Compiler Design

LECTURE-20

Language processing:
Introduction to compiler construction

About this course

- This part will address compilers for programming languages
- Depth-first approach
  - Instead of covering all compiler aspects very briefly, we focus on particular compiler stages
  - Focus: optimization and compiler back issues
- This course is complementary to the compiler course at the VU
- Grading: (heavy) practical assignment and one or two take-home assignments

About this course (cont'd)

- Book
  - Recommended, not compulsory: Seti, Aho and Ullman,"Compilers Principles, Techniques and Tools" (the Dragon book)
  - Old book, but still more than sufficient
  - Copies of relevant chapters can be found in the library
- Sheets are available at the website
- Idem for practical/take-home assignments, deadlines, etc.

Topics

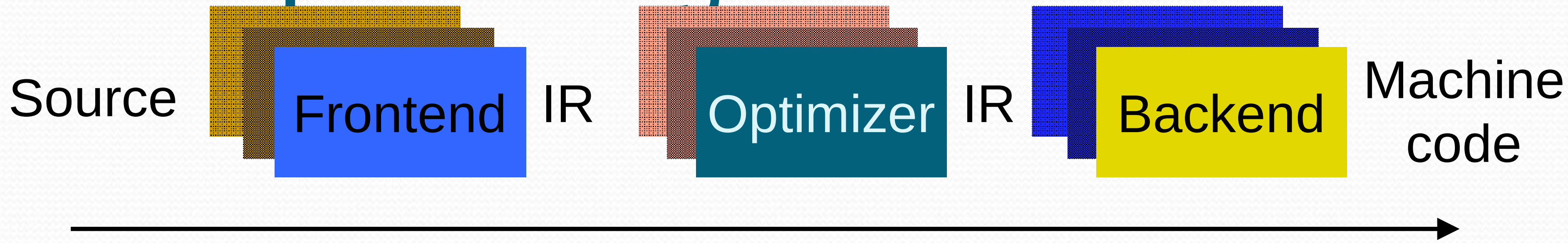
- Compiler introduction
  - General organization
- Scanning & parsing
  - From a practical viewpoint: LEX and YACC
- Intermediate formats
- Optimization: techniques and algorithms
  - Local/peephole optimizations
  - Global and loop optimizations
  - Recognizing loops
  - Dataflow analysis
  - Alias analysis

Topics (cont'd)

- Code generation
  - Instruction selection
  - Register allocation
  - Instruction scheduling: improving ILP
- Source-level optimizations
  - Optimizations for cache behavior

Compilers: general organization

Compilers: organization

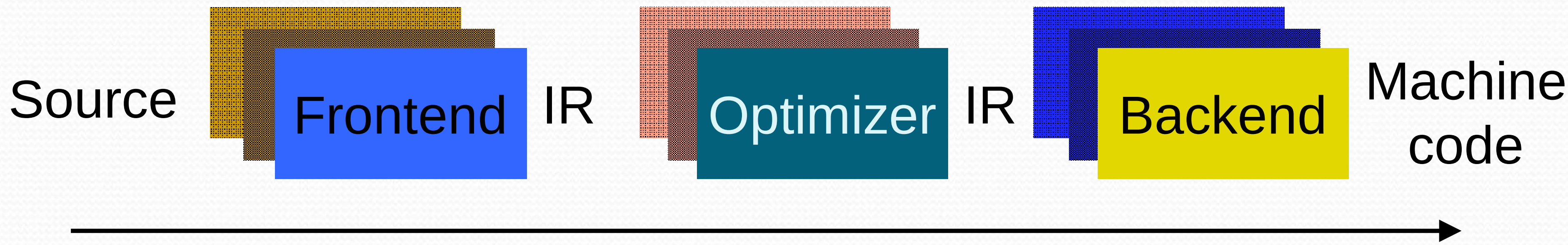


- Frontend
  - Dependent on source language
  - Lexical analysis
  - Parsing
  - Semantic analysis (e.g., type checking)

