Yacc - Yet Another Compiler-Compiler

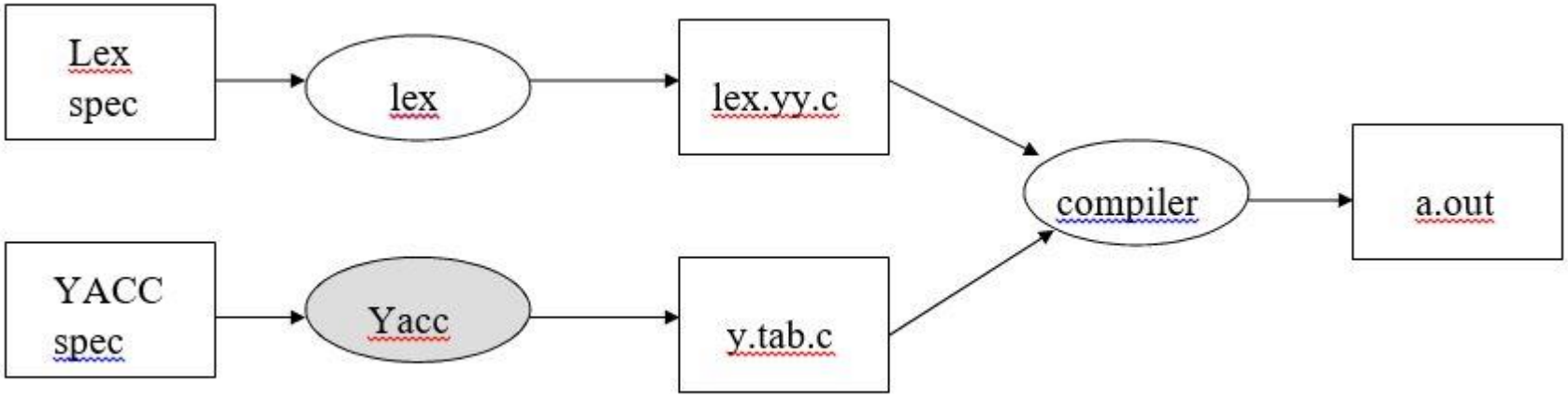
Lex v.s. Yacc

- Lex
  - Lex generates C code for a lexical analyzer, or scanner
  - Lex uses patterns that match strings in the input and converts the strings to tokens
- Yacc
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Introduction

- 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.

How YACC Works



Works with Lex

LEX
yylex()

