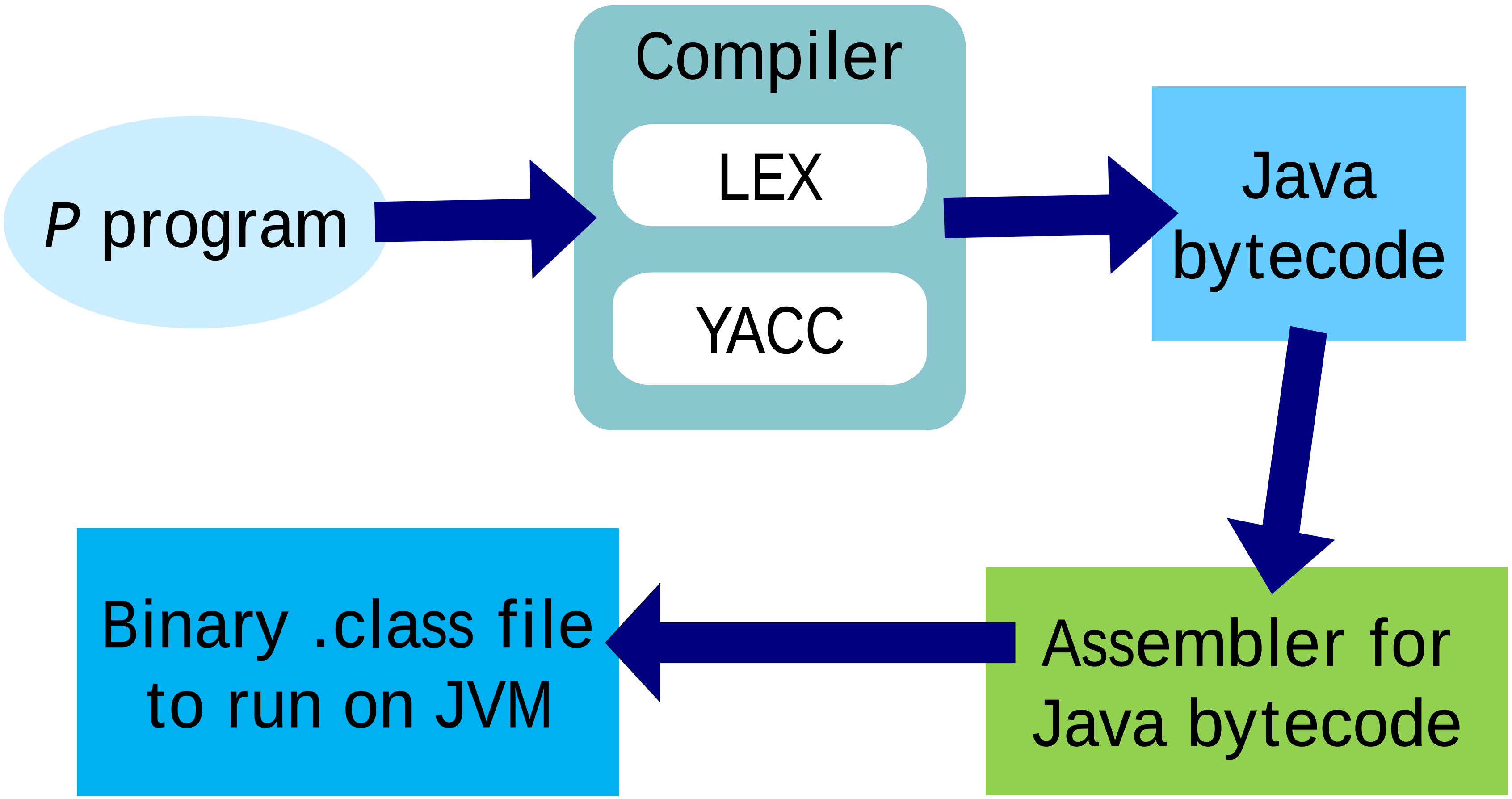
Introduction to Compiler Design

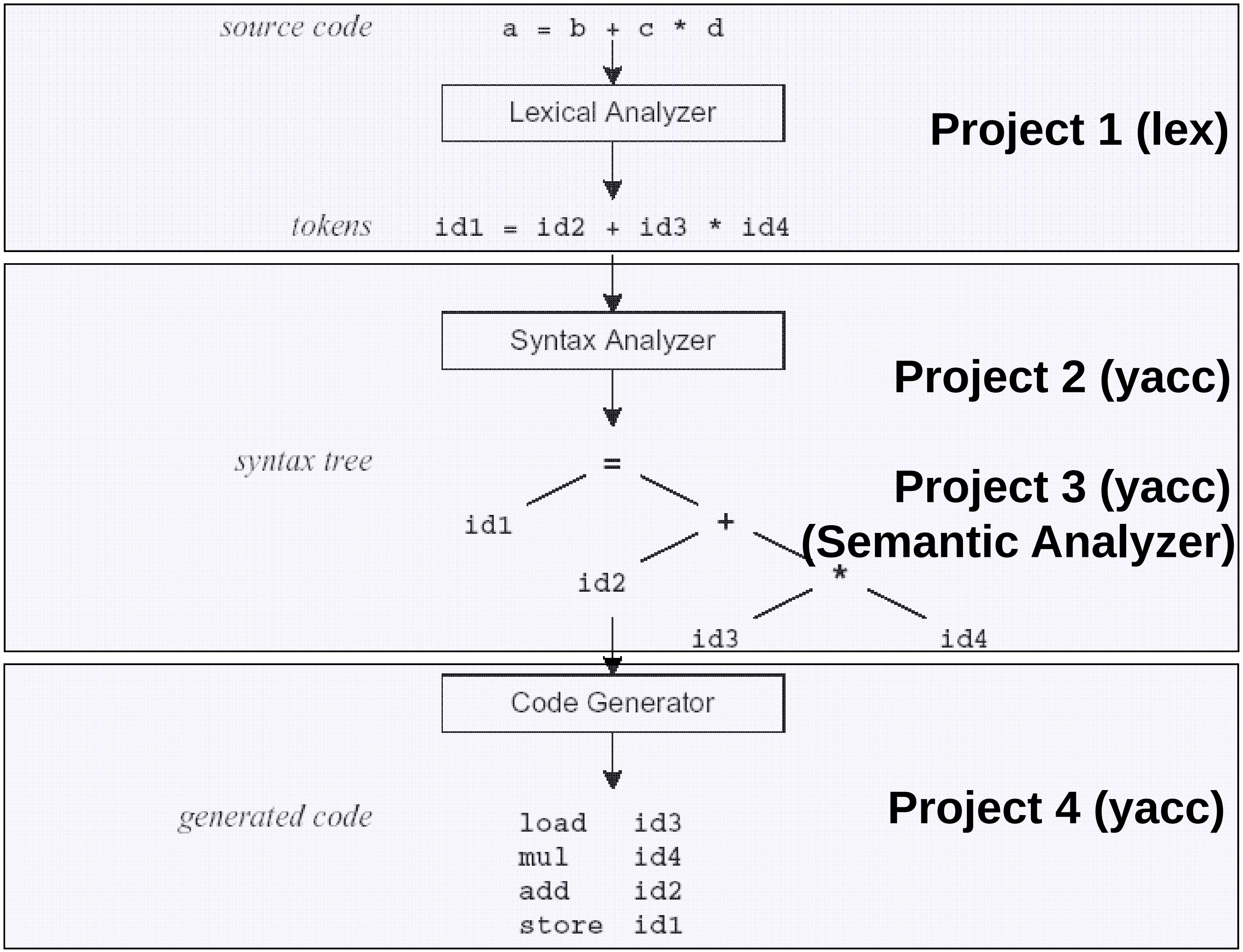
Yacc: The Parser Generator

Professor Yi-Ping You
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

The Goal of Term Project

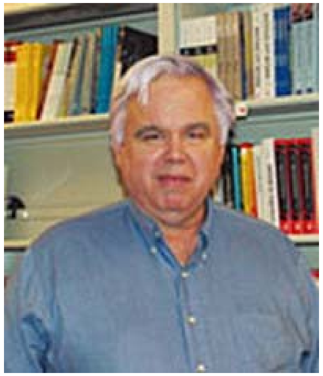


Compilation Flow



What is YACC?