Compilers: organization (cont'd)

- Optimizer
  - Independent part of compiler
  - Different optimizations possible
  - IR to IR translation
  - Can be very computational intensive part

Compilers: organization (cont'd)



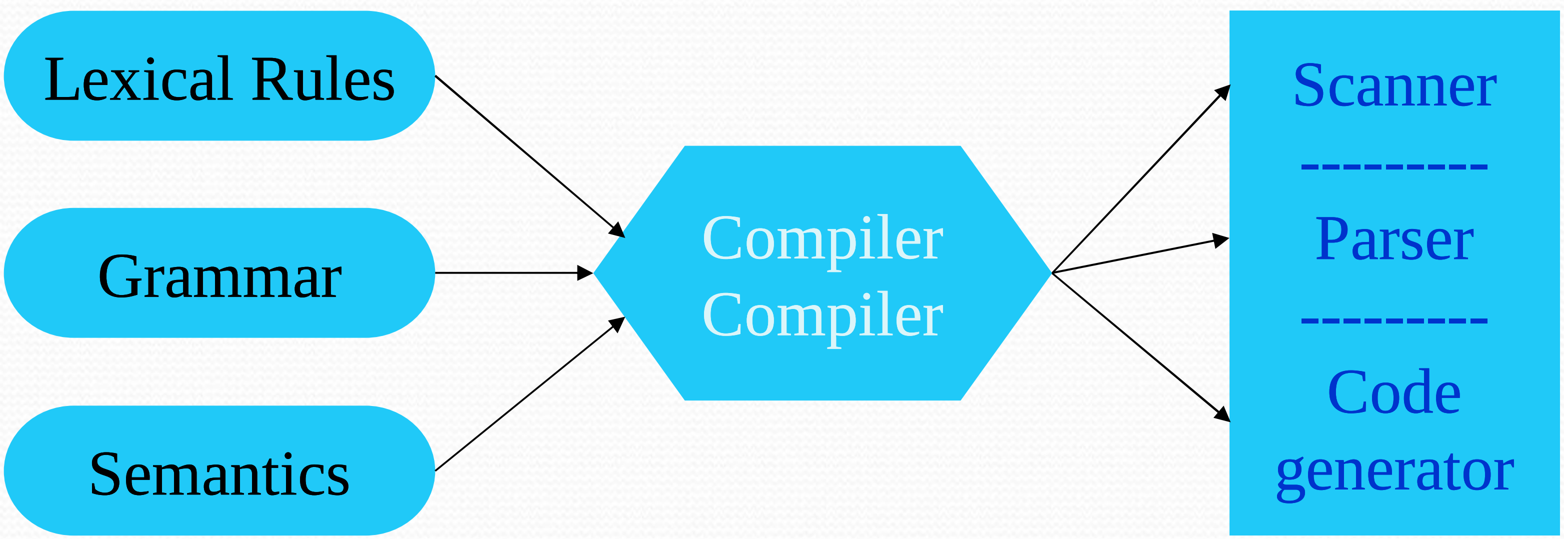
- Backend
  - Dependent on target processor
  - Code selection
  - Code scheduling
  - Register allocation
  - Peephole optimization

Frontend

Introduction to parsing using LEX and YACC

Overview

- Writing a compiler is difficult requiring lots of time and effort
- Construction of the scanner and parser is routine enough that the process may be automated



