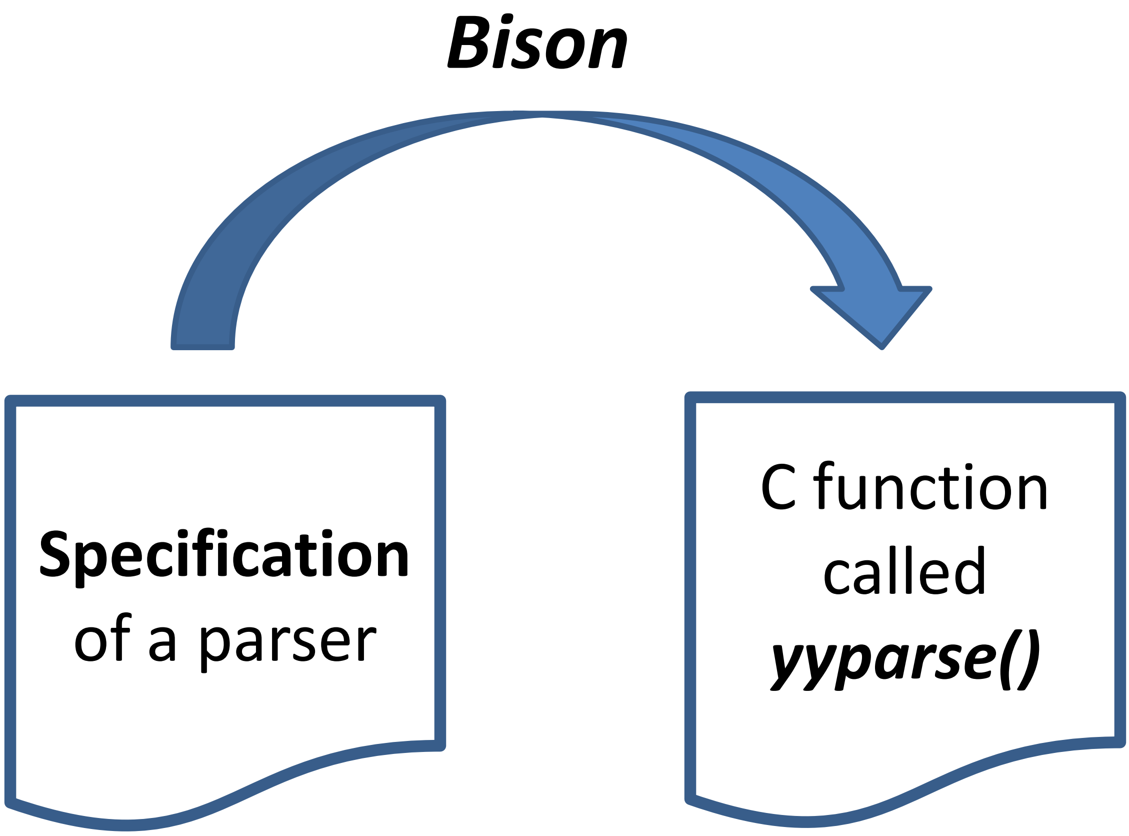
Lexical and Syntax Analysis

(of Programming Languages)

Bison, a Parser Generator

Bison: a parser generator



Context-free grammar *with a **C action** for each production.*

*Match the input string and **execute the actions** of the productions used.*

Input to ***Bison***

The structure of a ***Bison*** (*.y*) file is as follows.

```
/* Declarations */

%%

/* Grammar rules */

%%

/* C Code (including main function) */
```

Any text enclosed in /* and */ is treated as a **comment**.

Grammar rules

Let α be any sequence of **terminals** and **non-terminals**. A **grammar rule** defining non-terminal *n* is of the form:

$$
\begin{array}{llll}
n & : & \alpha_1 & action_1 \\
 & | & \alpha_2 & action_2 \\
 & | & \cdots & \\
 & | & \alpha_n & action_n \\
 & ; & &
\end{array}
$$

Each **action** is a C statement, or a block of C statements of the form *{⋯}*.

