Today's Lecture

1. Projects
2. `lex` - breaks input into tokens
3. `yacc` - imposes structure on the token stream
4. A gallery of Unix utilities

`lex`: A Tool for Generating Lexers

- Many programs need their input separated into *tokens*
- This is a often non-trivial task (*e.g.* in a compiler)
- Hand-written *lexical analyzers* (*lexers*) are error prone—hard to modify and extend
- `lex` automates lexer generation
- Also works for trivial token streams, *e.g.* command-line arguments
- With `lex`, the set of tokens is *extensible*
- and modification of existing tokens is trivial
- Allows *modal* tokenization
- How does `sh.c` tokenize its input?

`lex`: Simple Example


```
 /* simple.lex */
 /* note that comments must be indented! */


%{  /* declarations */

 /*
  * C declarations, etc. may go here
  */

%}

%%  /* rules */

.|\n    ECHO;

%%

/* any C code */
/* note: comments here are sane */

main (int argc,char *argv[])
{
    yylex();
}

 /* end of simple.lex */
```


`lex`: Building & Running `simple`


```
$ ls
simple.lex
$ lex simple.lex
$ ls
lex.yy.c        simple.lex
$ gcc -ansi -o simple lex.yy.c -ll
$ ls
lex.yy.c        simple*         simple.lex
$
$ simple
a run of the program lex
a run of the program lex
produced from the
produced from the
description in simple.lex
description in simple.lex


it copies input characters
it copies input characters
to the output
to the output
$
```

`lex`'s Output

`lex` generates as its output a C program that reads standard input (by default).

Input is matched (via regular expressions) as specified by the `lex` "rules" section.

C code can be attached to the rules; when a rule matches, the attached code is executed.

The `lex` generated program may be integrated with other C code, compiled, *etc.*

What do you expect the programs generated by `lex` to contain?

