

Lex & Yacc

- **Programming Tools for writers of compilers and interpreters**
- **Also interesting for non-compiler-writers**
- **Any application looking for patterns in its input or having an input/command language is a candidate for Lex/Yacc**

Lex & Yacc

- lex and yacc help you write programs that transform structured input
 - lex -- generates a lexical analyzer
 - divides a stream of input characters into lexemes, identifies them (token) and may pass the token to a parser generator, yacc
 - lex specifications are regular expressions
 - yacc -- generates a parser
 - may do syntax checking only or create an interpreter
 - yacc specifications are regular grammar components

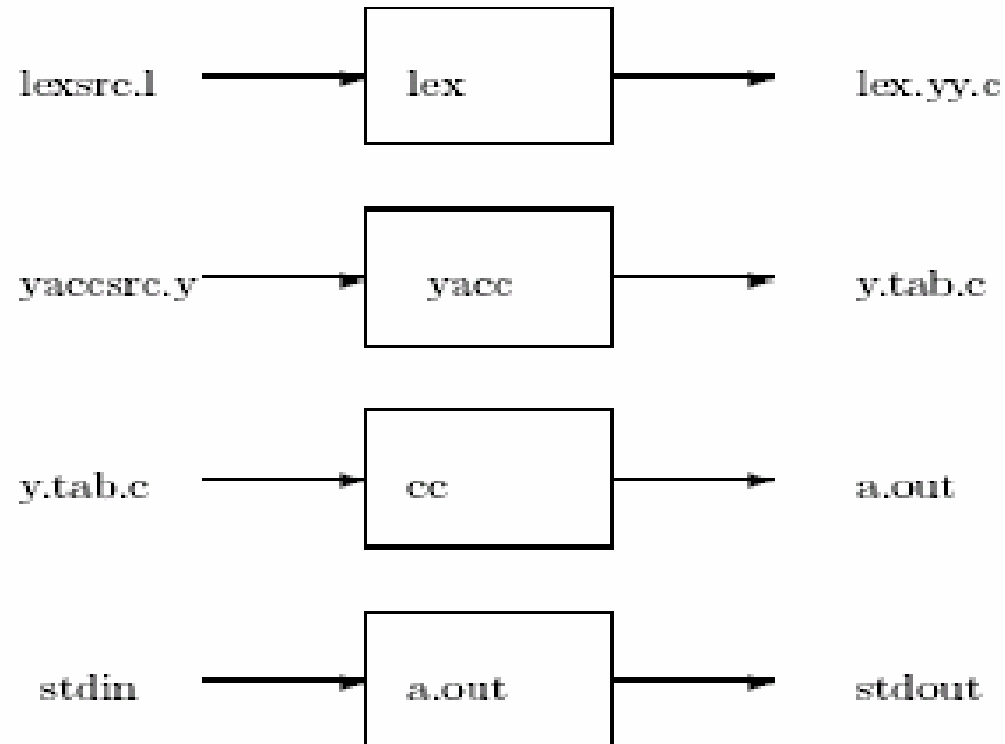
History of Lex & Yacc

- **Lex & Yacc were developed at Bell Laboratories in the 70's**
- **Yacc was developed as the first of the two by Stephen C. Johnson**
- **Lex was designed by Mike E. Lesk and Eric Schmidt to work with Yacc**
- **Standard UNIX utilities**

Lex

- **The Unix program “lex” is a “Lexical Analyzer Generator”**
 - Takes a high-level description of lexical tokens and actions
 - Generates C subroutines that implement the lexical analysis
 - The name of the resulting subroutine is “yylex”
- **Generally, yylex is linked to other routines, such as the bottom-up parsing procedures generated by YACC**

Yacc and Lex



Lex: breaking input stream into lexical tokens

- For example:

```
main() {  
    while (1)  
        x += 3.14;  
}
```

might be divided into these tokens

```
IDENTIFIER (main)  
LPAREN  
RPAREN  
LBRACE  
WHILE  
LPAREN  
WHOLENUM (1)  
RPAREN  
IDENTIFIER (x)  
PLUSEQ  
FLOAT (3.14)  
SEMI  
RBRACE
```

