * /
```

Bison Grammar Section

- A grammar
  - is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet (tokens) that are valid according to the language's syntax.
  - ```
    E → E + E
      | E – E
      | E * E
      | E / E
      | id
    ```
- A simple grammar that allows recursive math operations.
- There must always be at least one grammar rule.
    ```

Bison Grammar Section

- A Bison grammar rule has the following general form:
  - `result : components.......`

    `;`
- ***result*** is the nonterminal symbol that this rule describes.
- *components* are various terminal and nonterminal symbols that are put together by this rule.
- Example:

```
expr : expr '+' expr
;
```

- The grammar says that two grouping of type expr, with a '+' token in between, can be combined into a larger grouping of type expr.

Bison Grammar Section

- If ***components*** in a rule is empty, it means that *result* can match the empty string.
- For example, here is how to define a comma-separated sequence of zero or more expr groupings.
-
```
expseq :  `/* empty */
          |expseq1
          ;
expseq1:  expr
          | expseq1 `,' expr
          ;
```
- It is customary to write a comment ‘/* empty */’ in each rule with no components.

Bison – Grammar and Actions

- An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized.
- The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.
- An action consists of C statements surrounded by braces, much like a compound statement in C.
- Example:

```
exp : exp '+' exp { && = $1 + $3;
                    printf("hello world");
                  }
```

