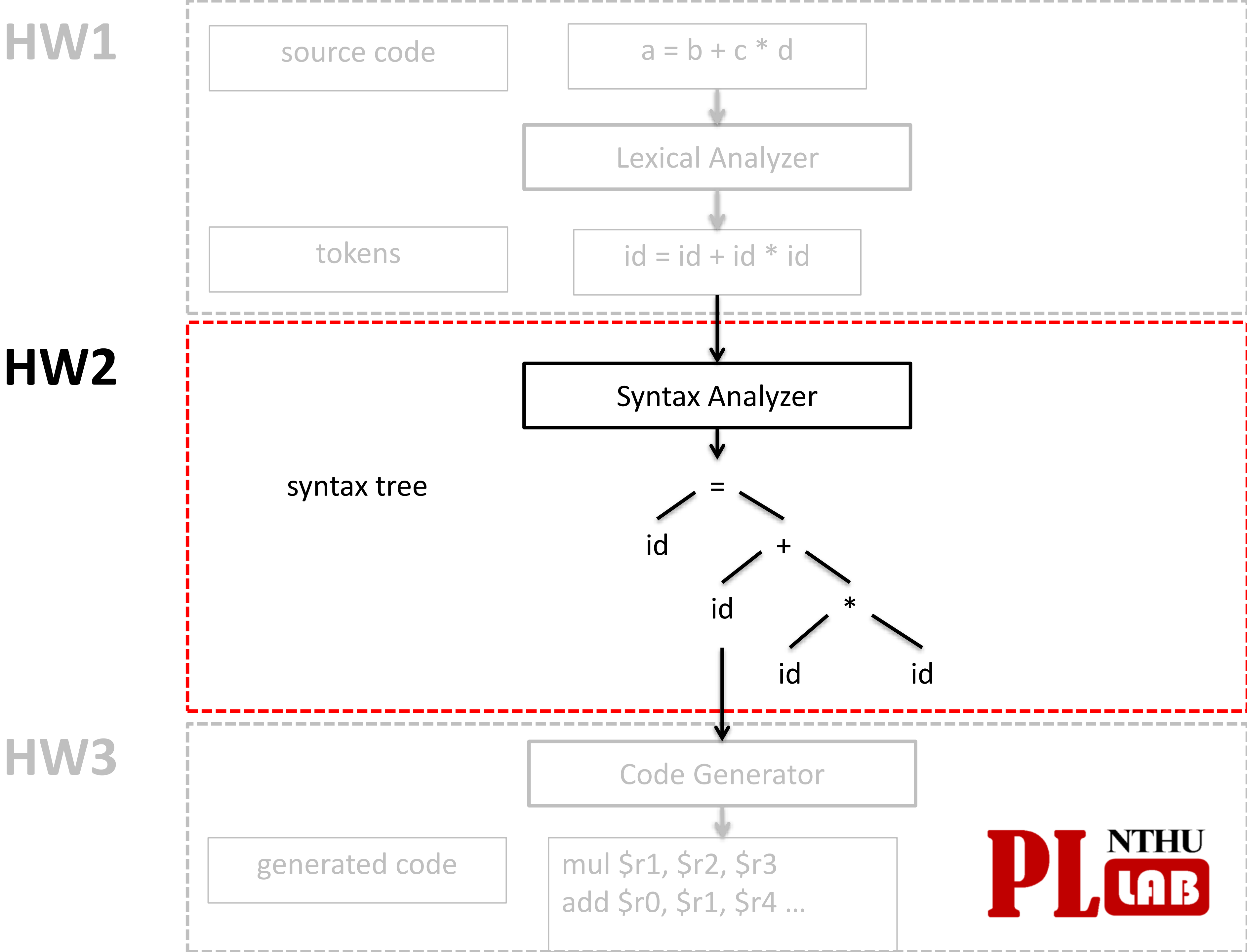
# CS340400 Compiler Design Homework 2

## Deadline

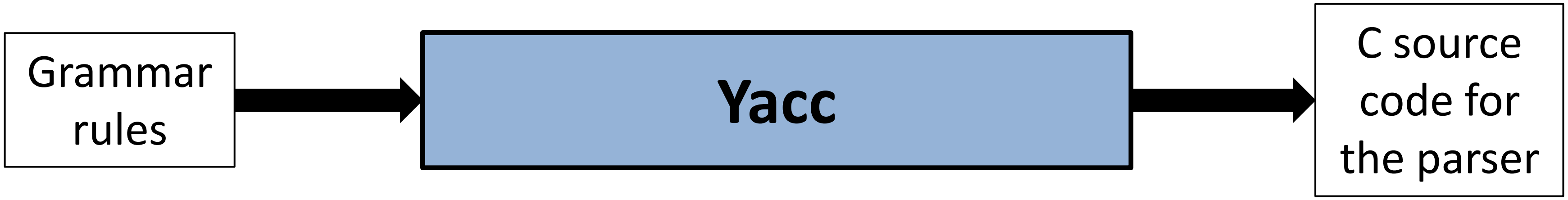
## 2024/06/02 12:00 pm

# Yacc: Yet Another Compiler-Compiler


HW1
source code
a = b + c * d
Lexical Analyzer
tokens
id = id + id * id
HW2
Syntax Analyzer
syntax tree
=
id
+
id
*
id
id
HW3