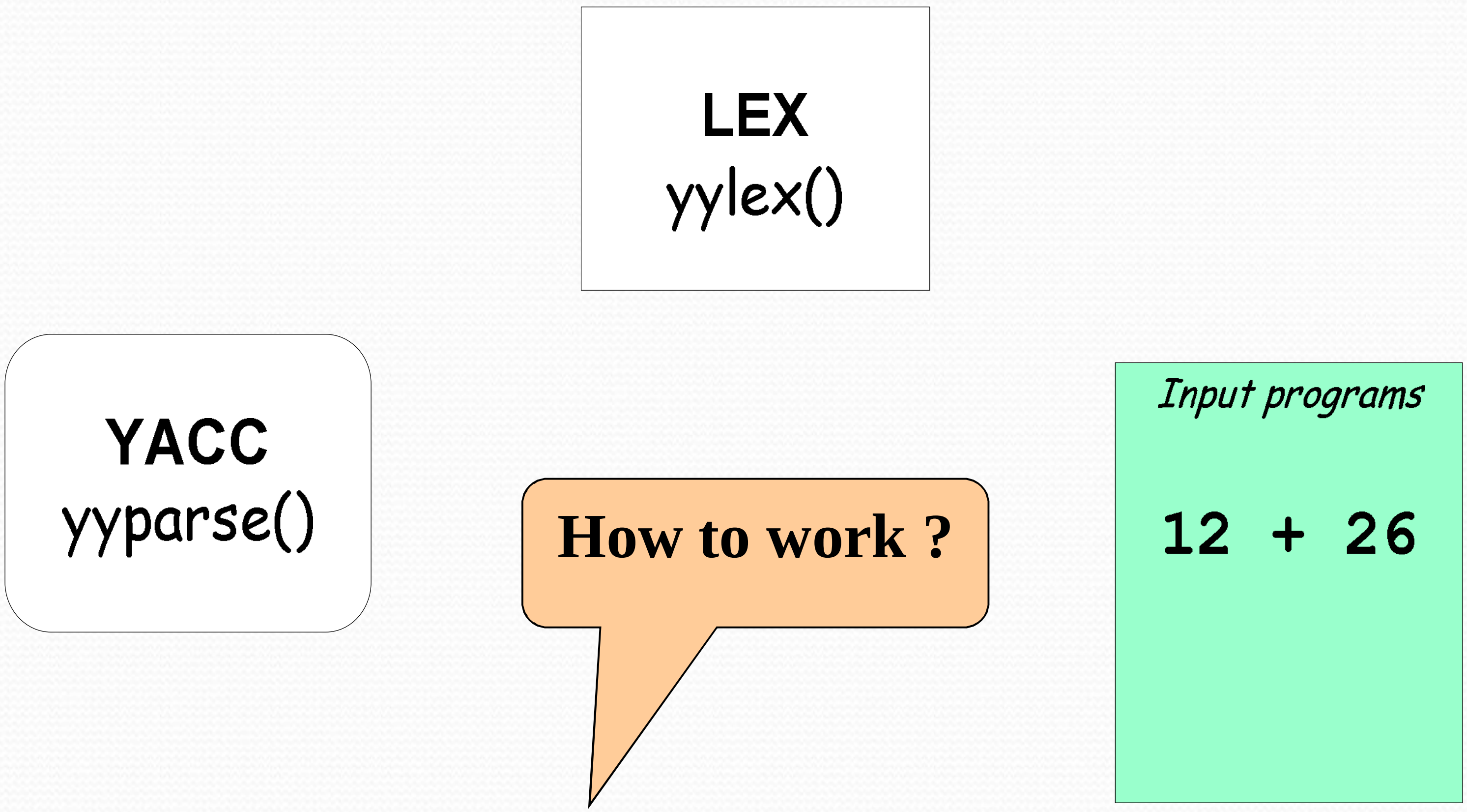
YACC

- 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
  - Input is a grammar (rules) and actions to take upon recognizing a rule
  - Output is a C program and optionally a header file of tokens

