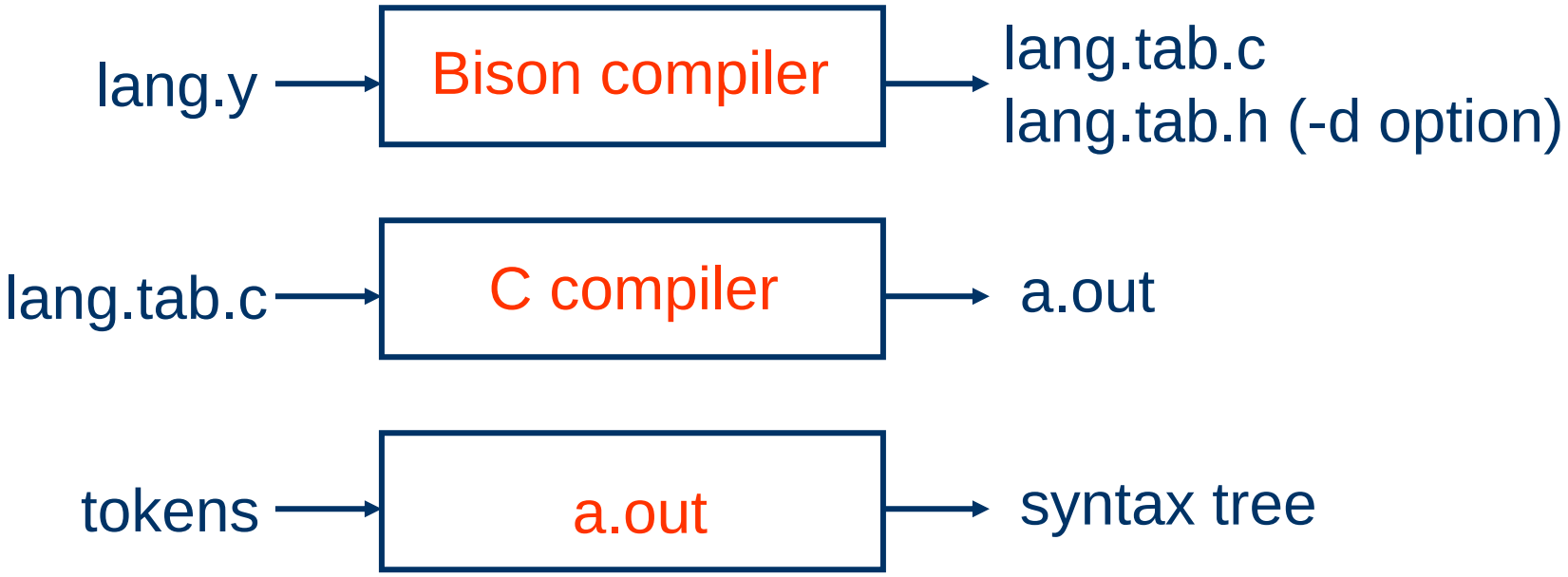
Yacc / Bison Parser Generator

Parser Generators

- Yacc = yet another compiler-compiler
- Bison = a faster version of Yacc



What is Yacc / Bison

- Bison produces the parser (it is not a parser)
- Bison can be also used to *check grammars for errors*
- In addition to parsing, Bison allows to define semantic actions and give values to grammar symbols
  - for example, evaluate arithmetic expressions as the expression is parsed and/or create a syntax/parse tree for the parse)

Yacc / Bison Input File

%{

Global C declarations

/* This code is copied into the generated source file */

%}

Bison declarations

%%

Grammar rules

%%

Additional C code

/* This code is copied into the generated source file */

Yacc / Bison vs. Lex / Flex

- The file format of Yacc / Bison is similar with Lex / Flex
- The two programs are meant to work together
- Lex / Bison assume a function *yylex()* to be used for tokenization
- *yylex()* can be hand-coded or it can be generated by Lex / Flex
- Yacc / Bison produces the parsing function called *yyparse()*

Yacc / Bison declarations

- %token <TOKEN-NAME>

  declare a token or terminal symbol

- %type <<VAL>> <SYMB0L>

  declare a symbol in the grammar to have the semantic value <VAL>

- %union { ... }

  declare a set of possible values for use in semantic actions (using C syntax)