Code Generator
generated code
mul $r1, $r2, $r3
add $r0, $r1, $r4 ...

# What is Yacc?

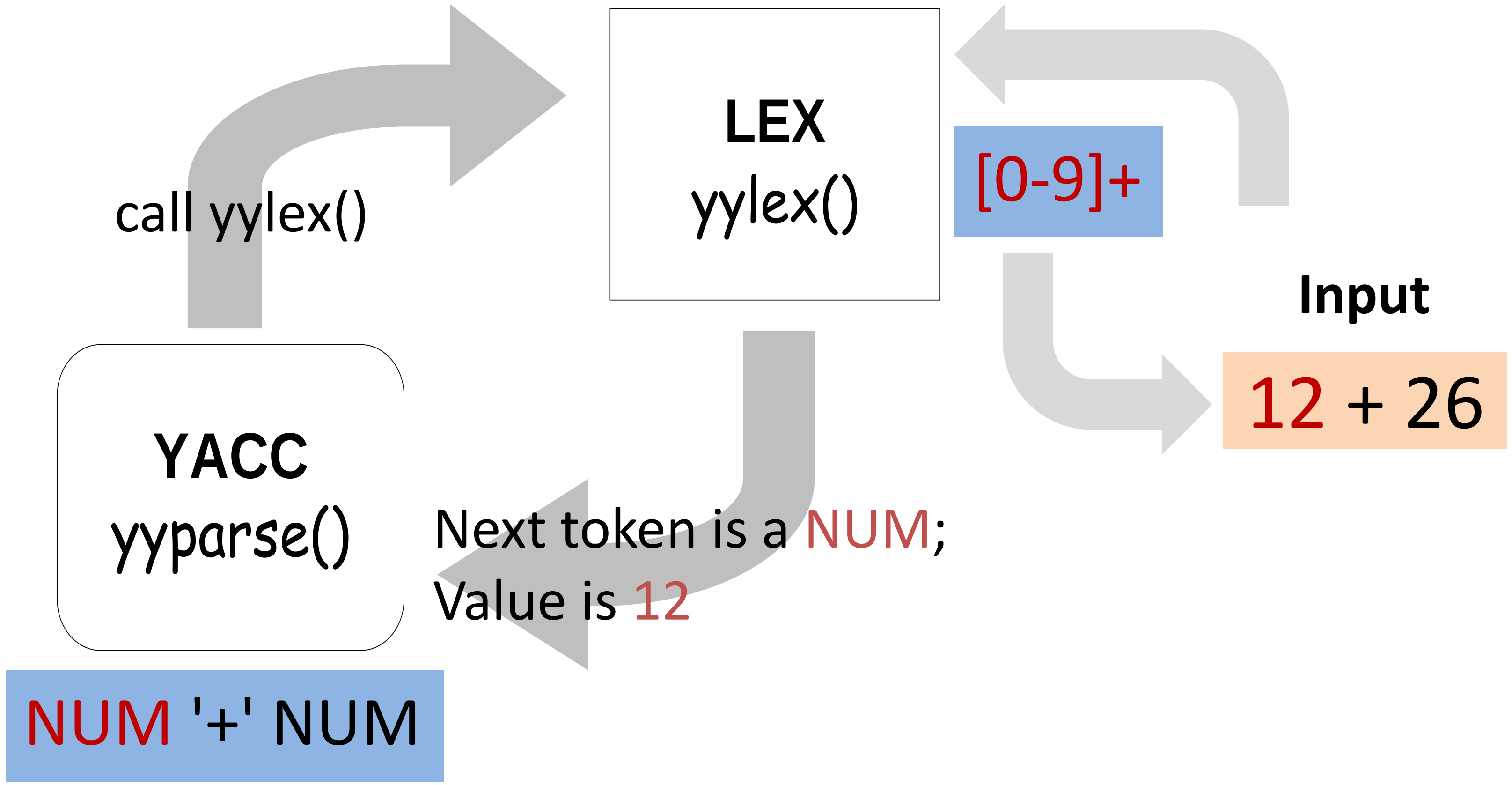
- A tool which can produce a parser with a given grammar
- A program designed to compile a LALR(1) grammar and produce the source code of the syntactic analyzer of the language defined by this grammar



