YACC

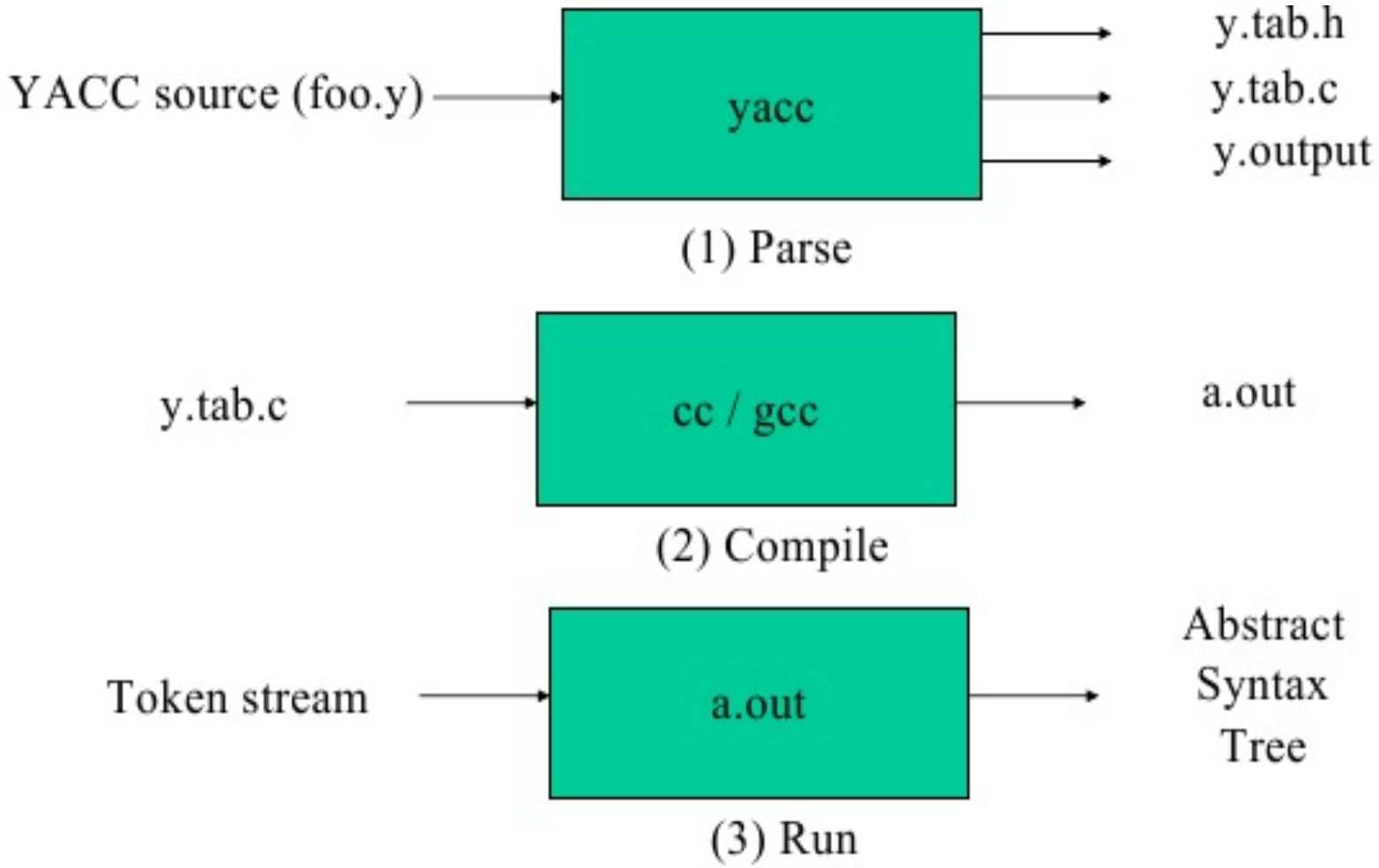
2022 Spring&Summer

5.5 Yacc: LALR(1) PARSING GENERATOR

- Yacc takes a specification file (usually with a .y suffix)
- produces an output file consisting of C source code for the parser (usually in a file called **y.tab.c** or **ytab.c**)
- A Yacc specification file has the basic format