- What is YACC ?
  - Tool which will produce a parser for a given grammar
  - YACC (Yet Another Compiler-Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar
- Original written by Stephen C. Johnson, 1975
- Variants:
  - yacc (AT&T)
  - bison: a yacc replacement (GNU)
  - BSD yacc
  - PCYACC (Abraxas Software)



A YACC Example

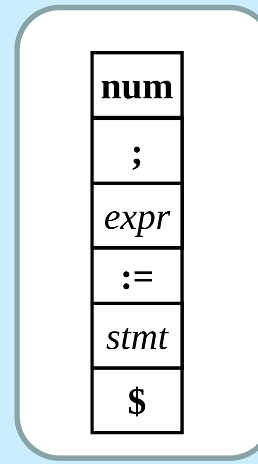
$stmt \rightarrow \textbf{id} := expr \; ;$

$expr \rightarrow expr + \textbf{num} \mid \textbf{num}$

Input: a := 3 + 5;

Output:
reducing to expression from NUMBER…
reducing to expression…
reducing to statement…

```
%token ID ASSIGN PLUS NUMBER SEMI
%%
statement: ID ASSIGN expression SEMI
           {printf("reducing to statement…\n");}
         ;
expression: expression PLUS NUMBER
            { $$ = $1 + $3;
              printf("reducing to expression…\n");
            }
          |  NUMBER
            { $$ = $1;
              printf("reducing to expression from NUMBER…\n");
            }
          ;
%%
```



YACC Source Program

- Yacc program is separated into three sections by %% delimiters
- The general format of Yacc source is

```
{declarations}
%%
{grammar rules}
%%
{user subroutines}
```

{declarations}	(optional)
%% {grammar rules}	(required)
%% {user subroutines}	(optional)

- The absolute minimum Yacc program is

```
%%
 S: ;
```

General Format of YACC Program

```
%{
        C declarations and includes
%}

%token <name1> <name2> …
%start <symbol>
…
```

Declarations

```
%%
```

```
<grammar rule>      <action>
<grammar rule>      <action>
…
```

Rules

```
%%
```

```
User subroutines (C code)
```

Routines

Grammar Rule Section

- Each rule contains LHS and RHS, separated by a **colon** and end by a **semicolon**
  - White spaces or tabs are allowed
- Actions may be associated with rules and are executed when the associated production is reduced
- E.g., $stmt \rightarrow \mathbf{id} := expr \mid expr$

```
stmt: ID ASSIGN expr        {<C code>};
stmt: expr                  {<C code>};
```

```
stmt: ID ASSIGN expr        {<C code>}
    | expr                  {<C code>}
    ;
```

YACC Actions

- Actions are C code
- Actions can include references to attributes associated with terminals and non-terminals in the productions
- Actions may be put inside a rule
  - Action performed when symbol is pushed on stack
  - E.g.,

```
A   : B {<action1>} C {action2};
```

```
ACT: {<action1>};
A   : B ACT C {action2};
```

- Safest (i.e. most predictable) place to put action is at end of rule

Communication between Actions and Parser

- The $ symbol is used to facilitate communication between the actions and the parser
  - The pseudo-variable **$$** presents the value returned by the complete action
  - To obtain the values returned by previous actions and the lexical analyzer, we use the pseudo-variable **$1**, **$2**, ...
  - E.g.,