# How to Write Yacc?

```
Declaration Section      →  (optional)
%%
Grammar Section          →  (required)
%%
C Code Section           →  (optional)
```

# How YACC Cooperates with LEX?



# Interface between Lex and Yacc

- The interface is **`y.tab.h`**, which is produced by Yacc.
- How to create `y.tab.h` and use it?
  - `$ yacc -d parser.y`
    - The command will produce `y.tab.h` and `y.tab.c`.
  - Include `y.tab.h` in the Lex program.

scanner

```
%{
…
#include "y.tab.h"
%}

%%
"+"       { return '+';}
[0-9]+    { return NUM;}
…
%%
```

**Terminals**

parser

```
…
%token NUM

%%

expression : expression '+' NUM
           ;
…
```

**Non-terminals**

# yylval

- A symbol in Yacc may carry a value with `yylval`
  - For example, a numeric value 42, or a pointer to a string `"Hello world!"`
- Default type of `yylval` is `int`
  - The type of `yylval` can be overwritten with `%union`
  - E.g.
    - Lex

```
// by default, type of yylval is int
[0-9]+ { yylval = atoi(yytext); return NUM; }
// type of yylval is overwritten with %union
[0-9]+ { yylval.intVal = atoi(yytext); return NUM; }
```

- Yacc: Use `$$, $1, $2,` …… to access values of reduced symbols

# %union

- YYSTYPE is the type defined by `%union` in `y.tab.h`.
- All symbols, include terminal and nonterminal symbols are of YYSTYPE.

```
yacc -d test.y
```

```
// y.tab.h
…
extern YYSTYPE yylval;
```

```
// test.y

%union{
      int intVal;
      double dval;
      struct symbol *sym;
}

%token <intVal> NUM
%%
```

```
// test.l

%{
#include "y.tab.h"
%}
…
%%

[0-9]+ { yylval.intVal = atoi(yytext); return NUM; }
[a-zA-Z]+ { yylval.sym = check(yytext); return VARIABLE; }
```

# How to Write Yacc?

```
Declaration Section
%%
Grammar Section
%%
C Code Section
```

C declaration

```
%{
// header files.
#include <stdio.h>

// Other variables you want to
define.
%}
```

Yacc declaration

```
%token<intVal> ID NUM
```

token: ID is a terminal

The Definition Section
C code will be copied to the top of generated C program.
Define tokens, start symbol, terminal and non-terminal type, association ……

NTHU PL LAB

# How to Write Yacc?

Declaration Section
%%
Grammar Section
%%
C Code Section

```
expression: expression '+' NUM {
    $$ = $1 + $3;
  }
  | expression '-' NUM {
    $$ = $1 - $3;
  }
  | NUM {
    $$ = $1;
  };
```

The grammar section is where to write your own grammar.

non-terminal : grammar_rule_1 { actions_1 }

...

| grammar_rule_n { actions_n } ;

# Grammar Section

Grammar

expr → expr '+' term | term
term → term '*' factor | factor
factor → '(' expr ')' | ID | NUM

Grammar Section in Yacc file

```
expr    : expr '+' term
        | term
        ;
term    : term '*' factor
        | factor
        ;
factor  : '(' expr ')'
        | ID
        | NUM
        ;
```

# Semantic Routines

```
expr : expr '+' term    { C code }
     | term             { C code }
     ;
term : term '*' factor  { C code }
     | factor           { C code }
     ;
factor : '(' expr ')'   { C code }
       | ID
       | NUM
       ;
```