Organization of a Lex program

```
<declarations>  
%%  
<translation rules>  
%%  
<auxiliary procedures>
```

- **Translation rules consist of a sequence of patterns associated with actions**
- **Lex reads the file and generates a scanner**
 - Repeatedly locates the “longest prefix of the input that is matched by one or more of the patterns”
 - When the action is found, lex executes the associated action
 - In the case of a tie:
 - Use whichever regexp uses the most characters
 - If same number of characters, the first rule wins
 - The pre-defined action REJECT means “skip to the next alternative”

Simple Example

```
%%
```

```
. | \n      ECHO; /* matches any character or a  
                new line */
```

```
%%
```

This program copies standard input
to
standard output.

Disambiguation Rules

Given the following lex rules:

is am are	{printf("Verb\n");}
island	{printf("Island\n");}
[a-zA-Z]+	{printf("Unknown\n");}

How does lex choose *island* instead of *is* when it sees it?

1. lex patterns only match a given input character or string
2. lex executes the action for the longest possible match for the current input.

Regular Expressions in Lex

- References to a single character
 - `x` the character "x"
 - `"x"` an "x", even if x is an operator
 - `\x` an "x", even if x is an operator
 - `(x)` an x
 - `[xy]` the character x or y
 - `[x-z]` the character x, y or z
 - `[^x]` any character except x
 - `.` any character except newline
- Repetitions and options
 - `x?` an optional x
 - `x*` 0,1,2, ... instances of x
 - `x+` 1,2,3, ... instances of x

Regular Expressions in Lex

- Position dependant

- \hat{x} an x at the beginning of a line
- $x\$$ an x at the end of a line

- Misc

- $x|y$ an x or a y
- $\{xx\}$ section the translation of xx from the definitions
- x/y an x but only if followed by y
- $x\{m,n\}$ m through n occurrences of x
- $x\{n\}$ n occurrences of x
- $\langle y \rangle x$ an x when in start condition y

Lex Regular Expressions: Examples

- **0** matches only the character '0'
- **0123** matches the sequence of characters '0' '1' '2' '3'
- **\n** matches newline
- **[\n]** matches newline and space
- **(abc){3}** matches exactly 3 occurrences of the string "abc", i.e., "abcabcabc" is matched
- **[0-9]+** matches, e.g. "1", "000", "1234" but not an empty string

Lex Regular Expressions: Examples

- $(012)/a$ matches the string "012" if followed by "a". Note that "a" is not matched by this expression!
- $([a-z]+)/ \{$ matches a lower-case string, but only if followed by "{".
- $[a-z]^+$
- $[0-9]|[a-z]$ matches either a number or a lower-case letter.
- $.$ matches any character except for newline $\backslash n$
- $(-?[0-9]^+)$ matches an integer with an optional unary minus. For example, "123" or "-0123" is matched by this expression
- $^[\ \t]^*\backslash n$ matches any line which is not entirely whitespaced

Lex Declarations and Translation

Rules Section

- Any line that begins with a blank or a tab and is not part of a *lex rule* or *definition* is copied verbatim to the generated program

```
extern int token; /* global declaration, placed  
                  outside definition of yylex() */
```

```
%%
```

```
int i,j,k; /* local declaration, placed  
          inside procedure yylex() */
```

- Anything between “%{” and “}%” is copied verbatim

```
%{
```

```
/* this is Ostermann's program... */
```

```
#include <stdio.h>
```

```
}%
```

Lex Auxiliary Procedures Section

- All source code following the second “%%” is copied verbatim to the generated program
- In the declarations section, any line that is not copied verbatim is a macro definition:

```
word [^ \t\n]+  
D [0-9]  
E [DEde] [+ -]? {D}+  
%%
```

Example Lex Input File for a simple Calculator (calc.l)

```
%{
#include "y.tab.h"
extern int yylval; /* expected by yacc; bison does that
                    automatically */

%}
%%
[0-9]+      { yylval = atoi(yytext); return NUMBER;}
[ \t];     /* ignore whitespace */
\n         {return 0;} /* logical EOF */
"+"        {return PLUS;}
"-"        {return MINUS;}
"*"        {return TIMES;}
"/"        {return DIVIDE;}
%%
```