from lexer   from previous action

```
expr : '(' expr ')' { $$ = $2; };
```

$$ $1 $2 $3

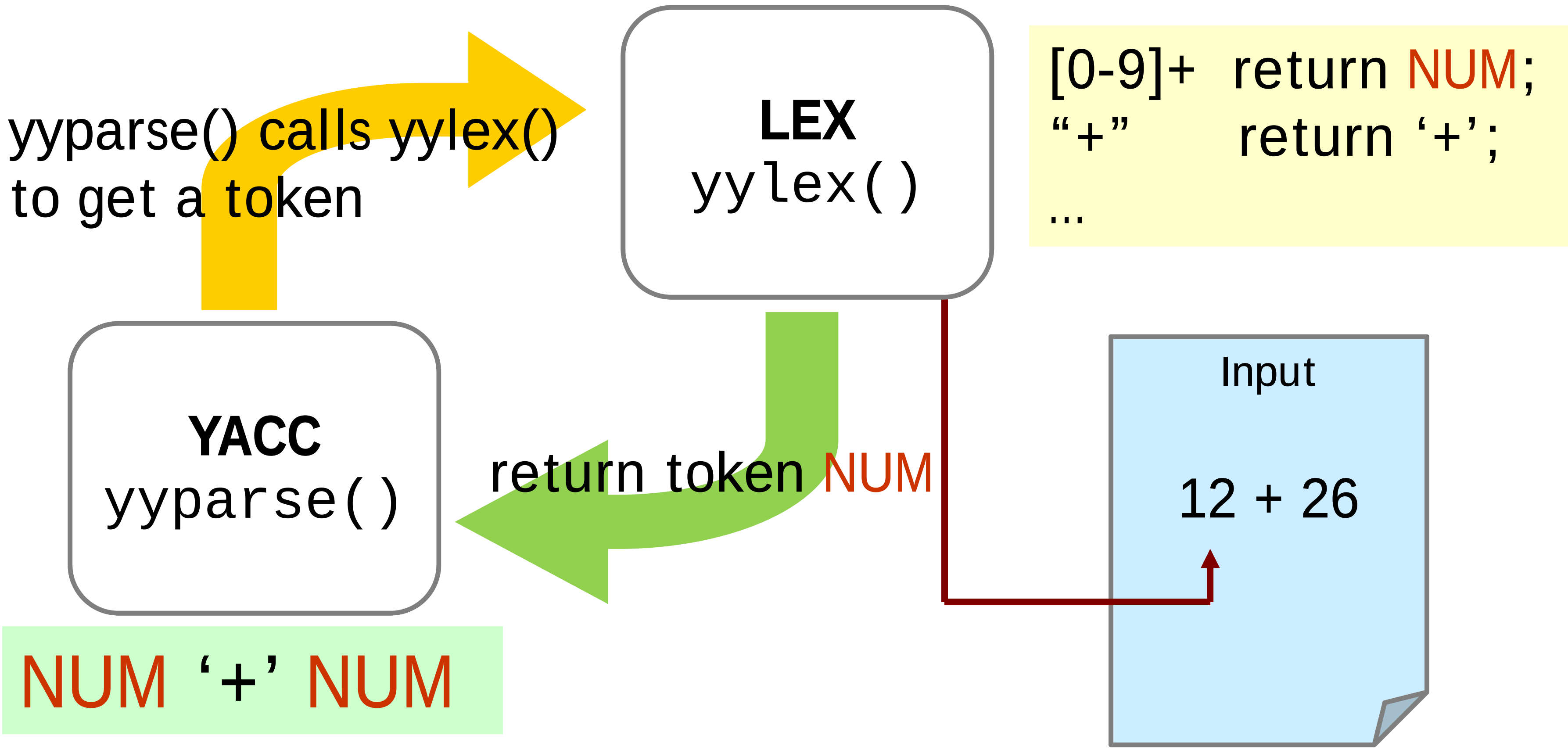
- LHS: `$$`   RHS: `$1 $2 ......`
- Default action: { `$$ = $1;` }

YACC Actions (Cont'd)

- In many applications, output is not done directly by the actions
- A data structure, such as a parse or syntax tree, is constructed in memory
- E.g.,

```
expr : expr '+' expr
        {
        $$ = node('+', $1, $3);
        }
```

How YACC Works with LEX?