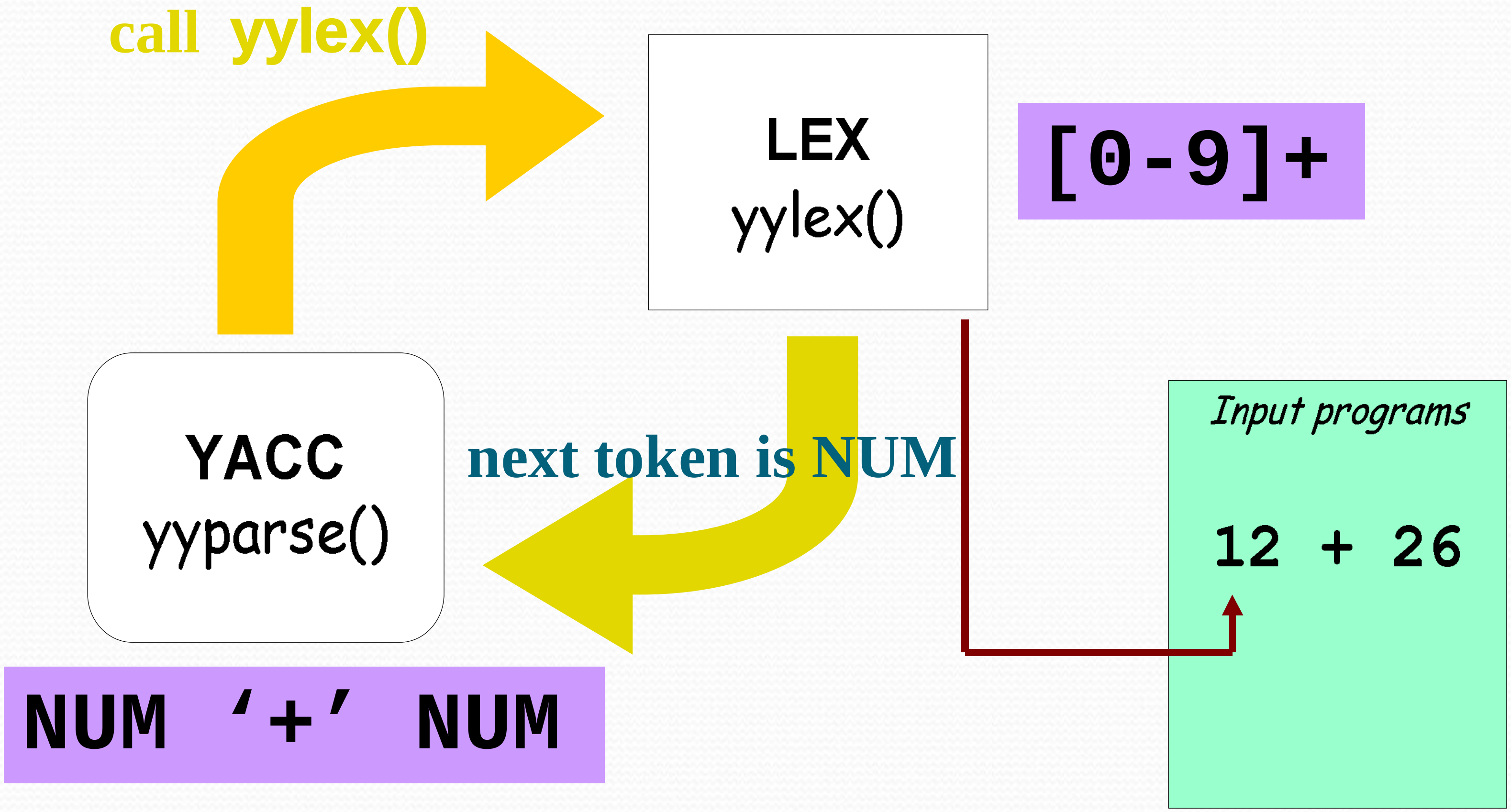
LEX

- Lex is a scanner generator
  - Input is description of patterns and actions
  - Output is a C program which contains a function yylex() which, when called, matches patterns and performs actions per input
  - Typically, the generated scanner performs lexical analysis and produces tokens for the (YACC-generated) parser