Here's the program produced by the `simple.lex` description:

```
# include "stdio.h"
# define U(x) x
# define NLSTATE yyprevious=YYNEWLINE
# define BEGIN yybgin = yysvec + 1 +
# define INITIAL 0
# define YYLERR yysvec
# define YYSTATE (yyestate-yysvec-1)
# define YYOPTIM 1
# define YYLMAX BUFSIZ
# define output(c) putc(c,yyout)
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr):getc(yyin))==10 \
                ?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
# define unput(c) {yytchar= (c);if(yytchar=='\n')yylineno--;*yysptr++=yytchar;}
# define yymore() (yymorfg=1)
# define ECHO fprintf(yyout, "%s",yytext)
# define REJECT { nstr = yyreject(); goto yyfussy;}
int yyleng; extern char yytext[];
int yymorfg;
extern char *yysptr, yysbuf[];
int yytchar;
FILE *yyin = {stdin}, *yyout = {stdout};
extern int yylineno;
struct yysvf {
        struct yywork *yystoff;
        struct yysvf *yyother;
        int *yystops;};
struct yysvf *yyestate;
extern struct yysvf yysvec[], *yybgin;

/*
 * C declarations, etc. can go here
 */

# define YYNEWLINE 10
yylex(){
int nstr; extern int yyprevious;
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr){
case 0:
if(yywrap()) return(0); break;
case 1:
ECHO;
break;
case -1:
break;
default:
fprintf(yyout,"bad switch yylook %d",nstr);
} return(0); }
/* end of yylex */

main (int argc,char *argv[])
{
```



```
    yylex();
}
int yyvstop[] = {
0,

1,
0,
0};
# define YYTYPE char
struct yywork { YYTYPE verify, advance; } yycrank[] = {
0,0,    0,0,    1,3,    0,0,
0,0,    0,0,    0,0,    0,0,
0,0,    0,0,    0,0,    1,3,
0,0};
struct yysvf yysvec[] = {
0,      0,      0,
yycrank+-1,     0,              0,
yycrank+0,      yysvec+1,       0,
yycrank+0,      0,              yyvstop+1,
0,      0,      0};
struct yywork *yytop = yycrank+11;
struct yysvf *yybgin = yysvec+1;
char yymatch[] = {
00  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,012 ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
0};
char yyextra[] = {
0,0,0,0,0,0,0,0,
0};
#ifndef lint
static  char ncform_sccsid[] = "@(#)ncform 1.6 88/02/08 SMI"; /* from S5R2 1.2 */
#endif

int yylineno =1;
# define YYU(x) x
# define NLSTATE yyprevious=YYNEWLINE
char yytext[YYLMAX];
struct yysvf *yylstate [YYLMAX], **yylsp, **yyolsp;
char yysbuf[YYLMAX];
char *yysptr = yysbuf;
```



```
int *yyfnd;
extern struct yysvf *yyestate;
int yyprevious = YYNEWLINE;
yylook(){
        register struct yysvf *yystate, **lsp;
        register struct yywork *yyt;
        struct yysvf *yyz;
        int yych, yyfirst;
        struct yywork *yyr;
# ifdef LEXDEBUG
        int debug;
# endif
        char *yylastch;
        /* start off machines */
# ifdef LEXDEBUG
        debug = 0;
# endif
        yyfirst=1;
        if (!yymorfg)
                yylastch = yytext;
        else {
                yymorfg=0;
                yylastch = yytext+yyleng;
                }
        for(;;){
                lsp = yylstate;
                yyestate = yystate = yybgin;
                if (yyprevious==YYNEWLINE) yystate++;
                for (;;){
# ifdef LEXDEBUG
                        if(debug)fprintf(yyout,"state %d\n",yystate-yysvec-1);
# endif
                        yyt = yystate->yystoff;
                        if(yyt == yycrank && !yyfirst){  /* may not be any transitions */
                                yyz = yystate->yyother;
                                if(yyz == 0)break;
                                if(yyz->yystoff == yycrank)break;
                                }
                        *yylastch++ = yych = input();
                        yyfirst=0;
                tryagain:
# ifdef LEXDEBUG
                        if(debug){
                                fprintf(yyout,"char ");
                                allprint(yych);
                                putchar('\n');
                                }
# endif
                        yyr = yyt;
                        if ( (int)yyt > (int)yycrank){
                                yyt = yyr + yych;
                                if (yyt <= yytop && yyt->verify+yysvec == yystate){
                                        if(yyt->advance+yysvec == YYLERR)       /* error transitions */
                                                {unput(*--yylastch);break;}
```

```
					*lsp++ = yystate = yyt->advance+yysvec;
					goto contin;
					}
				}
# ifdef YYOPTIM
			else if((int)yyt < (int)yycrank) {		/* r < yycrank */
				yyt = yyr = yycrank+(yycrank-yyt);
# ifdef LEXDEBUG
				if(debug)fprintf(yyout,"compressed state\n");
# endif
				yyt = yyt + yych;
				if(yyt <= yytop && yyt->verify+yysvec == yystate){
					if(yyt->advance+yysvec == YYLERR)	/* error transitions */
						{unput(*--yylastch);break;}
					*lsp++ = yystate = yyt->advance+yysvec;
					goto contin;
					}
				yyt = yyr + YYU(yymatch[yych]);
# ifdef LEXDEBUG
				if(debug){
					fprintf(yyout,"try fall back character ");
					allprint(YYU(yymatch[yych]));
					putchar('\n');
					}
# endif
				if(yyt <= yytop && yyt->verify+yysvec == yystate){
					if(yyt->advance+yysvec == YYLERR)	/* error transition */
						{unput(*--yylastch);break;}
					*lsp++ = yystate = yyt->advance+yysvec;
					goto contin;
					}
				}
			if ((yystate = yystate->yyother) && (yyt= yystate->yystoff) != yycrank){
# ifdef LEXDEBUG
				if(debug)fprintf(yyout,"fall back to state %d\n",yystate-yysvec-1);
# endif
				goto tryagain;
				}
# endif
			else
				{unput(*--yylastch);break;}
		contin:
# ifdef LEXDEBUG
			if(debug){
				fprintf(yyout,"state %d char ",yystate-yysvec-1);
				allprint(yych);
				putchar('\n');
				}
# endif
			;
			}
# ifdef LEXDEBUG
		if(debug){
			fprintf(yyout,"stopped at %d with ",*(lsp-1)-yysvec-1);
```



```
			allprint(yych);
			putchar('\n');
			}
# endif
		while (lsp-- > yylstate){
			*yylastch-- = 0;
			if (*lsp != 0 && (yyfnd= (*lsp)->yystops) && *yyfnd > 0){
				yyolsp = lsp;
				if(yyextra[*yyfnd]){		/* must backup */
					while(yyback((*lsp)->yystops,-*yyfnd) != 1 && lsp > yylstate){
						lsp--;
						unput(*yylastch--);
						}
					}
				yyprevious = YYU(*yylastch);
				yylsp = lsp;
				yyleng = yylastch-yytext+1;
				yytext[yyleng] = 0;
# ifdef LEXDEBUG
				if(debug){
					fprintf(yyout,"\nmatch ");
					sprint(yytext);
					fprintf(yyout," action %d\n",*yyfnd);
					}
# endif
				return(*yyfnd++);
				}
			unput(*yylastch);
			}
		if (yytext[0] == 0  /* && feof(yyin) */)
			{
			yysptr=yysbuf;
			return(0);
			}
		yyprevious = yytext[0] = input();
		if (yyprevious>0)
			output(yyprevious);
		yylastch=yytext;
# ifdef LEXDEBUG
		if(debug)putchar('\n');
# endif
		}
	}
yyback(p, m)
	int *p;
{
if (p==0) return(0);
while (*p)
	{
	if (*p++ == m)
		return(1);
	}
return(0);
}
```



```
	/* the following are only used in the lex library */
yyinput(){
	return(input());
	}
yyoutput(c)
  int c; {
	output(c);
	}
yyunput(c)
   int c; {
	unput(c);
	}
```


