match, at the same length, lex will use the one which comes first in the rules section.
- If no pattern matches the input string, lex's default action is to copy the input to the output. The `ECHO` macro above simply makes this explicit.

---

lex Regular Expressions

These are substantially the same as those already introduced (thank goodness - BR)

`[ ]`
: class of characters. Eg. `[Bb]` matches either of B or b. The - character defines a range. A leading ^ character changes this to match any character *not* in the specified range.

`?`
: zero or one instance of

`*`
: zero or more instances of

`+`
: one or more instances of

`.`
: (dot) matches any character except the newline (`\n`)

`\`
: escapes (or "turns off") the special meaning of a single following metasymbol, such as those above.

`" "`
: string, enclosed in double quotes - characters are taken literally (as presented). Same effect on metasymbols as `\`.

`{xxx}`
: The regular expression that the name `xxx` represents.

---

lex Global Variables

`yytext`
: a pointer to the first character of the current lexeme - the one which generated a regular expression match.

`yyleng`
: an integer giving the length of the current lexeme

`yylval`
: yacc variable used to pass the value of a matched token back into yacc (see later).

`yyin`
: lex input file (default `stdin`)

`yyout`
: lex output file (default `stdout`)

`yymore()` & **`yyless(n)`**
: pushes next match or removes n characters from the `yytext` string (Enables a programmer to match patterns not expressible by a single regular expression). These two are actually auxiliary functions (see mext slide), not lex variables.

lex Auxiliary Functions

`yylex()`
is the lexical analyser function part of `lex.yy.c` When lex is called as a function from the parser, this is the name of the called function.

`input()`
retrieves a single character, returns 0 on EOF.

`output(c)`
write a single character to the output.

`unput(c)`
put a single character back on the input to be reread.

`yywrap()`
cleans up on EOF.

lex provides default functions for `input()`, `output()` and `unput()` which read and write to/from `stdio` (ie, `yyin` and `yyout`), so that a minimal lex program can operate as a **UNIX filter**.

Example lex Source File, part 2.

```
%{
unsigned cc=0, wc=0, lc=0;
%}

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

%%

{word}  {wc++; cc += yyleng;}

{eol}   {cc++; lc++;}

.       cc++;

%%

main()
{
    yylex();
    printf("%d %d %d", lc, wc, cc);
}

yywrap()
{
    return(1);
}
```

This lecture is available in PostScript format. The tutorial for this lecture is Tutorial #03 .
[Previous Lecture] [Lecture Index] [Next Lecture]

Next | Up | Previous | Contents

Next: Semantic Analysis - Dictionaries **Up:** ho **Previous:** Lexical analysis - Lex **Contents**

Subsections

Syntactic analysis - Yacc & Parse Trees

An infix calculator written using Lex and Yacc

($CS5031/e*/infix2/*)

We normally use Lex and Yacc together; Lex for the simple parts (e.g. numbers, names, operators) and Yacc for more complex parts (e.g. expressions, statements). For this example, we will keep the same operators as the postfix calculator, but we will give them their usual associativity (left), precedence (* and / before + and -) and we will also need brackets.

Yacc part

```
%{
#include <stdio.h>     /* C declarations used in actions */
%}

%union {int a_number;}          /* Yacc definitions */
%start line
%token <a_number> number
%type <a_number> exp term factor

%%

/* descriptions of expected inputs     corresponding actions (in C) */

line    : exp ';'               {printf ("result is %d\n", $1);}
        ;
exp     : term                  {$$ = $1;}
        | exp '+' term          {$$ = $1 + $3;}
        | exp '-' term          {$$ = $1 - $3;}
        ;
term    : factor                {$$ = $1;}
```

```
        | term '*' factor       {$$ = $1 * $3;}
        | term '/' factor       {$$ = $1 / $3;}
        ;
factor : number                 {$$ = $1;}
        | '(' exp ')'           {$$ = $2;}
        ;

%%                      /* C code */

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

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

Lex part

```
%{
#include "y.tab.h"
%}
%%

[0-9]+                  {yylval.a_number = atoi(yytext); return number;}
[ \t\n]                 ;
[-+*/( );]              {return yytext[0];}
.                       {ECHO; yyerror ("unexpected character");}