LEX and YACC: a team



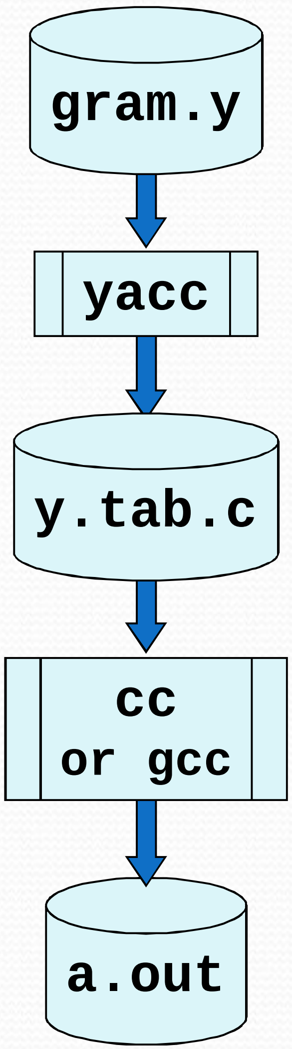
LEX and YACC: a team



Availability

- lex, yacc on most UNIX systems
- bison: a yacc replacement from GNU
- flex: *f*ast *lex*ical analyzer
- BSD yacc
- Windows/MS-DOS versions exist

YACC
Basic Operational Sequence



File containing desired grammar in YACC format

YACC program

C source program created by YACC

C compiler

Executable program that will parse grammar given in gram.y

YACC File Format

```
Definitions

%%

Rules

%%

Supplementary Code
```

The identical LEX format was actually taken from this...

Rules Section

- Is a 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
        ;
```

Definitions Section

Example

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

This is called a terminal

The start symbol (non-terminal)

Sidebar

- LEX produces a function called yylex()
- YACC produces a function called yyparse()

- yyparse() expects to be able to call yylex()

- How to get yylex()?
- Write your own!

- If you don't want to write your own: Use LEX!!!

Sidebar

```
int yylex()
{
  if(it's a num)
    return NUM;
  else if(it's an id)
    return ID;
  else if(parsing is done)
    return 0;
  else if(it's an error)
    return -1;
}
```

Semantic actions

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

Semantic actions (cont'd)

$1

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

Semantic actions (cont'd)

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

$2

Semantic actions (cont'd)

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

$3

Default: $$ = $1;

Bored, lonely? Try this!

```
yacc -d gram.y
```

- Will produce:

```
y.tab.h
```

Look at this and you'll never be unhappy again!

```
yacc -v gram.y
```

- Will produce:

```
y.output
```

Shows "State Machine"®

scanner.l

Example: LEX

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
wspc        [ \t\n]+
semi        [;]
comma       [,]
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{comma}     { return COMMA; }        /* Necessary? */
{semi}      { return SEMI; }
{id}        { return ID;}
{wspc}      {;}
```

Example: Definitions

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

decl.y

Example: Rules

```
/* This production is not part of the "official"
 * grammar. It's primary purpose is to recover from
 * parser errors, so it's probably best if you leave
 * it here. */

line :  /* lambda */
     | line decl
     | line error {
               printf("Failure :-(\n");
               yyerrok;
               yyclearin;
             }
     ;
```

Example: Rules

```
decl :  type ID list { printf("Success!\n"); } ;

list :  COMMA ID list
     | SEMI
     ;
type :   INT | CHAR | FLOAT
     ;

%%
```

Example: Supplementary Code

```
extern FILE *yyin;
main()
{
    do {
        yyparse();
    } while(!feof(yyin));
}
yyerror(char *s)
{
   /* Don't have to do anything! */
}
```

Bored, lonely? Try this!