lex: Lexing Decimal Numerals


```
 /* decimal.lex */

%%

[\n\t ]         ;

-?((([0-9]+)|([0-9]*\.[0-9]+))([eE][-+]?[0-9]+)?) {
                printf("number: %s\n", yytext);
        }

\n      { putchar('\n'); }

.       { printf("not a number: %s\n", yytext); }

%%

 /* end of decimal.lex */
```

```
$ lex decimal.lex
$ gcc -ansi -o decimal lex.yy.c -ll
$ decimal
45
number: 45
3.2
number: 3.2
-.123
number: -.123
2.03e17
number: 2.03e17
e.23
not a number: e
number: .23
2e-2
number: 2e-2
$
```

lex: Simple Calculator


```
 /* stack-calc.lex */

%{
#define STACK_SIZE 100

double stack[STACK_SIZE];
int sp = 0;

void fatal(char *s, int c);
%}

%%
[\n\t ] ;

-?((([0-9]+)|([0-9]*\.[0-9]+))([eE][-+]?[0-9]+)?) {
                if (sp < STACK_SIZE) sscanf(yytext, "%lf", &(stack[sp++]));
                else fatal("stack overflow", 1);
        }

^print  { if (sp > 0) printf("%lf\n", stack[sp-1]);
          else fatal ("stack underflow", 2); }

^add    { if (sp > 1) { stack[sp-2] = stack[sp-2] + stack[sp-1];  sp--;
          } else fatal ("stack underflow", 2); }

^pop    { if (sp > 0) sp--;
          else fatal ("stack underflow", 2); }

.       { fprintf(stderr, "not a number: %s\n", yytext); }
%%

void fatal(char *s, int c) { fprintf(stderr, "%s\n", s); exit(c); }

 /* end of stack-calc.lex */
```


lex: Simple Calculator (cont.)


```
$ lex stack-calc.lex
$ gcc -ansi -o stack-calc lex.yy.c -ll
$ stack-calc
1
2
3
4
5
add
print
9.000000
add
add
add
print
15.000000
add
stack underflow
$
```


lex: Counting words à la wc


```
 /* wc.lex */

%{
unsigned int n_chars = 0,
             n_words = 0,
             n_lines = 0;
%}

word [^ \t\n]+
eol  \n

%%
{word}  { n_words++; n_chars += yyleng; }
{eol}   { n_lines++; n_chars++; }
.       n_chars++;
%%

main(int argc, char *argv[])
{
        yylex();
        printf("%d %d %d\n", n_lines, n_words, n_chars);
}

 /* end of wc.lex */
```

```
$ lex wc.lex
$ gcc -ansi -o mywc lex.yy.c -ll
$ mywc <wc.lex
24 55 326
$ wc wc.lex
      24      55     326 wc.lex
$
```


lex: Parsing Command-Lines


```
 /* cmdline-args.lex */

%{
int verbose = 0;          /* true, if we're in verbose mode */
int n_fname_pieces = 0;   /* number of pieces of a filename arg we've seen */
char *pname;              /* the program's name */
%}

 /* states:     FNAME is entered when lexing a cmdline filename option */
%s FNAME

%%
[ ]+            ; /* ignore spaces when in INITIAL state */
<FNAME>[ ]+     ; /* ignore spaces when in FNAME state */

-h(elp)?        { printf("usage: %s [-help | -h] [-verbose] -f filename\n", pname); }

-verbose        { printf("verbose mode is on\n");  verbose = 1; }

-f              { BEGIN FNAME; n_fname_pieces = 1; }

<FNAME>[^ \n]+  { printf("using file '%s'\n", yytext);
                  n_fname_pieces++;     /* another piece of the filename */
                  BEGIN INITIAL;        /* back to INITIAL state */
                }

[^ ]+           ;
%%

main(int argc, char *argv[])
{
        pname = argv[0];        /* initialize for help msg */
        yylex();
}

 /* end of cmdline-args.lex */
```

```
$ print -n ' -verbose -f fileA' | cmdline-args
verbose mode is on
using file 'fileA'
$
```