# Semantic Routines with yylval

```
expr : expr '+' term      { $$ = $1 + $3; }
     | term               { $$ = $1; }
     ;
term : term '*' factor    { $$ = $1 * $3; }
     | factor             { $$ = $1; }
     ;
factor : '(' expr ')'     { $$ = $2; }
       | ID
       | NUM
       ;
```

# Symbol Value Numbering

$1

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

# Symbol Value Numbering

```
expr : expr '+' term      { $$ = $1 + $3; }
     | term               { $$ = $1; }
     ;
term : term '*' factor    { $$ = $1 * $3; }
     | factor             { $$ = $1; }
     ;
factor : '(' expr ')'     { $$ = $2; }
       | ID
       | NUM    $2
       ;
```

# Symbol Value Numbering

```
expr : expr '+' term      { $$ = $1 + $3; }
     | term               { $$ = $1; }
     ;
term : term '*' factor    { $$ = $1 * $3; }
     | factor             { $$ = $1; }
     ;
factor : '(' expr ')'     { $$ = $2; }
       | ID
       | NUM        $3
       ;
```

// Default action: { $$ = $1;}

# How to Write Yacc?

Declaration Section
%%
Grammar Section
%%
C Code Section

```
int main(void) {
  yyparse();
  return 0;
}

int yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
  return 0;
}

// Other functions you defined.
```

The C Code Section
will be copied to the bottom of generated C program.

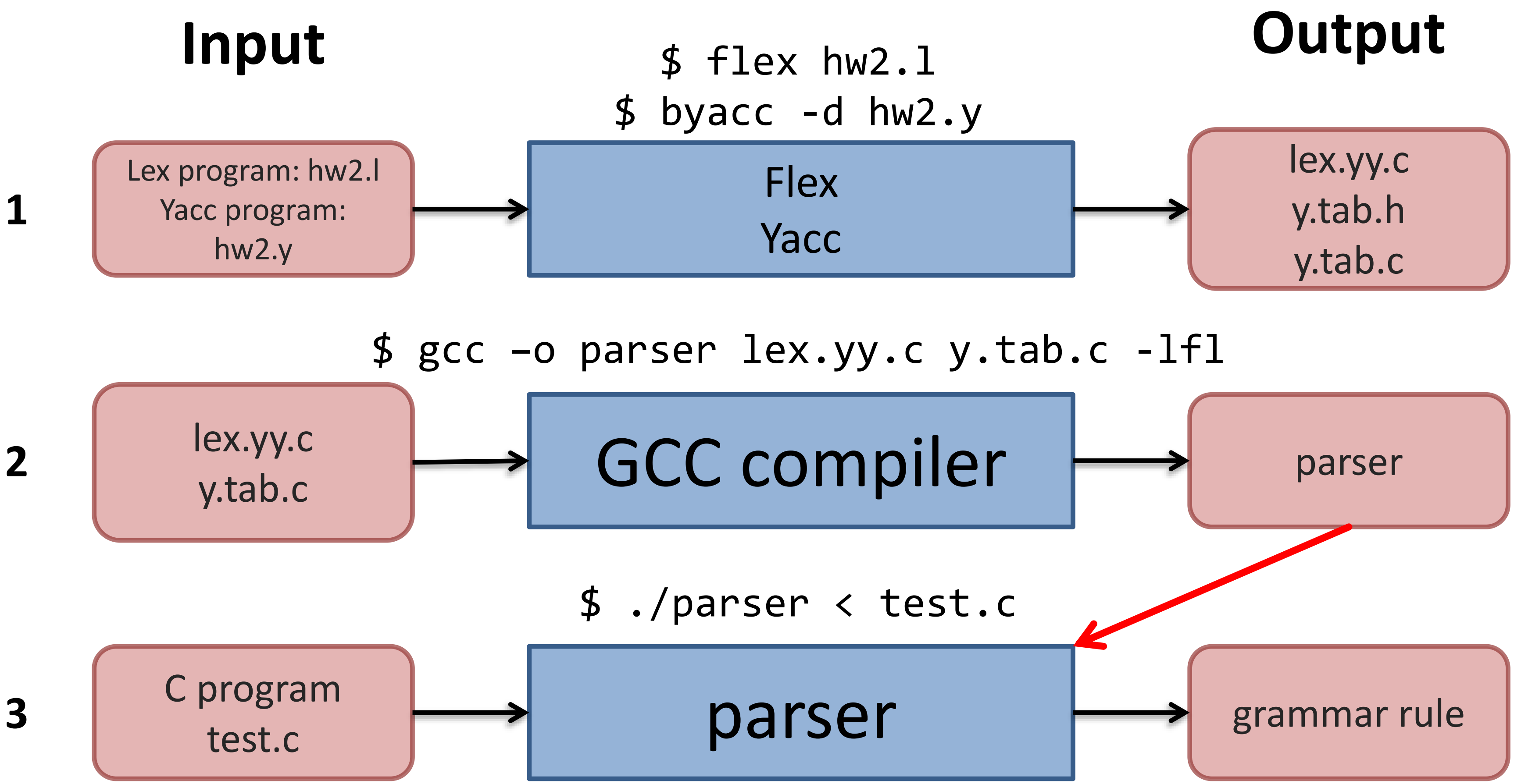
# How to Write Yacc?

```
Declaration Section
%%
Grammar Section
%%
C Code Section
```

```
%{
#include <stdio.h>
%}
%union { int intVal; }
%token<intVal> ID NUM '=' '+' '-'
%type<intVal> statement expression

%%
statement: ID '=' expression
  | expression { printf("= %d\n", $1); };

expression: expression '+' NUM { $$ = $1 +
$3; }
  | expression '-' NUM { $$ = $1 - $3; }
  | NUM { $$ = $1; };

%%
int main(void) {
  yyparse();
  return 0;
}

int yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
  return 0;
}
```