`yacc -d decl.y`

- Produced

`y.tab.h`

```
# define CHAR 257
# define COMMA 258
# define FLOAT 259
# define ID 260
# define INT 261
# define SEMI 262
```

Symbol attributes

- Back to attribute grammars…
- Every symbol can have a value
  - Might be a numeric quantity in case of a number (42)
  - Might be a pointer to a string ("Hello, World!")
  - Might be a pointer to a symbol table entry in case of a variable
- When using LEX we put the value into yylval
  - In complex situations yylval is a union
- Typical LEX code:

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

Symbol attributes (cont'd)

- YACC allows symbols to have multiple types of value symbols

```
%union {
     double dval;
     int    vblno;
     char*  strval;
}
```

Symbol attributes (cont'd)

```
%union {
        double  dval;
        int     vblno;
        char*   strval;
}
```

yacc -d

y.tab.h

…

extern YYSTYPE yylval;

```
[0-9]+    { yylval.vblno = atoi(yytext);
            return NUM;}
[A-z]+    { yylval.strval = strdup(yytext);
            return STRING;}
```

LEX file

include "y.tab.h"

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

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

```
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; }
```

Precedence / Association

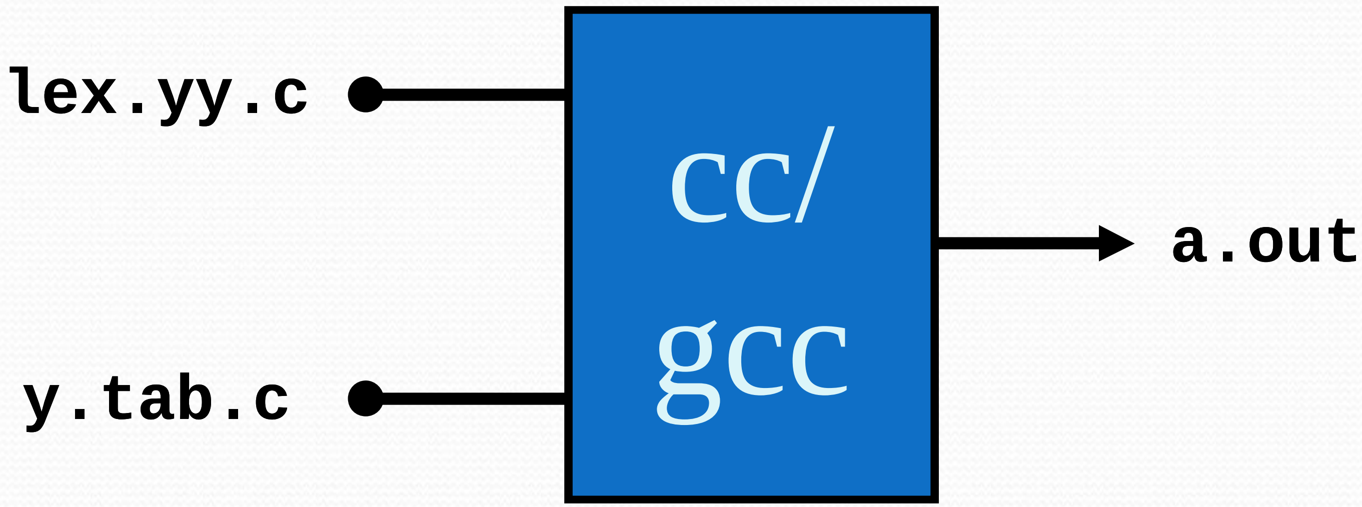
```
%right ‘=‘
%left  '<' '>' NE LE GE
%left  '+' '-‘
%left  '*' '/'
```

highest precedence

Big trick

Getting YACC & LEX to work together!