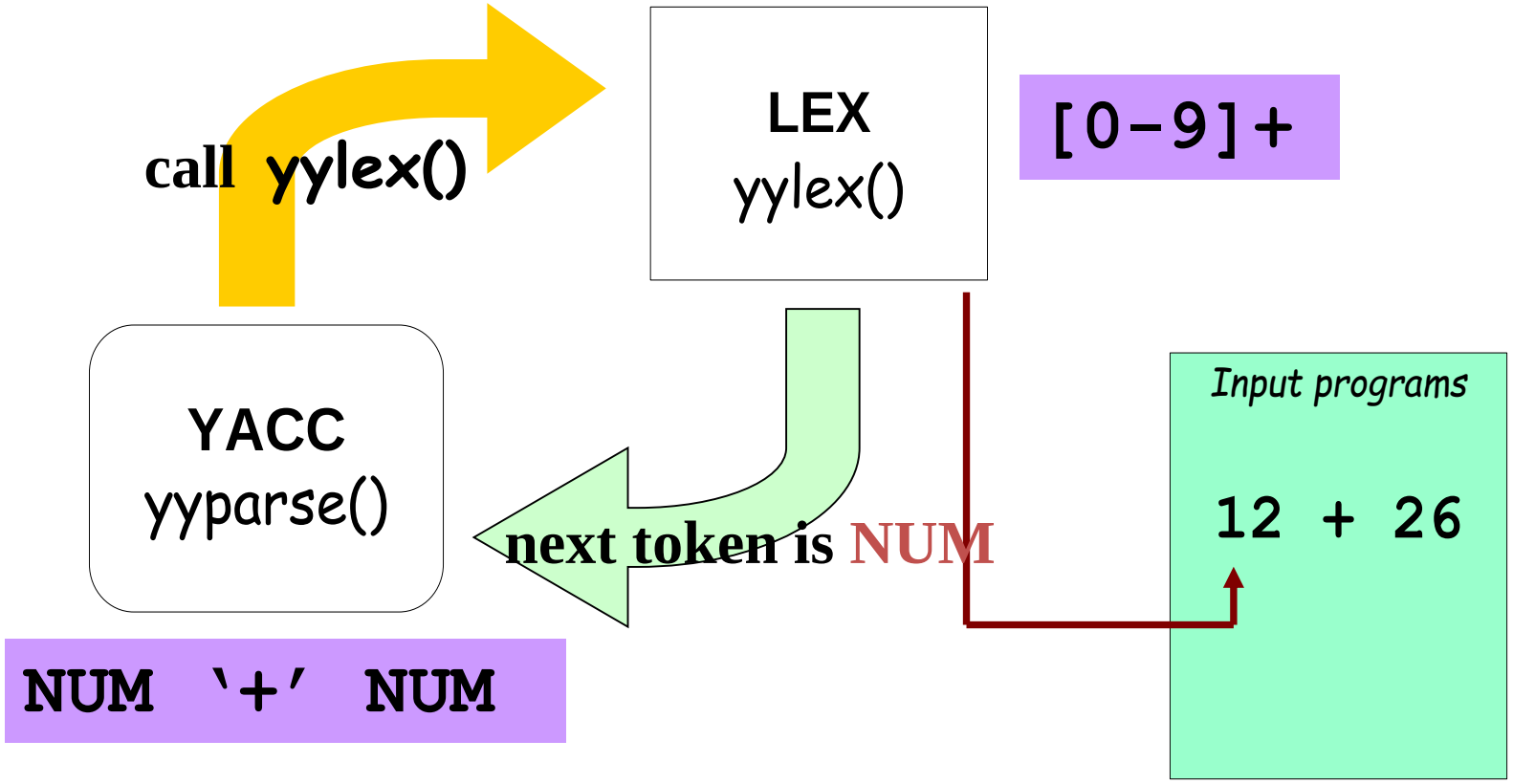
YACC
yyparse()

How to work ?

Input programs

```
12 + 26
```

Works with Lex



Yacc Specification

Declarations

%%

Translation rules

%%

Supporting C/C++ code

Similar structure to Lex

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;
}
```

YACC File Format

%{

C declarations

%}

yacc declarations

%%

Grammar rules

%%

Additional C code

– **Comments enclosed in /* ... */ may appear in any of the sections.**

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

It is a terminal

由 expr 開始parse

Start Symbol

- The first non-terminal specified in the grammar specification section.
- To overwrite it with %start declaraction.

```
%start  non-terminal
```

Rules Section

- This section defines grammar
- Example

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

Rules Section

- Normally written like this
- Example:

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

The Position of Rules

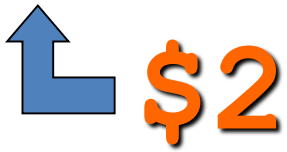
```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

The Position of Rules


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

The Position of Rules

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



The Position of Rules

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

$3

Default: $$ = $1;