A completed Yacc program.

# How to Use Yacc?



# Precedence / Association

- Consider two cases
  1. `1 - 2 - 3` (association)
  2. `1 - 2 * 3` (precedence)
- With grammar

```
expr: expr ' - ' expr
    | expr ' * ' expr
    | expr ' < ' expr
    | ' ( ' expr ' ) ';
```

- `1 - 2 - 3` is `(1 - 2) - 3` or `1 - (2 - 3)` ?
  - Define `'-'` operator to be left associated
- `1 - 2 * 3` is `1 - (2 * 3)`
  - Define the ' * ' operator to precede the ' - ' operator

# Precedence / Association

- In Yacc definition section:

```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
```

low precedence

high precedence

  - `%left` means left-associative
  - `%right` means right-associative
  - `%nonassoc` means non-associative

```
expr: expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
  | expr '*' expr { $$ = $1 * $3; }
  | expr '/' expr {
    if ($3 == 0) yyerror("divide 0");
    else $$ = $1 / $3;
  }
  | '-' expr %prec UMINUS { $$ = - $2; };
```

# Shift-Reduce Conflicts

- Shift-Reduce Conflicts:
  - Occurs when a grammar is written in a way such that a decision between shifting and reducing cannot be made
  - e.g. Dangling ELSE Ambiguity
- To resolve this conflict, Yacc will choose to shift
- NOT GOOD!! Eliminate them.

# Shift-Reduce Conflict Example

- Grammar:
  S: IF '(' expr ')' S
  | IF '(' expr ')' S ELSE S ;
- Input: `if (e1) if (e2) s1 else s2`
- When parser encounters `else`, it can either
  - Shift (-in `else` first): `else` becomes part of the inner if statement
    - `if (e1) { if (e2) s1 else s2 }`
  - Reduce (S first): `else` becomes part of the outer if statement
    - `if (e1) { if (e2) s1 } else s2`
- From: "A brief yacc tutorial", Saumya Debray, *The University of Arizona, Tucson, AZ 85721.*

NTHU PL LAB

# Reduce-Reduce Conflicts

- Reduce-Reduce Conflicts
```
start: expr | stmt;
expr: CONSTANT;
stmt: CONSTANT;
```
- Yacc resolves reduce-reduce conflicts by using the rule that occurs earlier in the grammar.
- NOT GOOD!! Eliminate them.