yacc: A Tool for Generating Parsers

`yacc` – Yet Another Compiler Compiler

Lexing only breaks input into tokens

However, it is often necessary to impose structure on the token stream

Examples: natural and computer languages

To understand tools like `yacc`, we need *grammars*

Context-free Grammars

Linguists led the way (1950's)

Context-free because enclosing context doesn't matter

Simplistic natural-language grammar:

$$
\begin{array}{rcl}
<\!\mathit{sentence}\!> & \longrightarrow & <\!\mathit{noun\ phrase}\!>\ <\!\mathit{verb\ phrase}\!> \\
<\!\mathit{noun\ phrase}\!> & \longrightarrow & <\!\mathit{adjective}\!>\ <\!\mathit{noun\ phrase}\!> \\
<\!\mathit{noun\ phrase}\!> & \longrightarrow & <\!\mathit{noun}\!> \\
<\!\mathit{noun}\ > & \longrightarrow & \text{dog} \\
<\!\mathit{adjective}\ > & \longrightarrow & \text{big} \\
<\!\mathit{adjective}\ > & \longrightarrow & \epsilon \\
<\!\mathit{verb\ phrase}\!> & \longrightarrow & <\!\mathit{verb\ phrase}\!>\ <\!\mathit{adverb}\!> \\
<\!\mathit{verb\ phrase}\!> & \longrightarrow & <\!\mathit{verb}\!> \\
<\!\mathit{verb}\ > & \longrightarrow & \text{runs} \\
<\!\mathit{adverb}\ > & \longrightarrow & \text{slowly} \\
<\!\mathit{adverb}\ > & \longrightarrow & \epsilon
\end{array}
$$

Productions (rules); Non-terminals *vs.* Terminals

Context-free Grammars (cont.)

Simplistic CFG for arithmetic expressions:

$$
\begin{array}{rcll}
<\!\mathit{expression}\!> & \longrightarrow & <\!\mathit{expression}\!> + <\!\mathit{expression}\!> & (1) \\
<\!\mathit{expression}\!> & \longrightarrow & <\!\mathit{expression}\!> * <\!\mathit{expression}\!> & (2) \\
<\!\mathit{expression}\!> & \longrightarrow & (<\!\mathit{expression}\!>) & (3) \\
<\!\mathit{expression}\!> & \longrightarrow & -<\!\mathit{expression}\!> & (4) \\
<\!\mathit{expression}\!> & \longrightarrow & \text{identifier} & (5)
\end{array}
$$

CFG can *derive* all valid strings in it's language (*e.g.*, simple arithmetic expressions or simple statements about a dog)

CFG can check whether a string is in it's language

CFG naturally corresponds to a parse tree

`yacc`'s Parser

- Given a CFG, a *parser* builds derivation tree for its input
- LALR(1) – Another common parser: LL(1)
- LALR(1) is in a class of parsers called *shift-reduce* parsers

---

Given a grammar, `yacc` creates a corresponding parser

Basic operation of a `yacc`-generated parser:

```
while ((t = get_token()) != EOF) {
    if t completes a rule then {
        reduce:
            pop rhs symbols off stack
            push lhs of rule onto stack
            change state
            execute user code for rule
    } else {
        shift:
            push t on stack
            change state
    }
}
```

yacc: Parsing Simple Expressions


```
 /* simple-arith.y */

%token NUMBER

%%
expression: expression '+' NUMBER       { $$ = $1 + $3; }
    |       expression '-' NUMBER       { $$ = $1 - $3; }
    |       '[' expression ']'          { printf("result is %d\n", $2); }
    |       NUMBER                      { $$ = $1; }
    ;

 /* end of simple-arith.y */
```

```
 /* simple-arith.lex */

%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+          { yylval = atoi(yytext); return NUMBER; }
[ \t]           ;
\n              return 0;  /* EOF */
.               return yytext[0];
%%

 /* end of simple-arith.lex */
```