Communication between LEX and YACC

- *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

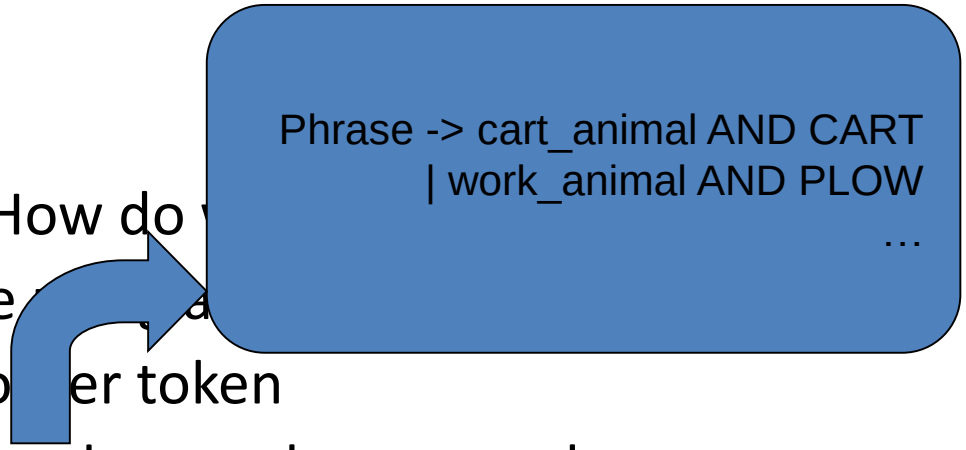
- *yylval* is used to return attribute information

Building YACC parsers

- In input.l spec, need to `#include "input.tab.h"`
- ```
  flex input.l
  bison -d input.y
  gcc input.tab.c lex.yy.c
  ```
  ```

YACC

- Rules may be recursive
- Rules may be ambiguous*
- Uses bottom up Shift/Reduce parsing
  - Get a token
  - Push onto stack
  - Can it reduced (How do
    - If yes: Reduce
    - If no: Get ano er token
- Yacc cannot look ahead more than one token



Yacc Example

- Taken from Lex & Yacc
- Simple calculator

```
a = 4 + 6
a
a=10
b = 7
c = a + b
c
c = 17
$
```

Grammar

```
%token    NUMBER CR
%%
lines       : lines       line
            | line
            ;
line        :     expr      CR                  {printf("Value = %d",$1); }
            ;
expr        :       expr '+' term                 { $$  =  $1 +    $3; }
            |       term                          { $$  =  $1;     /* default – can          omit */}
            ;
term        :       term '*' factor               { $$  =  $1 * $3; }
            |       factor
            ;
factor      :       '(' expr ')'                  { $$  =  $2; }
            |       NUMBER
            ;
%%
```

Scanner

```
%%
\+      {return('+'); }
\*      {return('*'); }
\(      {return('('); }
\)      {return(')'); }
[0-9]+  {yylval = atoi(yytext); return(NUMBER); }
[\n]    {return(CR);}
[ \t]       ;
%%
```

scanner.l

YACC Command

- Yacc (AT&T)
  - yacc –d *xxx.y*
- Bison (GNU)
  - bison –d**y** *xxx.y*

Precedence / Association

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

(1) 1 – 2 - 3

(2) 1 – 2 * 3

1. 1-2-3 = **(1-2)-3**? **or** 1-(2-3)?

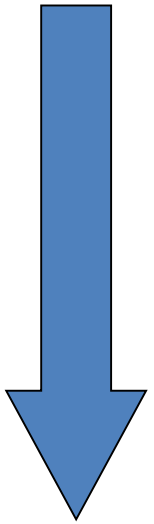
   Define '-' operator is left-association.

2. 1-2*3 = 1-(2*3)

   Define "*" operator is precedent to "-" operator

Precedence / Association

```
%right '='
%left  '<' '>' NE LE GE
%left  '+' '-'
%left  '*' '/'
```



highest precedence

Shift/Reduce Conflicts

- **shift/reduce conflict**
  - occurs when a grammar is written in such a way that a decision between shifting and reducing can not be made.
  - ex: IF-ELSE ambigious.
- To resolve this conflict, yacc will choose to shift.

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

`%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)

Reference Books

- lex & yacc, 2nd Edition
  - 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