LEX & YACC



Building Example

- Suppose you have a lex file called **`scanner.l`** and a yacc file called **`decl.y`** and want **`parser`**
- Steps to build...

```
lex scanner.l
yacc -d decl.y
gcc -c lex.yy.c y.tab.c
gcc -o parser lex.yy.o y.tab.o -ll
```

Note: scanner should include in the definitions section:

```
#include "y.tab.h"
```

YACC

- Rules may be recursive
- Rules may be ambiguous
- Uses bottom-up Shift/Reduce parsing
  - Get a token
  - Push onto stack
  - Can it be reduced (How do we know?)
    - If yes: Reduce using a rule
    - If no: Get another token
- YACC cannot look ahead more than one token

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

stack:

<empty>

input:
a = 7; b = 3 + a + 2

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
NAME
```

```
input:
= 7; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

SHIFT!

stack:

NAME ‘=‘

input:

7; b = 3 + a + 2

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
NAME '=' 7
```

```
input:
; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:

NAME ‘=‘ exp
```

```
input:
; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:

stmt
```

```
input:
; b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';'
```

```
input:
b = 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME
```

```
input:
= 3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '='
```

```
input:
3 + a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '='
NUMBER
```

```
input:
+ a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:

stmt ';' NAME '='
exp
```

```
input:
+ a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:

stmt ';' NAME '='
exp '+'
```

```
input:
a + 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '='
exp '+' NAME
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:
stmt ‘;’ NAME ‘=‘
exp ‘+’ exp
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:
stmt ‘;’ NAME ‘=‘
exp
```

```
input:
+ 2
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ';' NAME '='
exp '+'
```

```
input:
2
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

SHIFT!

```
stack:
stmt ‘;’ NAME ‘=‘
exp ‘+’ NUMBER
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

stack:

`stmt ';' NAME '=' exp '+' exp`

input:
<empty>

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:
stmt ';' NAME '='
exp
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

stack:

stmt ';' stmt

input:
<empty>

Shift and reducing

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

REDUCE!

```
stack:

stmt
```

```
input:
<empty>
```

Shift and reducing

```
stmt: stmt ‘;’ stmt
    | NAME ‘=‘ exp

exp: exp ‘+’ exp
   | exp ‘-’ exp
   | NAME
   | NUMBER
```

DONE!

```
stack:

stmt
```

```
input:
<empty>
```

IF-ELSE Ambiguity

- Consider following rule:

```
stmt:
   IF expr stmt
   | IF expr stmt ELSE stmt
   ...
```

Following state :   IF expr IF expr stmt . ELSE stmt

- Two possible derivations:

IF expr IF expr stmt . ELSE stmt IF expr IF expr stmt ELSE . stmt IF expr IF expr stmt ELSE stmt . IF expr stmt	IF expr IF expr stmt . ELSE stmt IF expr stmt . ELSE stmt IF expr stmt ELSE . stmt IF expr stmt ELSE stmt .

IF-ELSE Ambiguity

- It is a shift/reduce conflict
- YACC will always do shift first
- Solution 1 : re-write grammar

```
stmt    : matched
        | unmatched
        ;
matched: other_stmt
        | IF expr THEN matched ELSE matched
        ;
unmatched: IF expr THEN stmt
          | IF expr THEN matched ELSE unmatched
          ;
```

IF-ELSE Ambiguity

- Solution 2:

```
%nonassoc IFX
%nonassoc ELSE

stmt:
    IF expr stmt %prec IFX
    | IF expr stmt ELSE stmt
```

the rule has the same precedence as token IFX

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.
  - e.g.: IF-ELSE ambiguity
- To resolve this conflict, YACC will choose to shift

Reduce/Reduce Conflicts

- ***Reduce/Reduce Conflicts:***

  start : expr | stmt

  ;

  expr : CONSTANT;

  stmt : CONSTANT;

- YACC (Bison) resolves the conflict by reducing using the rule that occurs earlier in the grammar. **NOT GOOD!!**
- So, modify grammar to eliminate them

Error Messages

- Bad error message:
  - Syntax error
  - Compiler needs to give programmer a good advice
- It is better to track the line number in LEX:

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

Recursive Grammar

- Left recursion

```
  list:
          item
        | list ',' item
        ;
```

- Right recursion