Lex Details

- The input file to lex must end in “.l” or “.lex”
- Lex generates a C file as output
 - Called lex.yy.c by default
- Blanks and tabs terminate a regular expression
 - Programmer-defined actions are separated from regular expressions by a space or a tab character
- Each time a pattern is matched, the corresponding action is executed
 - The default action is ECHO, which is basically

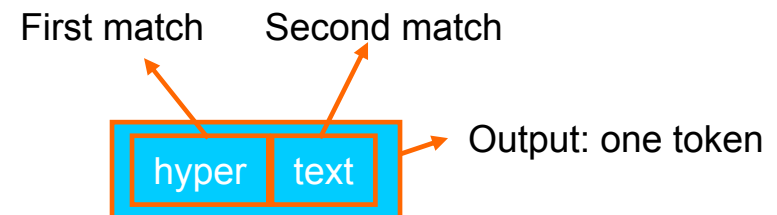
```
printf("%s", yytext);
```
- yytext is lex’s internal buffer to hold the current token
 - yyleng is the length of the matched token
- yylval is a global variable that contains a (possible) value associated with a token (we will discuss that in detail later). It is used by the parser.

More Lex Details: yymore

- `yymore()`
 - Append the next matched token to the end of the current matched token
 - Restart at start state, pretend that both regular expressions are a single token
- **Example:**

```
%%  
hyper {yymore();}  
text  {printf("Token is %s\n", yytext);}
```

Input: "hypertext"
Output: "Token is hypertext"



More Lex Details: `yyles`

- `yyles(n)`
 - Push back all but the first n characters of the token.
- **Example:**

```
\("[^"]*" { /* is the char before close quote a \ ? */  
  if (yytext[yytext-2] == '\\') {  
    yyles(yytext-1); /* return last quote */  
    yymore(); /* append next string */  
  }
```

Yacc Introduction

- Yacc is a theoretically complicated, but “easy” to use program that parses input files to verify that they correspond to a certain language
- Your main program calls `yyparse()` to parse the input file
- The compiled YACC program automatically calls `yylex()`, which is in `lex.yy.c`
- You really need a Makefile to keep it all straight

Yacc Introduction

- Yacc takes a grammar that you specify (in BNF form) and produces a parser that recognizes valid sentences in your language
- Can generate interpreters, also, if you include an action for each statement that is executed when the statement is recognized (completed)

The Yacc Parser

- Parser reads tokens; if token does not complete a rule it is pushed on a stack and the parser switches to a new state reflecting the token it just read
- When it finds all tokens that constitute the right hand side of a rule, it pops of the right hand symbols from the stack and pushes the left hand symbol on the stack (called a *reduction*)
- Whenever yacc reduces a rule, it executes the user code associated with the rule
- Parser is referred to as a *shift/reduce parser*
- yacc cannot run alone -- it needs lex

Simple Example

Statement -> id = expression
expression -> NUMBER
expression + NUMBER
expression - NUMBER

Parser actions: Input: x = 3 + 2 Scanner: id = NUMBER + NUMBER

id	Push id
id =	Push =
id = NUMBER	Push NUMBER
id = NUMBER +	Push +
id =	
NUMBER + NUMBER	Push NUMBER; Reduces expression -> NUMBER + NUMBER
id = expression	Pop NUMBER; pop +; pop expression; push expression Reduces statement -> id = expression
statement	Pop expression; pop =; pop id; push statement

Organization of a Yacc file

- Definition section
 - Declarations of tokens used in grammar, the types of values used on the parser stack and other odds and ends
 - For example, %token PLUS, MINUS, TIMES, DIVIDE
 - Declaration of non-terminals, %union, etc.

Organization of a Yacc file

- Rules section
 - A list of grammar rules in BNF form
 - Example:

```
expression:      expression PLUS expression      {$$ = $1 + $3;}
                | expression MINUS expression    {$$ = $1 - $3;}
                | NUMBER                          {$$ = $1;}
                ;
```

- Each rule may or may not have an associated action (actions are what make an interpreter out of a syntax checker)
- Action code can refer to the values of the right hand side symbols as \$1, \$2, ..., and can set the value of the left-hand side by setting \$\$=....