In order to communication by the tokens (ex: NUM), an interface is needed between LEX and YACC

Communication between LEX and YACC

- The interface could be a .h file produced by YACC
  - YACC produces `y.tab.h`
  - LEX includes `y.tab.h`

```
int yylex() {
  …
}
```

scanner.l

```
%{
#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

```
…
%token  CHAR, FLOAT, ID, INT
%%
…
```

Declaration Section

yacc -d parser.y

produces **y.tab.h**

The content of **y.tab.h**

```
# define CHAR 257
# define FLOAT 258
# define ID 259
# define INT 260
```

Communication between LEX and YACC (Cont'd)

- `yyparse()` calls `yylex()` when it needs a new token. YACC handles the interface details

In the Lexer:	In the Parser:
`return(TOKEN)`	`%token TOKEN` `TOKEN` used in productions
`return('c')`	`'c'` used in productions

- Every name not defined in the declaration section is assumed to represent a nonterminal symbol
- **`yylval`** is used to return attribute information

yylval Variable

- Used to store the attribute information of a symbol (i.e., a terminal or a nonterminal)
  - The value returned by the lexer (terminal)
    - E.g., in scanner.l

```
[0-9]+ {yylval = atoi(yytext); return NUM;}
```

  - The value returned by actions (nonterminal)
    - E.g., in parser.y

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

- Default data type: integer
- Yacc can also support values of other types including structures
  - Using **%union** in the declaration section

Define the Type of **`yylval`**

- The type of `yylval` is defined by **`%union`**

parser.y

```
%union {
   int value;
   double    dval;
   char*     text;
}
%%
expr: NUM PLUS NUM
     {$$ = $1 + $3;}
```

yacc -d

y.tab.h

```
…
typedef union {
   int         value;
   double      dval;
   char*       text;
} YYSTYPE;
extern YYSTYPE yylval;
```

scanner.l

```
      #include “y.tab.h”
%%
[0-9]+  {yylval.value = atoi(yytext); return NUM;}
[A-z]+  {yylval.text = strdup(yytext);
         return STRING;}
```

Declaration Section

- Includes:
  - Optional C code (%{ … %}) – copied directly into y.tab.c
  - YACC definitions (%token, %start, …) – used to provide additional information
    - %token – interface to lex
    - %start – start symbol
      - By default, start symbol is the LHS of the first grammar rule
    - Others: %left, %right, %nonassoc, %type, %union …

Define Associativities

- **%left** to describe left-associative operators
- **%right** to describe right-associative operators
- The keyword **%nonassoc** is used to describe operators, ~~like < or > in C (Ex: no a < b < c expression in C)~~
- **%prec** changes the precedence level associated with a particular grammar rule
  - **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal

Define Precedences

- All of the tokens on the same line are assumed to have the same precedence level and associativity
- The lines are listed in order of increasing precedence
  - Lowest first

Precedence and Associativity: Examples

```
%left '+' '-'
%left '*' '/'
%%
expr : expr '+' expr {$$ = $1 + $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | '-' expr %prec '*' {$$ = -$2;}
```

Higher precedence

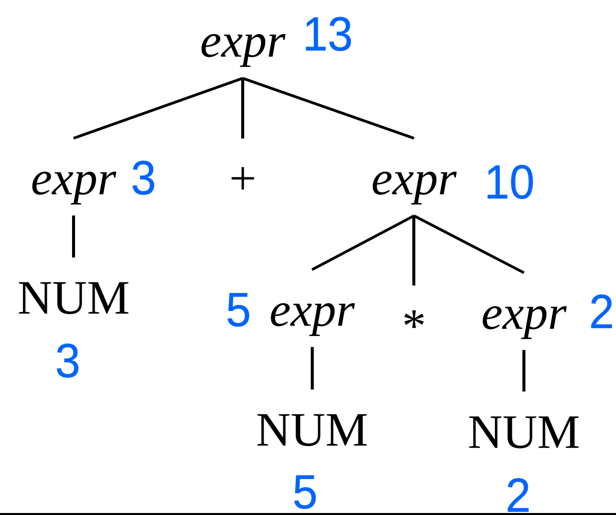
- Arithmetic operators are left-associative
- Unary minus may be given the same strength as multiplication, or even higher while binary minus has a lower strength than multiplication

Implementing a Calculator with Attributes

```
%left '+' '-'
%left '*' '/'
%%
expr : expr '+' expr {$$ = $1 + $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | '-' expr %prec '*' {$$ = -$2;}
     | NUM {$$ = $1;}
```

Higher precedence

- Input: 3 + 5 * 2



Associating Union Member Names

parser.y