Yacc / Bison declarations. Example

```
%{
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100
char *g_finalstr;
%}

%token SEMICOLON
%token MULT
%token DIV
%token PLUS
%token MINUS
%token RPAREN
%tokenLPAREN
%token<num> NUMBER

%union {
double num;
car *str;
}

%type  <str>    exprlist
%type  <num> expr
%type  <num> addexpr
%type  <num> multexpr
%type  <num> unaryexpr
%type  <num> simplexpr
```

Grammar Specification

- A grammar rule has the following form:

```
RULENAME:   COMPONENTS  . . .
                [  |  COMPONENTS  . . .  ]
```

- An empty rule is specified by simply giving no components (or use the comment */*empty*/*)
- Specify all terminals in uppercase letters and nonterminals in lowercase
- Terminals can either be specified by *%token* declarations or by writing them as character literals in the grammar

Grammar Specification. Example

```
exprlist:  exprlist expr
    I expr
    ;

expr:  addexpr SEMICOLON
    I SEMICOLON
    ;

addexpr:  addexpr PLUS
    multexpr
    I addexpr MINUS multexpr
    I multexpr
    ;

multexpr:  multexpr MULT
    unaryexpr
    I multexpr DIV unaryexpr
    I unaryexpr
    ;

unaryexpr:  MINUS unaryexpr
    I simplexpr
    ;

simplexpr:  LPAREN expr RPAREN
    I NUMBER
    ;
```

Conflict Resolutions

- A reduce/reduce conflict is resolved by choosing the production listed first
- A shift/reduce conflict is resolved in favor of shift
- A mechanism for assigning precedences and assocoativities to terminals
- Declarations
  - %left, %right,%nonassoc, %prec

Precedence and Associativity

- The precedence and associativity of operators are declared simultaneously (order of appearance)

```
%nonassoc  '<'        /* lowest */
%left  '+'  '-'
%right  '^'           /* highest */
```

- The precedence of a rule is determined by the precedence of its rightmost terminal
- The precedence of a rule can be modified by adding %prec <terminal> to its right end

Example

```
%{
#include  <stdio.h>
%}

%token NUMBER
%left  '+'  '-'
%left  '*'  '/'
%right  UMINUS

%%
```

Example

```
line :  expr  '\n'
        ;
expr:  expr  '+' expr
     |  expr  '-'  expr
     |  expr  '*'  expr
     |  expr  '/'  expr
     |  '-'  expr  %prec  UMINUS
     |  '('  expr  ')'
     |  NUMBER
     ;
```

Defining Semantics

- Defining semantics for rules → define actions the parser takes at each step
- Actions are represented by a statement block after each rule:

```
RULENAME: COMPONENTS ... {C STATEMENTS}
```

- Semantic actions allows the parser to perform various tasks at the same time, without having multiple passes
  - Primarily, they allows the parser to build a parse tree

Defining Semantics

- An important action is the ability to assign a value to a terminal or nonterminal
- Yacc / Bison has special syntax for this:
  - $$ refers to the current rule
  - $n refers to the Nth component, starting at 1
- Example:

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

- If $n is 0 or negative, one can reference symbols further down the stack

Example

```
expr:       addexpr SEMICOLON          { $$ = $1; }
    |       SEMICOLON                  { $$ = 0; }
    ;
addexpr:    addexpr PLUS multexpr      { $$ = $1 + $3; }
    |       addexpr MINUS multexpr     { $$ = $1 - $3; }
    |       multexppr                  { $$ = $1; }
    ;
```

Example

```
multexpr:  multexpr MULT unaryexpr      { $$ = $1 * $3; }
    |      multexpr DIV unaryexpr       { $$ = $1 / $3; }
    |      unaryexpr                    { $$ = $1; }
    ;
unaryexpr:  MINUS unaryexpr             { $$ = -1 * $2; }
    |      simplexpr                    { $$ = $1; }
    ;
simplexpr:  LPAREN expr RPAREN          { $$ = $2; }
    |      NUMBER                       { $$ = $1; }
    ;
```

Building a parse tree

- To create a sytax tree, one would do something like the following:

```
RULE:    C1  C2   ...
        {
                $$ = makeTreeNode ();
                addLeafs ($1,   $2,   ...);
        }
```

Using Yacc / Bison

```
int yyerror (char *s) {
    fprintf (stderr, “%s\n", s) ; return;
}

int main (int argc, char **argv) {
    if (yyparse () == 0) {
        fprintf (stdout, “%\nSuccess !\n") ;
    } else {
        fprintf (stdout, "Failure \n");
    }
    return 0;
}
```