Organization of a Yacc file

- Auxiliary subroutine section
 - Typically includes subroutines called from the actions
 - Are copied verbatim to the generated C file (the parser)
 - In large programs it may be more convenient to put the supporting code in a separate source file

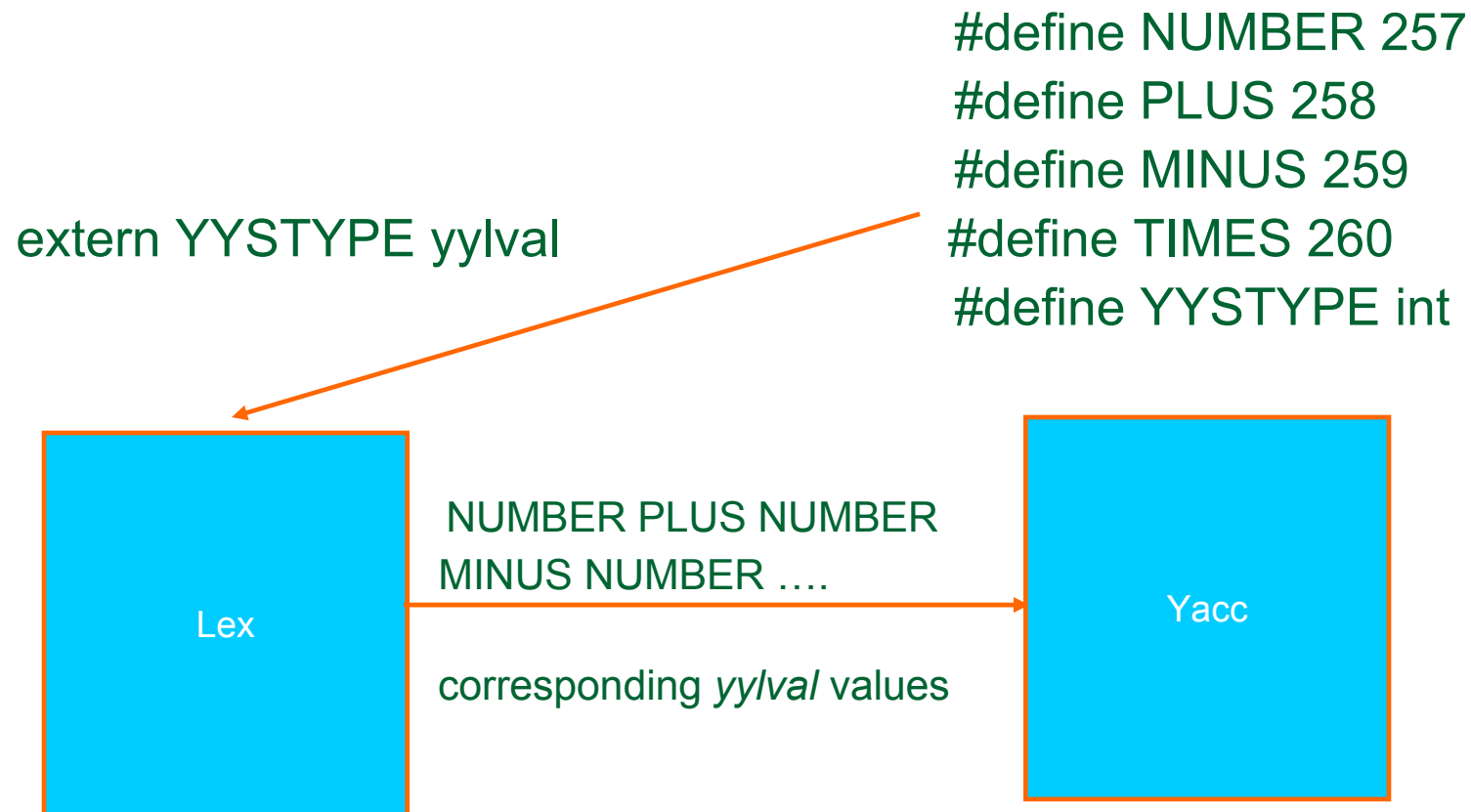
Symbol Values and Actions

- Every symbol in a yacc parser has a value
 - Terminal symbols (= Tokens from the scanner)
 - If a symbol represents a number, then its value is that number's value
 - If it represents a string, it probably is the pointer to the string
 - If it is a variable, the value is probably the index in the symbol table
 - Non terminal symbols can have any values you wish
 - When a parser reduces a rule (completes it), it executes the C code associated with it

Communication between Lex and Yacc

- Whenever Lex returns a token to the parser, that has an associated value, the lexer must store the value in the global variable *yylval* before it returns.
- The variable *yylval* is of the type `YYSTYPE`; this type is defined in the file `yy.tab.h` (created by yacc using the option `'-d'`).
- By default it is *integer*.
- If you want to have tokens of multiple valued types, you have to list all the values using the *%union* declaration

Communication between Lex and Yacc



Typed Tokens (%union declaration)

Example:

```
%token PLUS, MINUS, DIVIDE, TIMES
%union {
    double nval;
    char * varname;
}
%token <varname> NAME
%token <nval> NUMBER
%type <nval> expression /* %type sets the type for non-terminals */
%%
.....
```

Typed Tokens (%union declaration)

Yacc will create a header file y.tab.h like this:

```
#define NAME 257  
#define NUMBER 258  
#define UMINUS 259
```

```
typedef union {  
    double nval;  
    char * varname;  
} YYSTYPE;
```