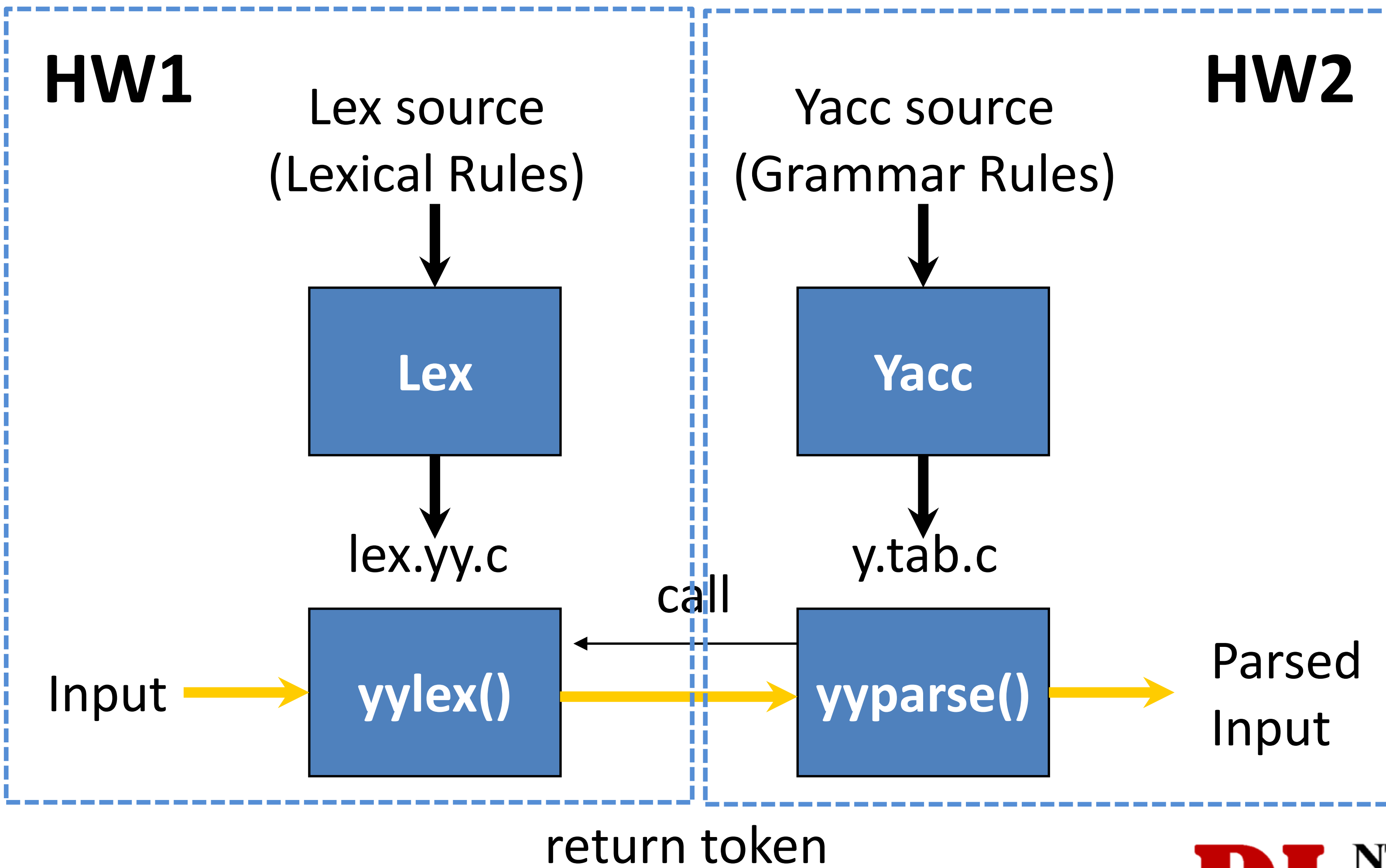
# Handling Conflicts

- General approach
  1. Use `yacc -v` to generate the file `y.output`
  2. Examine `y.output` to find reported conflicts
  3. For each conflict, examine your grammar and `y.output` to figure out why there's the conflict
  4. Transform your grammar to eliminate the conflict

# Yacc-Predefined Declaration

| Name | Function |
|---|---|
| `%start` | Specify the start symbol of grammar. |
| `%union` | Declare the collection of data types that semantic values may have. |
| `%token` | Declare a terminal symbol (token name) with no precedence or associativity specified. |
| `%type` | Declare the type of semantic values for a nonterminal symbol. |
| `%right` | Declare a terminal symbol (token name) that is right-associative. |
| `%left` | Declare a terminal symbol (token name) that is left-associative. |
| `%nonassoc` | Declare a terminal symbol (token name) that is non-associative.<br>Using it in a way that would be associative is a syntax error,<br>Ex: **x operand y operand z** has a syntactic error. |