{definitions}
%%
{rules}
%%
{auxiliary routines}

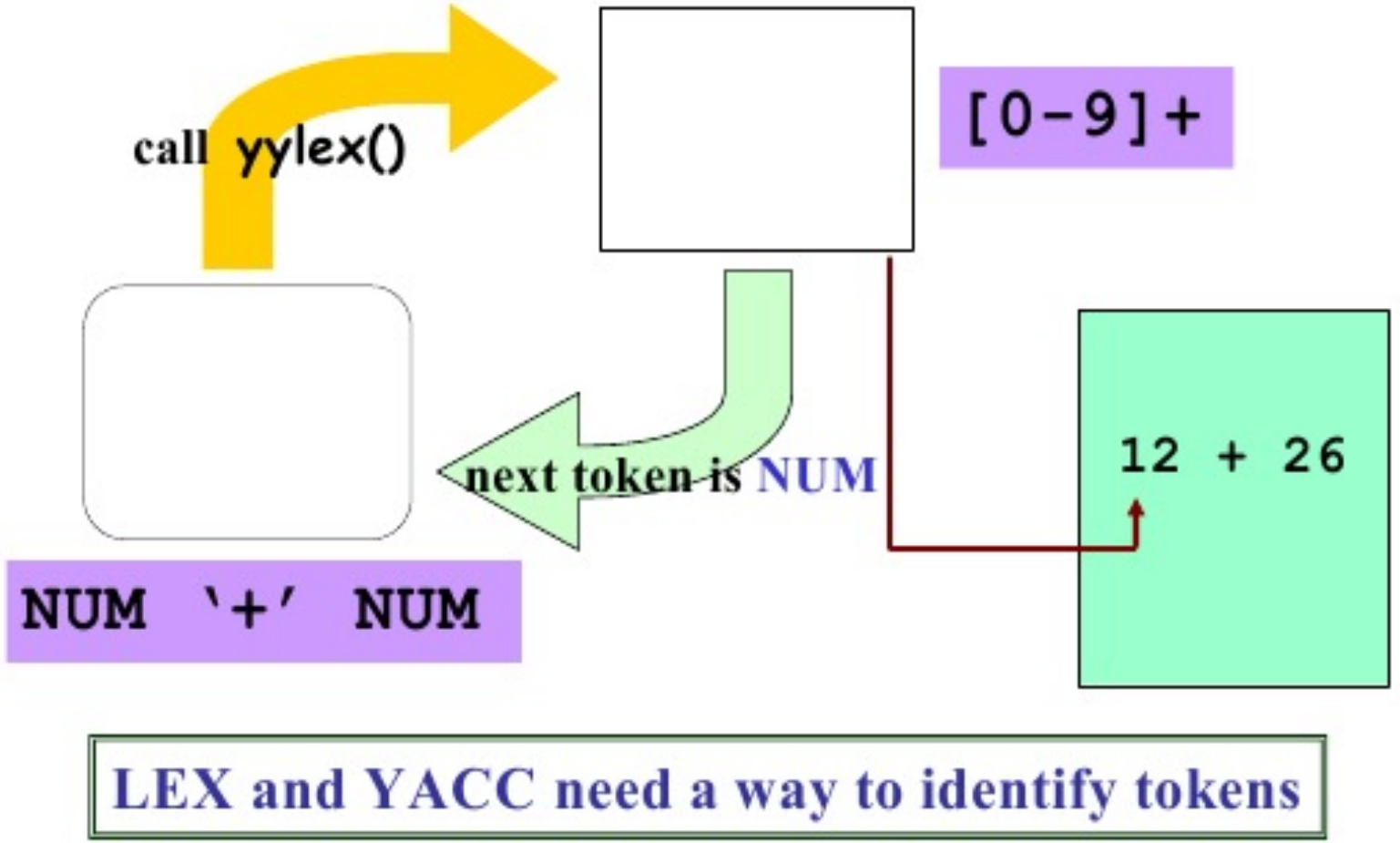
How YACC Works



An YACC File Example

```
%{
#include<stdio.h>
%}
%token NAME NUMBER
%%
statement:  NAME ‘=‘ expression
| expression
;
{ printf(“=%d\n”,$1); }
expression: expression ‘+’ NUMBER
        { $$ = $1 + $3; }
|
Expression ‘-’ NUMBER {$$ = $1 - $3;}
NUMBER
{ $$ = $1; }
;
%%
int yyerror(char *s)
{
fprintf(stderr, "%s\n", s);
return 0;
}
int main(void)
{
yyparse();
return 0;
}
```

Works with LEX



Communication between LEX and YACC

scanner.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{id}        { return ID;}
```

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token  CHAR, FLOAT, ID, INT
%%
```

```
yacc -d xxx.y
   produces
y.tab.h

# define CHAR 258
# define FLOAT 259
# define ID 260
# define INT 261
```

5.5 Yacc: LALR(1) PARSING GENERATOR

exp → *exp addop term* | *term*

addop → + | -

term → *term mulop factor* | *factor*

mulop → *

factor → *(exp)* | *number*

5.5 Yacc: LALR(1) PARSING GENERATOR

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUMBER
%%
command :   exp { printf (“%d\n”,$1);}
                ;           /*allows printing of the result */
exp: exp   ‘+’  term   {$$ = $1 + $3;}
     | exp   ‘-’   term   {$$ = $1 - $3;}
     | term                {$$ = $1;}
     ;
term:  term  ‘*’   factor   {$$ = $1* $3;}
      | factor  {$$ = $1;}
      ;
factor :NUMBER   {$$ = $1;}
        |    ‘(‘  exp  ‘)’   {$$=$2;}
        ;
%%
```

5.5 Yacc: LALR(1) PARSING GENERATOR