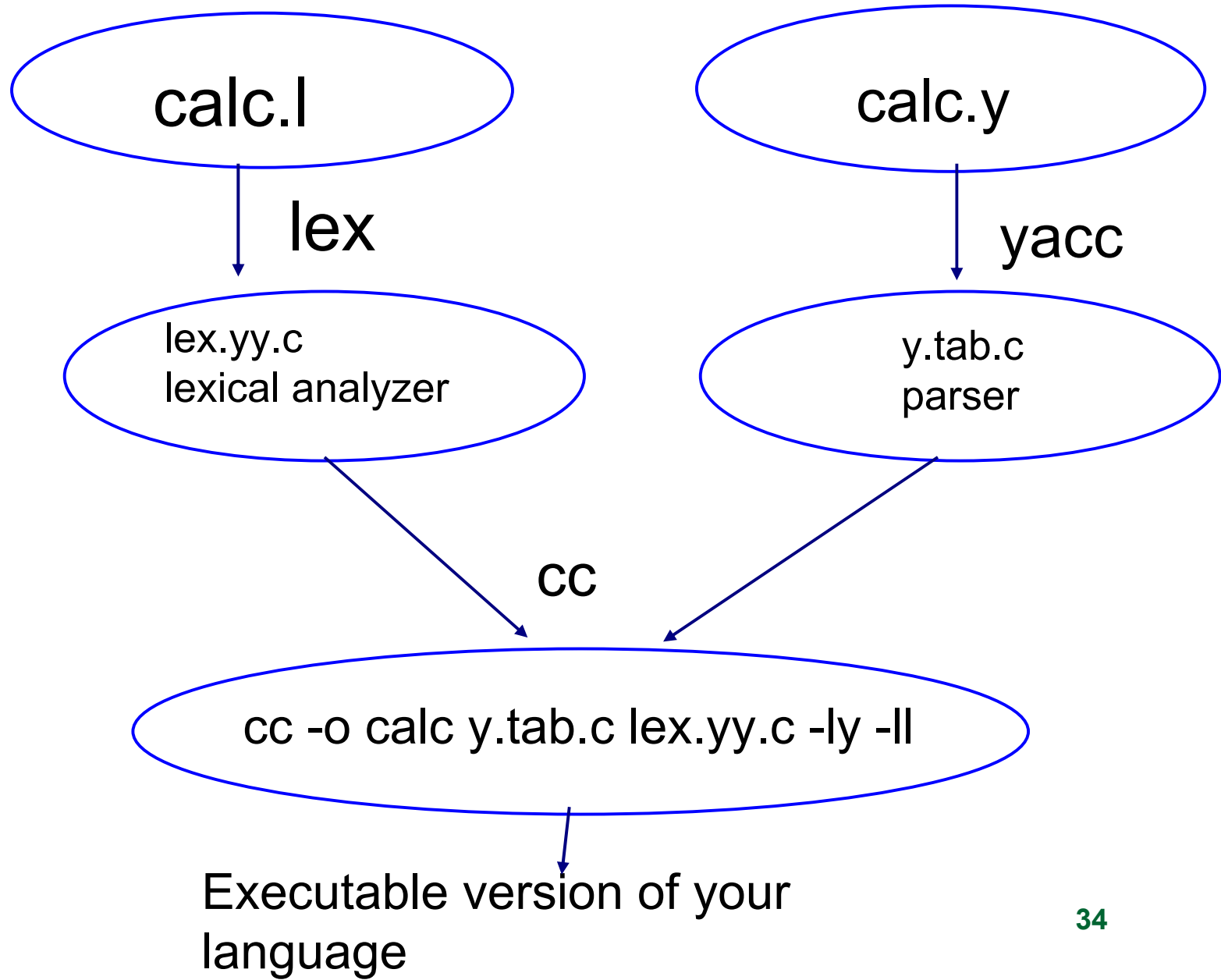
```
extern YYSTYPE yylval;
```

Yacc file for the calculator example (calc.y)

```
%token NUMBER, PLUS, MINUS, TIMES, DIVIDE
%left MINUS PLUS
%left TIMES DIVIDE'
%nonassoc UMINUS
%%
statement : expression          {printf(“=%d\n”,$1);}
          ;
expression:
    | expression PLUS expression {$$ = $1 + $3;}
    | expression MINUS expression {$$ = $1 - $3;}
    | expression TIMES expression {$$ = $1 * $3;}
    | expression DIVIDE expression {if ($3 == 0)
                                     yyerror(“divide by zero”);
                                     else
                                     $$ = $1 / $3;}
    | '-' expression %prec UMINUS {$$ = -$2;}
    | '(' expression ')' {$$ = $2;}
    | NUMBER {$$ = $1;}
    ;
%%
```