# Lex with Yacc



# Lex and Yacc

- Rewrite HW 1 Lex to interface with Yacc
  1. `#include "y.tab.h"`
     - `y.tab.h` defines terminal symbols
  2. Remove main function
     - The only main function is in the Yacc file
  3. Set `yylval` in your lex actions
     - So that you can use `$1`, `$2`, … in your Yacc file
  4. Return token or character in your lex actions
     - So that your Yacc knows what kind of token is extracted by Lex

# Homework 2 - Requirements

# Top-level Program Ingredients

- Global Variable Declarations
- Function Declarations
- Function Definitions

# Implement: Scalar Declaration

- ``type idents;''
  - ``type'' can be either
    - ``[const] [signed|unsigned] [long long|long|short] int''
    - ``[const] [signed|unsigned] (long long)|long|short|char''
    - ``[const] signed|unsigned|float|double|void''
    - ``const''
  - ``idents'' consists of 1 or more ``ident'' separated by commas, and each ``ident'' can be either a scalar or a single-level pointer scalar
    - `int a, *b, c;`
  - ``ident'' in ``idents'' can be initialized with ``ident = expr''
    - `int a = 123;`

# Implement: Array Declaration

- ``type arrays;''
  - ``arrays'' consists of one or more ``ident[expr]\[[expr]...\]''
  - ``ident[expr]\[[expr]...\]'' can be initialized with: ``ident[expr]\[[expr]...\] = arr_content''
  - ``arr_content'' format: `{' 1 or more ``expr'' / ``arr_content'' separated by commas `}'
  - E.g.
    - `int a[1][3];`
    - `float a[1], b[1 + 1][3] = {{0, 1, 2}, {3, 4, 5}};`

# Implement: Function Declaration

- ``type ident(parameters);" or ``type * ident(parameters);"
- ``parameters" consists of 0 or more parameters in the form of ``type ident" separated by commas
  - Only support scalar/single-level pointer parameters
- Parentheses are required even if there's no parameter

# Implement: Function Definition

- ``type ident(parameters) compound_stmt" or ``type * ident(parameters) compound_stmt"
- Functions are global and may not be nested within other functions
- ``compound_stmt" refers to compound statements defined in later pages

# Implement: Expression (``expr")

- Parentheses dictate a new precedence sequence for the enclosed ``expr"
- Support ``expr"s constructed by other ``expr"s with the following operators (some of those implemented in HW1)
  - + - * / % ++ -- < <= > >= == != = && || ! ~ ^ & | >> << [ ] ( )
    - Includes (post / pre-fix) (``++" / ``--"), unary (`+' / `-'), function invocation `(params...)`, array subscription, dereference (`*'), address-of (`&'), type-casting (``(type)", including single-level pointer types)
- Also includes
  - ``variable": ``ident" or ``ident[expr]\[[expr]...\]"
  - ``literal": single signless integer / signless floating-point number / char / string literal
  - ``NULL": Equals to integer ``0"
- For precedence and associativity, please refer to <https://en.cppreference.com/w/c/language/operator_precedence>

# Implement: Statement (``stmt")

- Expression Statement: ``expr;"
- IF / IF-ELSE Statement
- SWITCH Statement
- WHILE Statement
- FOR Statement
- RETURN, BREAK, CONTINUE Statement
- Compound Statement

# IF / IF-ELSE Statement

- ``if (expr) compound_stmt"
- ``if (expr) compound_stmt else compound_stmt"
- No ELSE-IF

# SWITCH Statement

- ``switch (expr) { switch_clauses }"
  - ``switch_clauses" consists of 0 or more ``switch_clause" seperated by space / tab / newline / nothing
  - ``switch_clause" is in the form of
    - ``case expr:" 0 or more ``stmt"
    - ``default:" 0 or more ``stmt"

# WHILE Statement

- ``while (expr) stmt''
- ``do stmt while (expr);''

# FOR Statement

- ``for (\[expr\] ; \[expr\] ; \[expr\]) stmt"

# RETURN, BREAK, CONTINUE Statement

- ``return expr;" or ``return;"
- ``break;"
- ``continue;"

# Compound Statement

- `{' 0 or more ``stmt''s / ``var_declaration''s `}'
- ``var_declaration'' refers to variable declarations requiring implementations in the previous pages