```
%union {
   int       value;
   char*     text;
   int       optype;
   int       nodetype;
}
%%
expr: NUM PLUS NUM {$$ = $1 + $3;}
```

- With terminals
  - `%token <value> NUM`
  - `%token <text> ID STRING`
  - `%left <optype> PLUS MINUS`
- With nonterminals
  - `%type <nodetype> expr stmt`

YACC Declaration Summary

- **`%start`**
  - Specify the grammar's start symbol
- **`%union`**
  - Declare the collection of data types that semantic values may have
- **`%token`**
  - Declare a terminal symbol (token type name) with no precedence or associativity specified
- **`%type`**
  - Declare the type of semantic values for a nonterminal symbol

YACC Declaration Summary (Cont'd)

- **`%right`**
  - Declare a terminal symbol (token type name) that is right-associative
- **`%left`**
  - Declare a terminal symbol (token type name) that is left-associative
- **`%nonassoc`**
  - Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error, Ex: x op. y op. z is syntax error)

User Subroutine Section

- You can use your routines in the same ways you use routines in other programming languages
- Two default routines will be provided by the library accessed by a `–ly` argument

```
main() {
  return yyparse();
}
```

```
#include <stdio.h>
yyerror(char *s) {
  (void) fprintf(stderr, “%s\n”, s);
}
```

Error Message

- Error message:
  - Syntax error
  - Compiler should give programmers a good advice
- It is better to track the line number like:

```
int yyerror(char *s) {
  fprintf(stderr, "line %d: %s\n:", lineno, s);
}
```

Notes: Debugging YACC Conflicts

- Sometimes you get shift/reduce errors if you run YACC on an incomplete program
  - Don't stress about these too much UNTIL you are done with the grammar
- If you get shift/reduce or reduce/reduce conflicts, YACC can generate information into a file, called **`y.output`**, for you when YACC is invoked with the –v option
  - `y.output`: the parsing table
- Unless instructed YACC will resolve all conflicts using the following two rules:
  - shift/reduce conflict: choose shift
  - reduce/reduce conflict: choose the conflicting production listed first in the yacc specification

y.output: An Example

```
%token DING DONG DELL
%%
rhyme : sound place;
sound : DING DONG;
place : DELL;
```

yacc -v

y.output

```
state 0
         $accept : . rhyme $end  (0)

          DING  shift 1
          .  error

          rhyme  goto 2
          sound  goto 3

state 1
          sound : DING . DONG  (2)

          DONG  shift 4
          .  error
```

y.output

```
state 2
         $accept : rhyme . $end  (0)

         $end  accept

state 3
         rhyme : sound . place  (1)

         DELL  shift 5
         .  error

         place  goto 6

state 4
         sound : DING DONG .  (2)

         .  reduce 2

state 5
         place : DELL .  (3)

         .  reduce 3

state 6
         rhyme : sound place .  (1)