Example 1

expr1.y

```
/* No declarations */

%%

e        :   'x'                    /* No actions */
         |   'y'
         |   '(' e '+' e ')'
         |   '(' e '*' e ')'

%%

/* No main function */
```

Terminal

Non-terminal

Output of ***Bison***

Bison generates a C function

```
int yyparse() {
   …
}
```

- Takes as input a stream of **tokens**.
- Returns **zero** if input conforms to grammar, and non-zero otherwise.

- Calls ***yylex()*** to get the next token.
- Stops when *yylex()* returns zero.

- When a grammar rule is used, that rule's action is **executed**.

Example 1, revisted

expr1.y

```
/* No declarations */

%%

e     :  'x'                    /* No actions */
      |  'y'
      |  '(' e '+' e ')'
      |  '(' e '*' e ')'

%%

int yylex() {
   char c  = getchar();
   if (c == '\n') return 0; else return c;
}

void main() {
   printf("%i\n", yyparse());
}
```

Running Example 1

At a command prompt '>':

```
> bison -o expr1.c  expr1.y

> gcc -o expr1 expr1.c -ly

> expr1
(x+(y*x))
0
```

Important!

Input

Output
(0 means successful parse)

Example 2

Terminals can be declared using a ***%token*** declaration, for example, to represent arithmetic variables:

expr2.y

```
%token VAR

%%

e       :   VAR
        |   '(' e '+' e ')'
        |   '(' e '*' e ')'

%%

/* main() and yylex() */
```

Example 2 (continued)

expr2.y

```
int yylex() {
    int c = getchar();

    /* Ignore white space */
    while (c == ' ') c = getchar();

    if (c == '\n') return 0;
    if (c >= 'a' && c <= 'z')
        return VAR;
    return c;
}

void main() {
    printf("%i\n", yyparse());
}
```

Return a *VAR* token

Example 2 (continued)

Alternatively, the *yylex()* function can be generated by ***Flex***.

expr2.lex

```
%{
#include "expr2.h"
%}

%%

" "            /* Ignore spaces */
\n             return 0;
[a-z]          return VAR;
.              return yytext[0];

%%
```

Generated by Bison

Running Example 2

At a command prompt '>':

> **bison** *--defines -o expr2.c expr2.y*

> ***flex*** *-o expr2lex.c expr2.lex*

> ***gcc*** *-o expr2 expr2.c expr2lex.c* ***–ly -lfl***

> *expr2*
*(a   +   (b * c))*
0

Generate
expr2.h

Parser

Lexer

Output
(0 means successful parse)

Example 3

Adding numeric literals:

expr3.y

```
%token VAR
%token NUM

%%

e       :   VAR
        |   NUM
        |   '(' e '+' e ')'
        |   '(' e '*' e ')'

%%

void main() {
    printf("%i\n", yyparse());
}
```

Numeric Literal

Example 3 (continued)

Adding numeric literals:

expr3.lex

```
%{
#include "expr3.h"
%}

%%

" "             /* Ignore spaces */
\n              return 0;
[a-z]           return VAR;
[0-9]+          return NUM;
.               return yytext[0];

%%
```

Numeric Literal

Semantic values of tokens

A token can have a **semantic value** associated with it.

- A *NUM* token contains an integer.
- A *VAR* token contains a variable name.

Semantic values are returned via the ***yylval*** global variable**.**

Example 3 (revisited)

Returning values via *yylval*:

expr3.lex

```
%{
#include "expr3.h"
%}
%%
" "             /* Ignore spaces */
\n              return 0;
[a-z]           { yylval = yytext[0];
                  return VAR; }
[0-9]+          { yylval = atoi(yytext);
                  return NUM;
                }
.               return yytext[0];
%%
```

Type of *yylval*

Problem: different tokens may have semantic values of different types. So what is type of *yylval*?