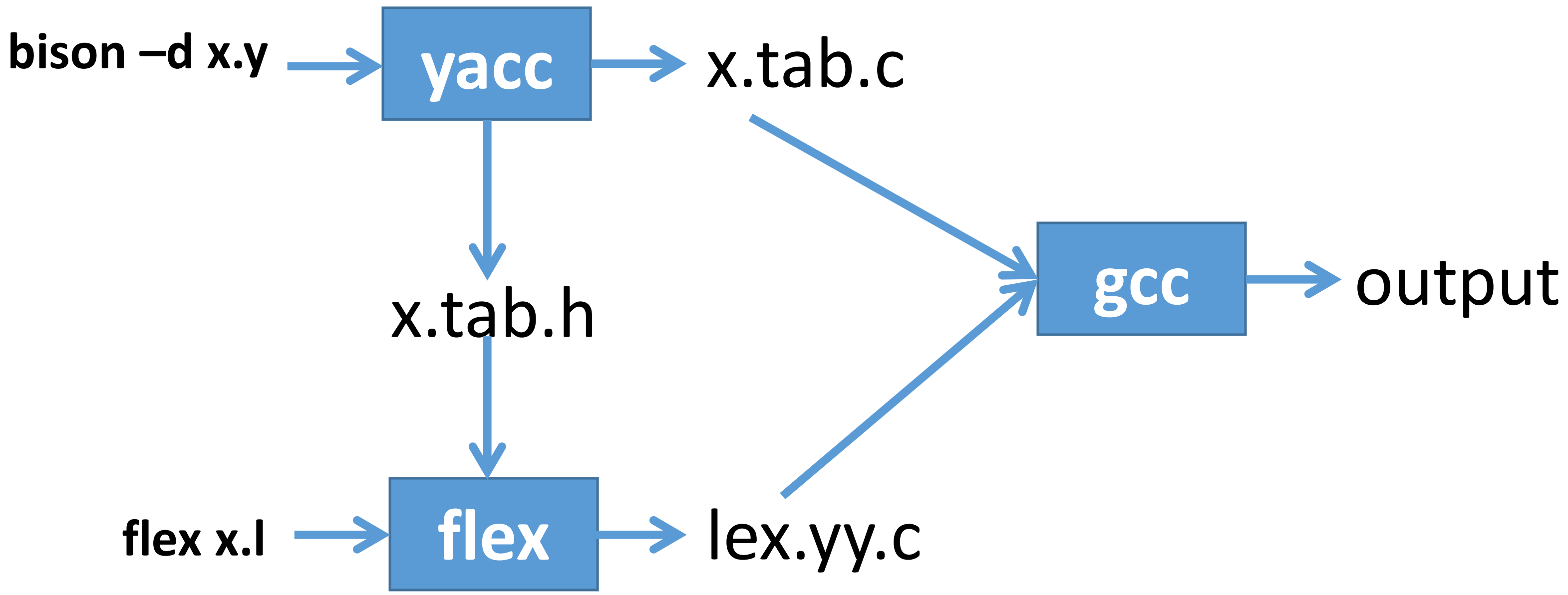
Semantic Values and Actions

- Actions can manipulate semantic values associated with a nonterminal.
  - ***$n*** refers to the semantic value (synthesized attribute) of the ***n-th*** symbol on the ***RHS***.
  - ***$$*** refers to the semantic value of the ***LHS*** nonterminal..
  - Typically, an action is of the form:

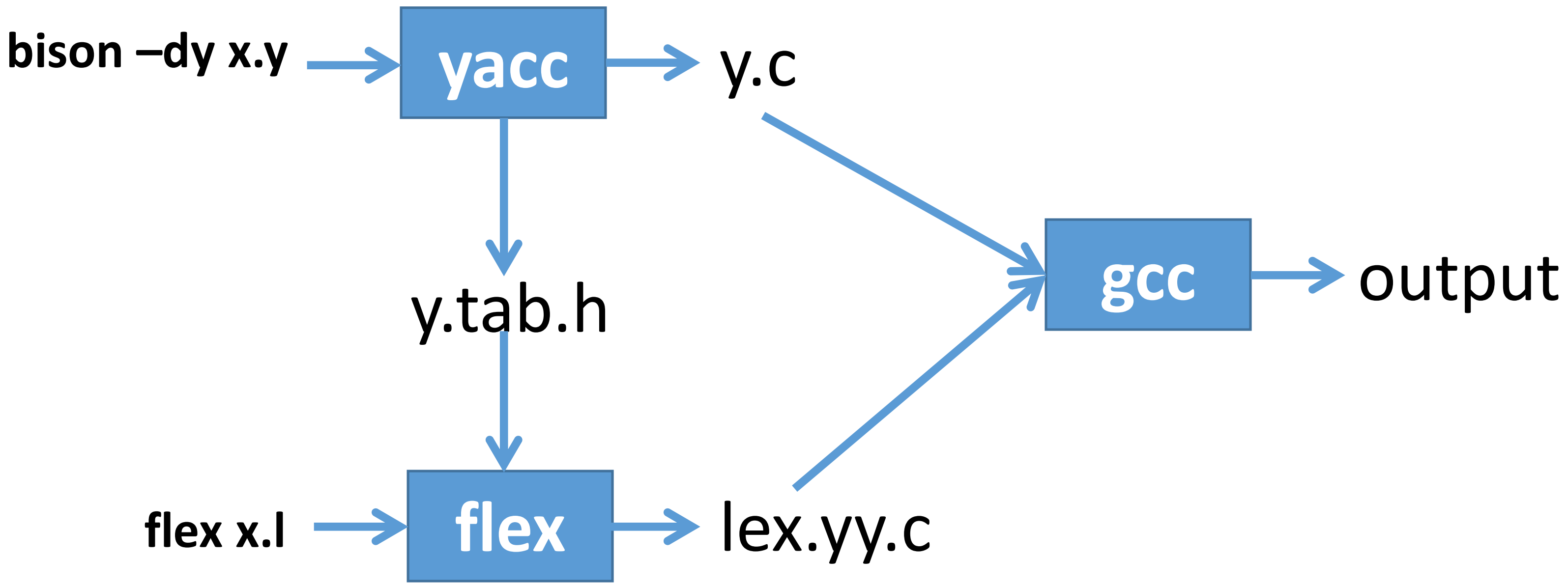
```
$$ = f( $1, $2, …$m)
```

- The types for the semantic values are specified in the declaration section.