         .  reduce 1
```

Using YACC with Ambiguous Grammars

- Dangling-else ambiguity
  - shift/reduce conflict: choose shift
    - This rule resolves the conflict arising from the dangling-else ambiguity correctly!
- We can change the default rules applied by Yacc
  - Precedence:
    - Tokens are given precedences in the order in which they appear in yacc's declaration part, lowest first
    - Tokens in the same declaration have the same precedence
  - Associativity:
    - `%left '+' '-'`
    - `%left '*' '/'`
    - `%right '=' '!'`

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Error Recovery

- Error recovery is performed via error productions
- An error production is a production containing the predefined terminal **error**
- After adding an error production,

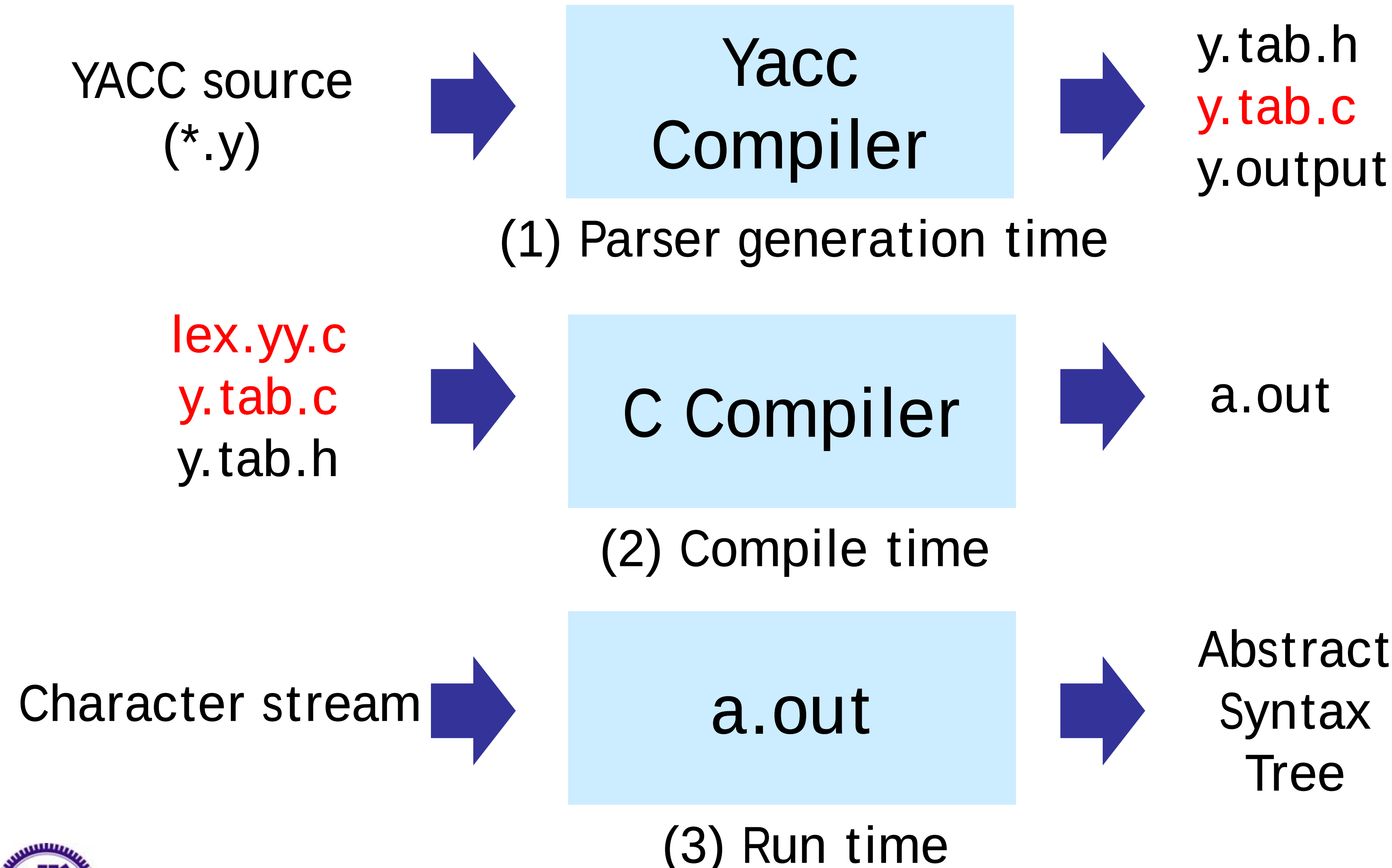