How it works

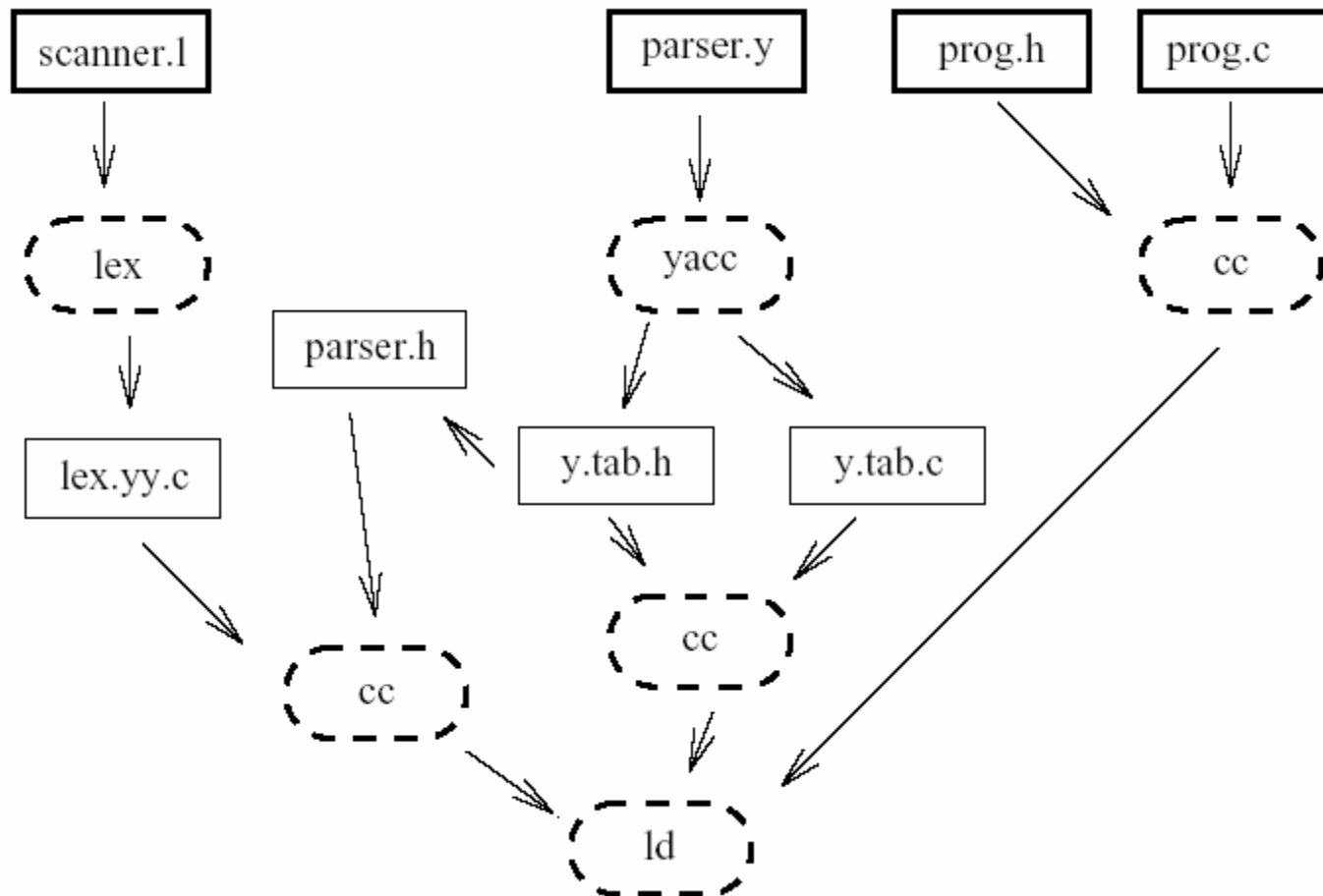
- yacc creates a C file that represents the parser for a grammar
- yacc requires input from a lexical analyzer; lexical analyzer no longer calls *yylex* because yacc does that
- Each token is passed to yacc as it is produced and handled by yacc; yacc defines the the token names in the parser as C preprocessor names in *y.tab.h*



Additional Functions of yacc

- **yyerror(s)**
 - This error-handling subroutine only prints a syntax error message.
- **yywrap()**
 - The wrap-up subroutine that returns a value of 1 when the end of input occurs.
 - supports processing of multiple input files as one
- **Both functions can be redefined by user (in the auxiliary subroutines section).**

Yacc, Lex, and Files Functional Diagram



Bigger Example “arith1” in archive

- This program understands a simple language of calculators
- A valid expression (expr) can be
 - A number
 - A number op expr
- It builds the data structure

```
struct assignment {  
    int    number[MAX_OPERATIONS];  
    int    operators[MAX_OPERATIONS];  
    int    nops;  
};
```

Bigger Example “arith1” (continued)

```
input :      lines
      |
      ;
```

```
lines :      oneline EOLN
      |      oneline EOLN lines
      ;
```

```
oneline :   expr | error
          ;
```

```
expr :      rhs;
```

```
rhs :       NUMBER | NUMBER oper rhs;
```

```
oper :      PLUS | MINUS | TIMES | DIVIDE;
```

Bigger Example “arith1” (continued)

```
struct opchain { /* operator chain */
int number;
int operator;
struct opchain *next;
};
```

```
%union {
    int number;
    int operator;
    struct assignment *pass;
    struct opchain* pop;
}
```

```
%token EOLN PLUS MINUS TIMES DIVIDE
```

```
%token <number> NUMBER
```

```
%type <pass> expr
```

```
%type <rhs> pop
```

```
%type <operator> oper
```