Compiler with Flex/lex and Yacc/Bison



Compiler with Flex/lex and Yacc/Bison



Bison Flex Different File – Command

- Process the **bison** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code): ***bison –d file_name.y***
- Use the **dir** command to verify that the following files were created: ***file_name.tab.c***

The C language source file that the **yacc** command created for the parser

file_name.tab.h A header file containing definitions for token names.

- Process the f**lex** specification file: ***flex file_name.l***
- Use the **dir** command to verify that the following file was created: ***lex.yy.c***
- Compile and link the two C language source files:

gcc file_name.tab.c lex.yy.c –o output

Bison Flex Different File – Example

Odd_even.l

Odd_even.y

Bison Flex Same File – Command

- Process the **bison** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

  bison -d file_name.y

- Use the **dir** command to verify that the following files were created:

  file_name.tab.c The C language source file that the **yacc** command created for the parser

  file_name.tab.h A header file containing definitions for token names.

- Compile the C language source files: ***gcc file_name.tab.c –o output***

Bison Flex Same File – Example

Odd_even_one.y

Bison Flex Different File – Example

id.l

id.y

Bison Flex Different File – Example

[**infix.l**]

[**infix.y**]

Bison Flex Different File – Example

postfix.l

postfix.y

Bison Flex Same File – Example

lfn_com.y

Bison Flex Same File – Example

[rpn_com.y](rpn_com.y)

Bison Example

Create a parser for this grammar:

```
expression => expression + term
            | expression - term
            | term
term    =>  term * factor
            | term / factor
            | factor
factor =>   ( expression )
            | NUMBER
```

Bison/Yacc file for example (1)

Structure of Bison or Yacc input:

```
%{
/* C declarations and #DEFINE statements go here */
  #include <stdio.h>
  #define YYSTYPE double
%}
/* Bison/Yacc declarations go here */
%token NUMBER     /* define token type NUMBER */
%left '+' '-'     /* + and - are left associative */
%left '*' '/'     /* * and / are left associative */

%%
/* grammar rules go here */
%%
/* additional C code goes here */
```

Bison/Yacc file for example (2)

```
%%          /* Bison grammar rules */
input     : /* empty production to allow an empty input */
          | input line
          ;
line      : expr '\n'    { printf("Result is %f\n", $1); }
expr      : expr '+' term    { $$ = $1 + $3; }
          | expr '-' term    { $$ = $1 - $3; }
          | term             { $$ = $1; }
          ;
term      : term '*' factor { $$ = $1 * $3; }
          | term '/' factor { $$ = $1 / $3; }
          | factor           { $$ = $1; }
          ;
factor    : '(' expr ')'     { $$ = $2; }
          | NUMBER           { $$ = $1; }
          ;
```

Bison/Yacc file for example (3)

- $1, $2, ... represent the actual values of tokens or non-terminals (rules) that match the production.
- $$ is the result.

rule — pattern to match — action

```
expr    : expr '+' term  { $$ = $1 + $3; }
        | expr '-' term  { $$ = $1 - $3; }
        | term           { $$ = $1; }
        ;
```

Example:

if the input matches **`expr + term`** then set the result (**`$$`**) equal to the sum of **`expr`** plus **`term`** (**`$1`** + **`$3`**).

Scanner function for double

- Now yylex must know that yylval is "extern double".
- Here is example of using scanf to parse numbers.

```
int yylex( void ) {
    int c = getchar();               /* read from stdin */
    if (c < 0) return 0;             /* end of the input*/
    while ( c == ' ' || c == '\t' ) c = getchar( );
    if ( isdigit(c) || c == '.' ) {
    ungetc(c, stdin);          /* put c back into input */
    scanf ("%lf", &yylval); /* get value using scanf */
    return NUMBER;             /* return the token type */
    }
    return c; /* anything else... return char itself */
}
```

Other C functions: main

- you need a write a **main()** function that starts the parser.
- For a simple parser, **main()** calls **yyparse()**.

```
/* main method to run the program */
int main( ) {
       printf("Type some input. Enter ? for help.\n");
       yyparse( );
}
```