```
main ( )
{ return yyparse( );
}
int yylex(void)
{   int c;
    while( ( c = getchar ( ) )== ‘ ’ );
   /*eliminates blanks */
   if ( isdigit(c) ) {
      unget (c,stdin) ;
      scanf (“%d”,&yylval ) ;
      return (NUMBER ) ;
}
if (c== ‘\n’) return 0;
/* makes the parse stop */
return ( c ) ;
}
```

5.5 Yacc: LALR(1) PARSING GENERATOR

```
int yyerror (char * s)

{ fprintf (stderr, “%s\n”,s ) ;

return 0;
}/* allows for printing of an error message */
```

5.5 Yacc: LALR(1) PARSING GENERATOR

Two ways of recognizing tokens:

1. Any character inside single quotes in a grammar rule will be recognized as itself.

2. Symbolic tokens may be declared in a YACC %token declaration .

%token NUMBER

%start symbol (define the start symbol .)

3. Action code is placed at the end of each grammar rule choice, although it is also possible to write embedded actions within a choice.

5.5 Yacc: LALR(1) PARSING GENERATOR

4. Take advantage of Yacc pseudovariables.

- When a grammar rule is recognized, each symbol in the rule possesses a value, which is assumed to be an integer unless changed by the programmer.
- These values are kept on a value stack by Yacc.

the Yacc pseudovariables in the specification file

This data type is always defined in Yacc by the C perprocessor symbol **YYSTYPE**.

#define YYSTYPE double

- inside the brackets %{ . . .%} in the definition section of the Yacc specification file.

5.5 Yacc: LALR(1) PARSING GENERATOR

- Different values for different grammar rules.
  - *exp* → *exp addop term* | *term*
  - *addop* → + | -

- There are two ways to do this.

  (1)Declare the union directly in the Yacc specification using the **%union** Yacc declaration:

  %union { double val; char op;}

5.5 Yacc: LALR(1) PARSING GENERATOR

```
%token NUMBER
%union { double val;
        char op;}
%type <val> exp  term  factor  NUMBER
%type <op> addop  mulop
%%
command : exp {printf("%d\n",$1);}
          ;
exp : exp op term  {  switch ($2);{
                  case '+' : $$=$1+$3; break;
                   case '-' : $$=$1 - $3; break;
                    }
                 }
    | term    {$$ = $1;}
    ;
……
```

5.5 Yacc: LALR(1) PARSING GENERATOR

(2)The second alternative：

Define a new data type in a separate include file define **YYSTYPE** to be this type.

the appropriate values must be constructed by hand in the associated action code.

5.5 Yacc: LALR(1) PARSING GENERATOR

5. All nonterminals achieve their values by such user-supplied actions.

- Tokens may also be assigned values, this is done during the scanning process.
- Yacc assumes that the value of a token is assigned to the variable yylval.

5.5 Yacc: LALR(1) PARSING GENERATOR

6. In the third section, *yyparse* is declared to return an integer value, which is 0 if the parse succeeds, and 1 if it does not.

- The yyparse procedure calls a scanner procedure(*yylex*.)
- Yacc expects the end of input to be signaled by a return of the null value 0 by yylex.
- The *yyerror* procedure prints an error message when an error is encountered during the parse.

5.5 Yacc: LALR(1) PARSING GENERATOR

It is necessary to execute some code prior to the complete recognition of a grammar rule choice during parsing.

decl→ *type var-list*

type→ ***int*** | ***float***

var-list→ *var-list*, ***id*** | ***id***

5.5 Yacc: LALR(1) PARSING GENERATOR

```
decl : type {current_type=$1}
       var-list
      ;
type : INT     {$$=INT_TYPE;}
     | FLOAT      {$$=FLOAT_TYPE; }
     ;
var_list :var_list ‘,’ ID
         {setType(tokenString,current_type); }
         |ID
         {setType(tokenString,current_type); }
          ;
```

5.5 Yacc: LALR(1) PARSING GENERATOR

Yacc interprets an embedded action

```
A : B { /* embedded action */ } C ;
A : B E C ;
E:  {/* embedded action */}
```

5.5 Yacc: LALR(1) PARSING GENERATOR

Yacc has disambiguating rules built into it

Yacc disambiguates by preferring the reduction by the grammar rule listed first in the specification file.

%left '+' '-'

%left '*' (specified in the definitions)

- the operators + and - have the same precedence and are left associative
- the operator * is left associative and has higher precedence than + and -