Bigger Example “arith1” (continued)

```
Input      :      lines | ;
lines     :      oneline EOLN | oneline EOLN lines;
Oneline   :      expr { doline($1); } | error;
expr      :      rhs
{
    struct assignment *pass;
    struct opchain *pop;
    pass = malloc(sizeof(struct assignment));
    for (pop = $1; pop; pop = pop->next) {
        pass->numbers[pass->nops] = pop->number;
        pass->operators[pass->nops] = pop->operator;
        ++pass->nops;
    }
    $$ = pass;
}
```

Bigger Example “arith1” (continued)

```
rhs      :      NUMBER
          {
            $$ = malloc(sizeof(struct opchain));
            $$->number = $1;
          }
          |      NUMBER oper rhs
          {
            $$ = malloc(sizeof(struct opchain));
            $$->operator = $2;
            $$->number = $1;
            $$->next = $3;
          }
          ;

/* one of the 4 operators we understand */
oper     :      PLUS    { $$ = PLUS; }
          |      MINUS  { $$ = MINUS; }
          |      TIMES  { $$ = TIMES; }
          |      DIVIDE { $$ = DIVIDE; }
          ;
```

Bigger Example “arith1” (calc.h -- header file)

```
#define MAX_OPERATIONS 100
struct assignment {
    int numbers[MAX_OPERATIONS];
    int operators[MAX_OPERATIONS];
    int nops;
};
/* externals */
extern int yydebug;

/* routine decls */
void doline(struct assignment *pass);
int yyparse(void);
```

Bigger Example “arith1” (calc.c – main program)

```
int main(int argc, char *argv[])
{
    yydebug = 1; /* enable debugging */
    /* parse the input file */
    yyparse();
    exit(0);
}

void doline(struct assignment *pass)
{
    printf("Read a line:\n");
    doexpr(pass);
}
```

Bigger Example “arith1” (calc.c – main program)

```
static void doexpr(struct assignment *pass)
{
    int i, sum, nextterm;
    printf(" Number of operations: %d\n", pass->nops);
    printf(" Question: '");
    sum = pass->numbers[0];
    for (i=0; i < pass->nops; ++i) {
        printf(" %d", pass->numbers[i]);
        if (i+1 < pass->nops) {
            nextterm = pass->numbers[i+1];
            switch(pass->operators[i]) {
                case PLUS    : printf(" +"); sum += nextterm; break;
                case MINUS   : printf(" -"); sum -= nextterm; break;
                case TIMES   : printf(" *"); sum *= nextterm; break;
                case DIVIDE  : printf(" /"); sum /= nextterm; break;
                default      : printf("? "); break;
            }
        }
    }
    printf("\n answer is %d\n\n", sum);
}
```

September 2003

September 2003							October 2003							
S	M	T	W	T	F	S	S	M	T	W	T	F	S	
	1	2	3	4	5	6		5	6	7	8	9	10	11
7	8	9	10	11	12	13	12	13	14	15	16	17	18	
14	15	16	17	18	19	20	19	20	21	22	23	24	25	
21	22	23	24	25	26	27	26	27	28	29	30	31		
28	29	30												

Monday	Tuesday	Wednesday	Thursday	Friday	Sat/Sun
September 1	2	3	4	5	6
					7
8	9	10	11	12	13
					14
15	16	17	18	19	20
					21
22	23	24	25	26	27
				TA: lecture	28
29	30				
	TA: Project discussion (part 1 and part 2)				

October 2003

October 2003							November 2003						
S	M	T	W	T	F	S	S	M	T	W	T	F	S
			1	2	3	4							1
5	6	7	8	9	10	11	2	3	4	5	6	7	8
12	13	14	15	16	17	18	9	10	11	12	13	14	15
19	20	21	22	23	24	25	16	17	18	19	20	21	22
26	27	28	29	30	31		23	24	25	26	27	28	29
							30						

Monday	Tuesday	Wednesday	Thursday	Friday	Sat/Sun
		October 1	2	3	4
	TA: Project discussion (part 1 and part 2)		TA: lecture	Quiz 2	Deadline. Project 1, Part 1
6	7	8	9	10	11
Class cancelled (make-up class)					
13	14	15	16	17	18
	Midterm Exam				
20	21	22	23	24	25
27	28	29	30	31	