# Output Format

- Print the syntax tree to stdout
  - <scalar_decl>``scalar_declaration"</scalar_decl>
  - <array_decl>``array_declaration"</array_decl>
  - <func_decl>``function_declaration"</func_decl>
  - <func_def>``function_definition"</func_def>
  - <expr>``expr"</expr>
  - <stmt>``stmt"</stmt>
- In each tag, strip away all whitespaces ([ \t\n]), except those in char / string literals
- Literals are canonicalized
  - Integer Literals: `atoi` then printf with `"%d"`
  - Double Literals: `atof` then printf with `"%f"`
  - Char Literals: No Change from Input. Keep the quotes.
  - String Literals: No Change from Input. Keep the quotes.
- There should be no raw newline in the output
- Follow `golden_parser` in the case of ambiguity

# Formatting Output for Debug Purpose

- Our output follows the XML Format
- One can format it using an arbitrary XML formatter for debugging
  – $ tidy –xml –i –q input.txt
  – (p.s. This one looks good: https://jsonformatter.org/xml-formatter
- Use formatter only when you know there are something wrong in your output

# yyerror()

- Called whenever an error is encountered during parsing
  - It must be supplied by the Yacc user
- Though there would be no syntax error in the input, one can supply the following:

```c
void yyerror(char * msg) {
  fprintf(stderr, "Error at line %d: %s\n", lineNo, curLine)
  exit(1);
}
```

# Report

- For students who **cannot finish** the homework
  - Explain the Lex-Yacc interaction
  - Describe your understanding on the difficulties you faced
  - Describe how you tried to overcome those difficulties

# Grading Policies

- Any Yacc Conflict or Compiler Warning: -20 points
- Late Submission: -10 points/day
- Executable, but not complying to specifications: 20% off your original score if you apply for a manual review (Reviews are not guaranteed to be accepted.)
- Non-executable: A flat grade of 40 points if you turn in your codes and report (late submission penalty applies)
- **Cheating: You will receive zero credit!**

# Grading Policies

- Scalar Declarations without initialization: +10 pts
- Array Declarations without initialization: +10 pts
- Function Declarations: +10 pts
- Expressions (arithematics, ++/--, unary +/-, (expr), function invocation, array subscription, dereference, address-of, type-casting): +10 pts
- Expressions (arithematics, comparisons, logical operations): +10 pts
- Expressions (all operations): +10 pts
- Full Implementation of Variable Declarations: +10 pts
  - Scalar / Array Declarations with initialization
- Statements: +20 pts
- Function Definitions: +10 pts
- Note: There's dependency between these items, so exact grading is not possible.

# Submission

- Source code
  - Upload to eeclass
    - The revised Lex source code of your lex scanner named \`scanner.l\`
    - Your Yacc parser source code named \`parser.y’\`
    - A \`makefile\` for TAs to compile your code
      - We'll \`make\` in a directory, so make sure you use relative paths in your makefile
  - The compiled output must be named \`parser\` and marked as an executable
- Report (if you can’t turn in a working executable)
  - Upload your code and report in PDF format to eeclass

# How we run and test your program

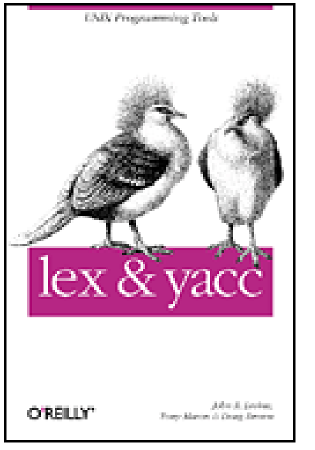
- How we compile
  - Copy your ‘scanner.l’, ‘parser.y’, ‘makefile’ in to a folder
  - Execute command ‘make’ to compile an executable ‘parser’
  - $ ./parser < input.txt > output.txt
- How we test
  - $ diff output.txt golden_answer.txt

# Before you submit

- Make sure your makefile work well on the server
- Compare your output with golden_parser output
- Use 'diff' command to compare outputs
  - $ diff my_parser.txt golden_parser.txt

# Reference

- **lex & yacc**
  - by John R.Levine, Tony Mason & Doug Brown
  - O′ Reilly
  - ISBN: 1-56592-000-7




- **Mastering Regular Expressions**
  - by Jeffrey E.F. Friedl
  - O′ Reilly
  - ISBN: 1-56592-257-3