yacc: simple-arith Sample Output


```
$ ls
simple-arith.lex  simple-arith.y
$ lex simple-arith.lex
$ ls
lex.yy.c          simple-arith.lex  simple-arith.y
$ yacc -d simple-arith.y
$ ls
lex.yy.c          simple-arith.y    y.tab.h
simple-arith.lex  y.tab.c
$ gcc -ansi -o simple-arith y.tab.c lex.yy.c -ly -ll
$ ls
lex.yy.c          simple-arith.lex  y.tab.c
simple-arith*     simple-arith.y    y.tab.h
$ simple-arith
1+2
$ simple-arith
[1+2]
result is 3
$ simple-arith
[2-1+9]
result is 10
$ simple-arith
[0-3+]
syntax error
$
```


yacc: Adding Operators to simple-arith


```
 /* extended-simple-arith.y */

%token NUMBER

%%
expression: expression '+' expression   { $$ = $1 + $3; }
    |       expression '-' expression   { $$ = $1 - $3; }
    |       expression '*' expression   { $$ = $1 - $3; }
    |       expression '/' expression   {
                    if ($3 == 0) yyerror("divide by zero");
                    else $$ = $1 / $3;
                }
    |       '(' expression ')'          { $$ = $2; }
    |       '-' expression              { $$ = -$2; }
    |       '[' expression ']'          { printf("result is %d\n", $2); }
    |       NUMBER                      { $$ = $1; }
    ;

 /* end of extended-simple-arith.y */
```

This grammar is *ambiguous*; input $1 + 2 - 3 * 4/5$ has more than one derivation:

- $(1 + 2) - ((3 * 4)/5)$
- $(1 + (2 - 3)) * (4/5)$
- many others

`yacc` issues *shift–reduce* conflicts for ambiguous grammars

To resolve conflicts, we need to assign precedence "weights" to operators, either implicitly or explicitly

yacc: Implicit Precedence

To remove shift–reduce conflicts implicitly, rewrite rules to "order" the derivation:

```
 /* implicit-precedence.y */

%token NUMBER

%%
statement:  '[' expression ']'          { printf("result is %d\n", $2); }

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

factor:     factor '*' primary       { $$ = $1 * $3; }
    |       factor '/' primary       {
                    if ($3 == 0)
                        yyerror("divide by zero");
                    else
                        $$ = $1 / $3;
                }
    |       primary
    ;

primary:    '(' expression ')'      { $$ = $2; }
    |       '-' primary             { $$ = -$2; }
    |       NUMBER                  { $$ = $1; }
    ;

 /* end of implicit-precedence.y */
```

Try this grammar with the lexer used in the `simple-arith` example.

yacc: Explicit Precedence

To remove shift–reduce conflicts explicitly, add `yacc` precedence rules:

```
 /* explicit-precedence.y */

%token NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%
statement:  '[' expression ']'         { printf("result is %d\n", $2); }

expression: expression '+' expression   { $$ = $1 + $3; }
    |       expression '-' expression   { $$ = $1 - $3; }
    |       expression '*' expression   { $$ = $1 * $3; }
    |       expression '/' expression   {
                    if ($3 == 0)
                        yyerror("divide by zero");
                    else
                        $$ = $1 / $3;
                }
    |       '(' expression ')'          { $$ = $2; }
    |       '-' expression %prec UMINUS { $$ = -$2; }
    |       '[' expression ']'          { printf("result is %d\n", $2); }
    |       NUMBER                      { $$ = $1; }
    ;

 /* end of explicit-precedence.y */
```

Try this grammar with the lexer used in the `simple-arith` example.

A Gallery of Unix Utilities

Useful filters

`grep`	select lines via regular-expression patterns
`head`	select first n lines
`tail`	select last n lines
`sort`	sort lines
`uniq`	find/count unique or repeated lines
`cut`	select fields or columns
`paste`	glue together two files column-wise
`join`	database join for pairs of sorted files
`tr`	translate/delete/squeeze characters
`cmp`	test if two files are equal
`diff`	find differences between similar files
`comm`	find lines common / missing from a pair of sorted files

Heavy duty guys

`sed`	Stream EDitor: a batch version of ed
`awk`	(Aho Weinberger Kernighan) general purpose language
`perl`	Kitchen sink; `awk` + `sh` + systems programming

Programs that write programs

`lex`	builds lexers from pattern specifications/C-actions
`yacc`	builds parsers from grammars

Next Lecture

- Graphics Shells
- Tcl/Tk
- Read `man` pages for:

  `wish`   `(man -M /usr/princeton/man wish)`