$$A \rightarrow \alpha\, B\, \beta \mid \alpha\, \mathbf{error}\, \beta$$

- on encountering an error in the middle of B, the parser
  - pops symbols from its stack until α,
  - shifts error, and
  - skips input tokens until a token in FIRST(β)

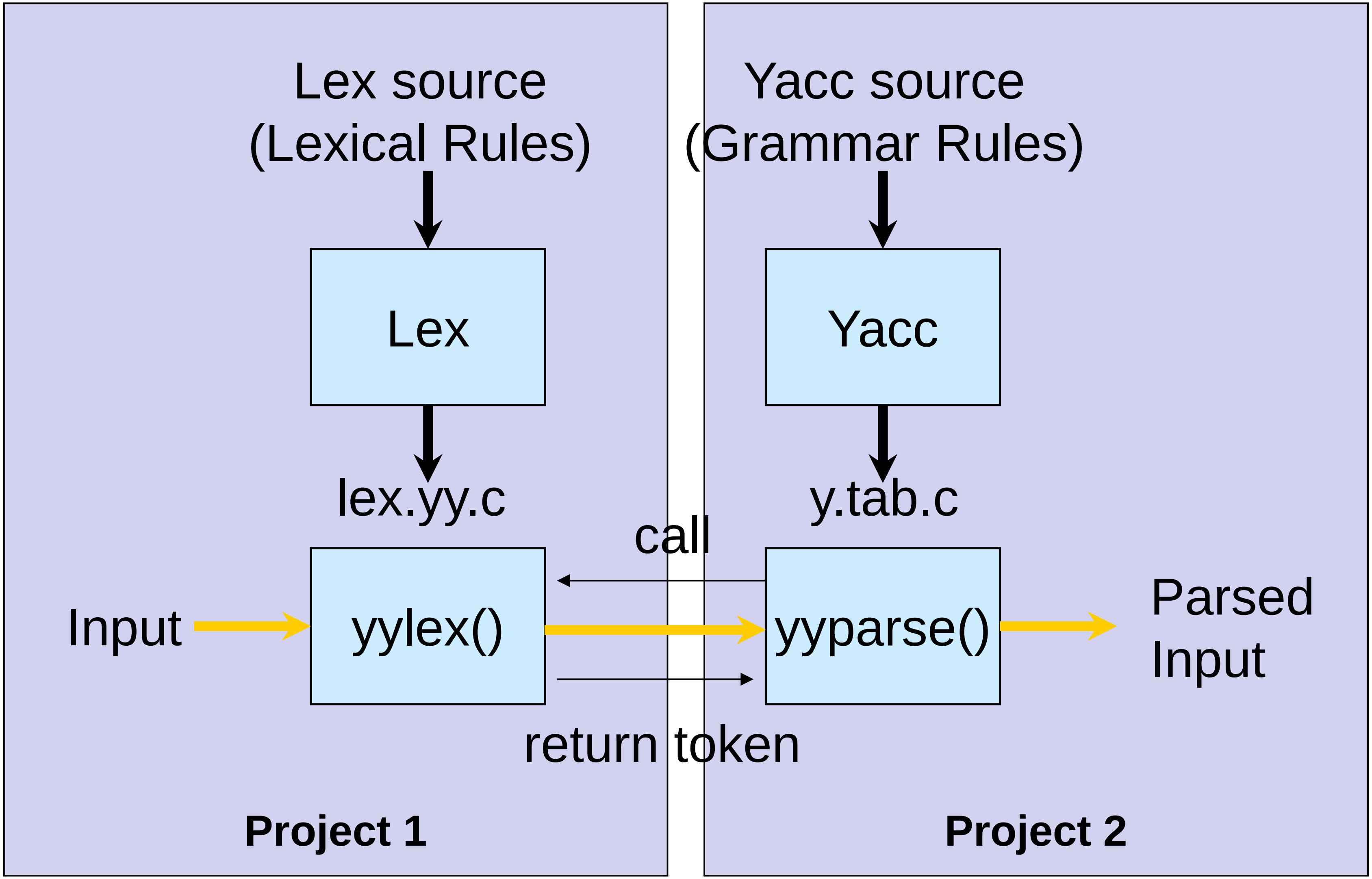
Error Recovery (Cont'd)

- The parser can report a syntax error by calling the function **`yyerror(char *)`**
- The parser will suppress the report of another error message for 3 tokens
- You can resume error report immediately by using the macro **`yyerrok`**
- Error productions are used for major nonterminals

How YACC Works?



Term Project: A *P* Compiler



Run LEX and YACC

- `yacc -d -v parser.y`
  - generates y.tab.c
  - -d: generates y.tab.h
  - -v: generates y.output
- `lex scanner.l`
  - #include “y.tab.h”
  - generates lex.yy.c
- `gcc lex.yy.c y.tab.c –ly –ll`
- `./a.out < example.c`