```
  list:
          item
        | item ',' list
        ;
```

- LR parser prefers left recursion
- LL parser prefers right recursion

YACC Example

- Taken from LEX & YACC
- Simple calculator

```
a = 4 + 6
a
a=10
b = 7
c = a + b
c
c = 17
pressure = (78 + 34) * 16.4
$
```

Grammar

```
expression ::= expression '+' term |
               expression '-' term |
               term

term       ::= term '*' factor |
               term '/' factor |
               factor

factor     ::= '(' expression ')' |
               '-' factor  |
               NUMBER |
               NAME
```

parser.h

```c
/*
 *     Header for calculator program
 */

#define NSYMS 20  /* maximum number
                     of symbols */

struct symtab {
  char *name;
  double value;
} symtab[NSYMS];

struct symtab *symlook();
```

0	name	value
1	name	value
2	name	value
3	name	value
4	name	value
5	name	value
6	name	value
7	name	value
8	name	value
9	name	value
10	name	value
11	name	value
12	name	value
13	name	value
14	name	value

•
•
•

parser.h

parser.y

```
%{
#include "parser.h"
#include <string.h>
%}

%union {
  double dval;
  struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER

%type <dval> expression
%type <dval> term
%type <dval> factor
%%
```

parser.y

```
statement_list:   statement '\n'
               |   statement_list statement '\n'
               ;

statement:  NAME '=' expression     { $1->value = $3; }
          |  expression          { printf("= %g\n", $1); }
          ;

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

parser.y

```
term:  term '*' factor { $$ = $1 * $3; }
    |  term '/' factor { if($3 == 0.0)
                             yyerror("divide by zero");
                         else
                             $$ = $1 / $3;
                       }
    | factor
    ;

factor: '(' expression ')' { $$ = $2; }
      | '-' factor         { $$ = -$2; }
      | NUMBER
      | NAME               { $$ = $1->value; }
      ;

%%
```

parser.y

```c
/* look up a symbol table entry, add if not present */
struct symtab *symlook(char *s) {
  char *p;
  struct symtab *sp;
  for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
      /* is it already here? */
      if(sp->name && !strcmp(sp->name, s))
            return sp;
      if(!sp->name) { /* is it free */
            sp->name = strdup(s);
            return sp;
      }
      /* otherwise continue to next */
  }
  yyerror("Too many symbols");
  exit(1);  /* cannot continue */
} /* symlook */
```

parser.y

```
yyerror(char *s)
{
   printf( "yyerror: %s\n", s);
}
```

parser.y

```
typedef union
{
  double dval;
  struct symtab *symp;
} YYSTYPE;

extern YYSTYPE yylval;

# define NAME 257
# define NUMBER 258
```

y.tab.h

calclexer.l

```
%{
#include "y.tab.h"
#include "parser.h"
#include <math.h>
%}
%%
```

calclexer.l

```
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
    yylval.dval = atof(yytext);
  return NUMBER;
  }

[ \t] ;              /* ignore white space */

[A-Za-z][A-Za-z0-9]*  { /* return symbol pointer */
                         yylval.symp = symlook(yytext);
                         return NAME;
                      }

"$"   { return 0; /* end of input */ }

\n    |.    return yytext[0];
%%
```

calclexer.l

Makefile

Makefile

```
LEX = lex
YACC = yacc
CC = gcc

calcu:      y.tab.o lex.yy.o
  $(CC) -o calcu y.tab.o lex.yy.o -ly -ll

y.tab.c y.tab.h: parser.y
  $(YACC)   -d parser.y

y.tab.o: y.tab.c parser.h
  $(CC) -c y.tab.c

lex.yy.o: y.tab.h lex.yy.c
  $(CC) -c lex.yy.c

lex.yy.c: calclexer.l parser.h
  $(LEX) calclexer.l
```

```
clean:
      rm *.o
      rm *.c
      rm calcu
```

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, e.g.:
x *op*. y *op*. z is syntax error)