Solution: a union type, which can be specified using the ***%union*** declaration, e.g.

```
%union{
    char var;
    int num;
}
```

yylval is either a ***char*** or an ***int***

Example 3 (revisted)

Returning values via *yylval*:

expr3.lex

```
%{    #include "expr3.h"    %}

%%

" "            /* Ignore spaces */
\n             return 0;
[a-z]          { yylval.var = yytext[0];
                 return VAR; }
[0-9]+         { yylval.num = atoi(yytext);
                 return NUM;
               }
.              return yytext[0];

%%
```

Tokens have types

The type of token's semantic value can be specified in a ***%token*** declaration.

```
%union{
    char var;
    int num;
}

%token <var> VAR;
%token <num> NUM;
```

Semantic values of non-terminals

A non-terminal can also have a **semantic value** associated with it.

In the action of a grammar rule:

- ***$n*** refers to the semantic value of the n^{th} symbol in the rule;
- ***$$*** refers to the semantic value of the result of the rule.

The type can be specified in a ***%type*** declaration, e.g.

```
%type <num> e;
```

Example 4

expr4.y

```
%{
    int env[256];   /* Variable environment */
%}
%union{  int num; char var;  }
%token <num> NUM
%token <var> VAR
%type  <num> e

%%

s   :   e                   { printf("%i\n", $1);     }

e   :   VAR                 { $$ = env[$1];            }
    |   NUM                 { $$ = $1;                 }
    |   '('  e  '+'  e  ')' { $$ = $2 + $4;            }
    |   '('  e  '*'  e  ')' { $$ = $2 * $4;            }

%%

void main() {   env['x'] = 100;  yyparse();   }
```

Exercise 1

Consider the following **abstract syntax**.

```
typedef enum { Add, Mul } Op;

struct expr {
    enum { Var, Num, App } tag;
    union {
        char var;
        int num;
        struct {
            struct expr* e1; Op op; struct expr* e2;
        } app;
    };
};

typedef struct expr Expr;
```

Modify Example 4 so that *yyparse()* constructs an abstract syntax tree.

Precedence and associativity

The **associativity** of an operator can be specified using a ***%left***, ***%right***, or ***%nonassoc*** directive.

%left '+’
%left ''*
%right '&'
%nonassoc '='

Operators specified in increasing order of **precedence**, e.g. '*' has higher precedence than '+'.

Example 5

expr5.y

```
%token VAR
%token NUM

%left '+'
%left '*'

%%

e       :   VAR
        |   NUM
        |   e  '+'  e
        |   e  '*'  e
        |   ( e )

%%

void main() {
   printf("%i\n", yyparse());
}
```

Conflicts

Sometimes *Bison* cannot deduce that a grammar is unambiguous, even if it is*.

In such cases, Bison will report:

- a **shift-reduce** conflict; or
- a **reduce-reduce** conflict.

* Not surprising: ambiguity detection is undecidable in general!

Shift-Reduce Conflicts

Bison does not know whether to consume more tokens (shift) or to match a production (reduce), e.g.

```
stmt  :  IF expr THEN stmt
      |  IF expr THEN stmt ELSE stmt
```

Bison defaults to shift.

Reduce-Reduce Conflicts

Bison does not know which production to choose, e.g.

```
expr          :  functionCall
              |  arrayLookup
              |  ID

functionCall  :  ID '(' ID ')'

arrayLookup   :  ID '(' expr ')'
```

Bison defaults to using the first matching rule in the file.

Variants of ***Bison***

There are ***Bison*** variants available for many languages:

Language	Tool
Java	JavaCC, CUP
Haskell	Happy
Python	PLY
C#	Grammatica
*	ANTLR

Summary

- ***Bison*** converts a context-free grammar to a parsing function called *yyparse()*.
- *yyparse()* calls *yylex()* to obtain next token, so easy to **connect** *Flex* and *Bison*.

Summary

- Each grammar production may have an **action**.
- Terminals and non-terminals have **semantic values**.
- Easy to construct **abstract syntax trees** inside actions using semantic values.
- Gives a declarative (**high level**) way to define parsers.