%%
int yywrap (void) {return 1;}
```

input descriptions

The format of the grammar rules for Yacc is:

```
name    : names and 'single character's
        | alternatives
        ;
```

Yacc definitions

`%start line`	means the whole input should match `line`
`%union`	lists all possible types for values associated with parts of the grammar and gives each a field-name
`%type`	gives an individual type for the values associated with each part of the grammar, using the field-names from the %union declaration
`%token`	declare each grammar rule used by YACC that is recognised by LEX and give type of value

Actions, C declarations & code:

`$$`	resulting value for any part of the grammar
`$1`, `$2`, etc.	values from sub-parts of the grammar
`yyparse`	routine created by YACC from (expected input, action) lists. (It actually returns a value indicating if it failed to recognise the input.)
`yylex`	routine called by yyparse for all its input. We are using getchar, which just reads characters from the input.
`yyerror`	routine called by yyparse whenever it detects an error in its input.

More Lex declarations and actions

`y.tab.h` gives LEX the names and type declarations etc. from YACC

`yylval` name used for values set in LEX e.g.

```
yylval.a_number = atoi (yytext);
yylval.a_name = findname (yytext);
```

How Lex and Yacc are used together:

flex : calcl.l → calcl.c
gcc : calcl.c → calcl.o
byacc : calcy.y → calcy.c
gcc : calcy.c → calcy.o
ld : calcl.o calcy.o → calc

calc : expression → result

Precedence and associativity

The example calculator above gives different precedence to the operators +, -, *, /, (and) by using a separate grammar rule for each precedence level. The operator associativities are also defined by the grammar rules, as you will investigate in the examples sheet.

The problem with these methods is that they tend to complicate the grammar and constrain language design. Yacc has an alternative mechanism that decouples the declarations of precedence and associativity from the grammar, using "%left", "%right" and "%nonassoc". e.g.:

```
  %nonassoc '='                   v
  %left '+' '-'                   v increasing
  %left '*' '/'                   v
  %right not                      v precedence
  %right '^'                      v              (i.e. exponentiation)
%%
  exp   : exp '+' exp | exp '-' exp | exp '*' exp | exp '/' exp
        | number | '(' exp ')' | exp '^' exp  | exp '=' exp | not exp
        ;
```

Yacc can also give different precedences to overloaded operators; e.g. if - is used both for negation and for subtraction and we want to give them different precedences (and negation the same precedence as for not):

```
| '-' exp %prec not
```

Yacc has other facilities, but those described above are among the most important. You can refer to the Yacc manual in the departmental library or at URL
http://www.cs.man.ac.uk/~pjj/cs2111/yacc/yacc.html
if necessary.

Exercises: Grammars for Expressions

1. In ANSI-C, assuming a=1 and b=2, what are the values of a, b and c after each of the following, and explain why:

```
a) a++, b++, c=a+b;
b) c= a+ ++b;
c) c= a++ +b;
d) c= a+++b;
e) c= a++ + ++b;
f) c= a+++++b;
```

2. In ANSI-C, what is the parse tree for, and value of, each of:

```
a) 1 + (2 + 3);
b) (1 + 2) + 3;
c) 1 + 2 + 3;
```

3. In ANSI-C, what is the parse tree for, and value of, each of:

```
a) 1 - (2 - 3);
b) (1 - 2) - 3;
c) 1 - 2 - 3;
```

4. Assuming that ^ means raising to a power (e.g. 3^4 = 3*3*3*3 = 81 = 0x51), what is the parse tree for, and hexadecimal value of, each of:

```
a) 2 ^ (3 ^ 3)
b) (2 ^ 3) ^ 3
c) 2 ^ (3 * 3)
d) 2 ^ 3 ^ 3
```

5. In ANSI-C, what is the parse tree for, and value of, each of:

```
a) (1 + 2) * (3 + 4);
b) 1 + (2 * 3) + 4;
c) ((1 + 2) * 3) + 4;
```

```
d) 1 + 2 * 3 + 4;
```

6. Assuming an input expression "1-2-3", what parse tree is produced by each of the following Yacc grammars:

```
a) exp : number | number '-' number ;
b) exp : number | exp '-' number ;
c) exp : number | number '-' exp ;
```

7. Assuming an input expression "1+2*3+4", what parse tree is produced by each of the following Yacc grammars, assuming number is as defined in the example in the lectures:

```
a) exp : number | number '+' number | number '*' number ;
b) exp : term | exp '+' term ;
   term : number | term '*' number ;
c) exp : term | exp '*' term ;
   term : number | term '+' number ;
d) exp : number | exp '+' number | exp '*' number ;
e) exp : number | exp '+' exp | exp '*' exp ;
```

8. We used this lex pattern to recognise numbers: [0-9]+
   What is the equivalent BNF, if we use yacc to recognise numbers?

Next | Up | Previous | Contents

Next: Semantic Analysis - Dictionaries **Up:** ho **Previous:** Lexical analysis - Lex **Contents**

Pete Jinks 2001-02-21

Answers to CS5031 Exercises 3: Grammars for Expressions

1. In ANSI-C, assuming a=1 and b=2, what are the values of a, b and c after each of the following, and explain why:

```
                          a       b       c
a) a++, b++, c=a+b;       2       3       5
b) c= a+ ++b;             1       3       4
c) c= a++ +b;             2       2       3
d) c= a+++b;              as (c)
e) c= a++ + ++b;          2       3       4
f) c= a+++++b;            illegal
```

The compiler's lexical analyser will try to recognise "++" and "+" operators before handing them over to its syntactic analyser, so it just dumbly tries to piece together as many "++" operators as it can (lex always tries to recognise the longest string it can). The syntactic analyser doesn't have a chance to make sense of "+++++", even though its only legal meaning is "++ + ++", as the lexical analyser has already recognised it as "++ ++ +" which is illegal in C. For similar reasons, (d) is treated as being the same as (c).

2. In ANSI-C, what is the parse tree for and value of each of:

```
a) 1 + (2 + 3);
b) (1 + 2) + 3;
c) 1 + 2 + 3;             as (b)
```

3. In ANSI-C, what is the parse tree for and value of each of:

```
a) 1 - (2 - 3);           2
b) (1 - 2) - 3;           -4
c) 1 - 2 - 3;             as (b)
```

4. Assuming that ^ means raising to a power (e.g. 3^4 = 3*3*3*3 = 81 = 0x51), what is the parse tree for and hexadecimal value of each of:

```
a) 2 ^ (3 ^ 3)            2 ^ 27 = 0x8000000
b) (2 ^ 3) ^ 3            2 ^ 9 = 0x200
c) 2 ^ (3 * 3)            2 ^ 9 = 0x200
d) 2 ^ 3 ^ 3              as (a)
```

5. In ANSI-C, what is the parse tree for and value of each of:

```
a) (1 + 2) * (3 + 4);     21
b) 1 + (2 * 3) + 4;       11
c) ((1 + 2) * 3) + 4;     13
d) 1 + 2 * 3 + 4;         as (b) = (1 + (2 * 3)) + 4;
```

6. Assuming an input expression "1-2-3", what parse tree is produced by each of the following Yacc grammars:

```
a) exp : number | number '-' number ;   stops after 1-2
b) exp : number | exp '-' number ;      as 3b
c) exp : number | number '-' exp ;      as 3a
```

7. Assuming an input expression "1+2*3+4", what parse tree is produced by each of the following Yacc grammars, assuming number is as defined in the example in the lectures:

```
a) exp : number | number '+' number | number '*' number ;        stops after 1+2
b) exp : term | exp '+' term ;                                   as 5b
   term : number | term '*' number ;
c) exp : term | exp '*' term ;                                   as 5a
   term : number | term '+' number ;
d) exp : number | exp '+' number | exp '*' number                as 5c
e) exp : number | exp '+' exp | exp '*' exp ;                    ambiguous, so unusa
```

8. We used this lex pattern to recognise numbers: [0-9]+
What is the equivalent BNF, if we use yacc to recognise numbers?

```
number : digit | number digit ;
digit : '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0' ;
```