YACC & LEX

OR ALTERNATIVELY, BISON AND FLEX

COMPILERS

- CppCon Talk on Compilation
- Compiler Explorer

SO YOU WANT TO BUILD YOUR OWN LANGUAGE….

- You’ll need a compiler or interpreter if it’s a high-level (anything above assembly) language!
- It’ll probably need to be easily parsed
- Should have reserved words (If, while, for… or something like that)
- Regular grammar - The language follows a CFG, basically
- Probably needs one symbol lookahead
- "If" shouldn’t just be "If", it should be "If something"

BUILD A COMPILER!

- A compiler allows you to turn a language into machine code.
- It needs:
  - Symbol table (a data structure storing information about identifiers (like variable names, function names, etc.). A reference point for the compiler to know the symbol's type, scope, and memory location)
  - Parser (to check whether the sequence of tokens follows the grammar of the language.)
  - Syntax analysis – Probably should be simple
  - Code generation – into machine code
  - Optimizer
  - Activation records – Not for itself, it needs to be able to build them.

OR, BUILD AN INTERPRETER!

- Interpreters interpret code into machine code at runtime.
- Has a syntax checker, parser, compiler & interpreter!
- Uses a symbol table in the interpreter.
- Not optimized.
- Uses interpreted activation records stored in RAM dynamically.

COULD YOU IMAGINE DOING THIS BY HAND?

- Not exactly efficient.
- Extremely complicated, especially writing the machine code compilation system.
- Have to keep a handle on all the activation records.
- Need to invent your own way of doing function context switching…

YACC AND LEX, THE BETTER WAY.

- **Yacc:** Yet Another Compiler Compiler
- Unix compiler compiler
- A parser generator tool widely used in compiler construction
- Converts formal user grammar specification using C into a grammar parser
- **Lex:** Short for Lexical Analyzer
- Unix regular expression string matcher generator
- Converts a series of rules into a lexer program in C
- These are basically dead now because Unix is old. We use the Linux versions:
  - GNU Bison
  - BSD Flex (short for “Fast Lex”)

LEX - THE LEXICAL ANALYZER

What is Lex?

- Problem-oriented specification for string matching.
- You give it a specification, and it produces a language that recognizes only regular expressions.
- This can do a lot of cool stuff but can’t be a compiler on its own (you need Yacc for that).
- Each matched string invokes some user code (all in C, so I hope you like C).
- Always chooses the longest match.
- Might need way more than one character lookahead.

BASIC LEX EXAMPLE

Lex programs by convention are stored in `.l` files.

- To compile Lex code, first you need to use `lex` or `flex`.
- Example:

```
lex somekindaprog.l
gcc -o something lex.yy.c -lfl
```

OR, if using lex and not flex:

```
gcc -o something lex.yy.c -ll
```

To run:

```
./something < input.txt > output.txt
```

LEX EXAMPLE: PATTERN MATCHING

Example of input and Lex rules:

```
Input - abcdefh
Rules - ab > abcdefh
```

Lex will find “ab” and start matching again from “c”… and then it won’t find anything else.

MORE LEX INFO

Lex.yy.c contains a subprogram yylex()

- By default:
    - Lex uses a default `main()`
    - Reads from `stdin`
    - Writes to `stdout`
- You can call `yylex` from your own code!
- Automatically incorporates itself into Yacc code.
- Has other functions too:
    - `yywrap`, for example, handles multiple files.
- However, most distributions of Linux do not provide all definitions (e.g., `yywrap` and `yyerror`), so you may have to find them yourself on the web.

LEX EXAMPLES

Replaces all blanks and tabs at the end of each line.

```
%%
[\t ]+$ ;
%%
```

LEX EXAMPLES

Replaces multiple blanks by a single blank. Also removes end-of-line spaces.

```
%%
[\t ]+$ ;
[\t ]+  printf (" ");
%%
```

Scans for both rules at the same time. The end-of-line statement is longer than just the spaces, so it has priority.

LEX EXAMPLE: SEARCHING FOR A WORD

Searches for the word "integer" and prints "found INT"

```
%%
integer  printf ("found INT");
%%
```

LEX FORMAT

Lex generates a deterministic finite automaton from the regular expressions in its source.
The full format for a Lex file is:

```
{definitions} //optional
%%
{rules}       //optional
%%
{subroutines} //optional
```

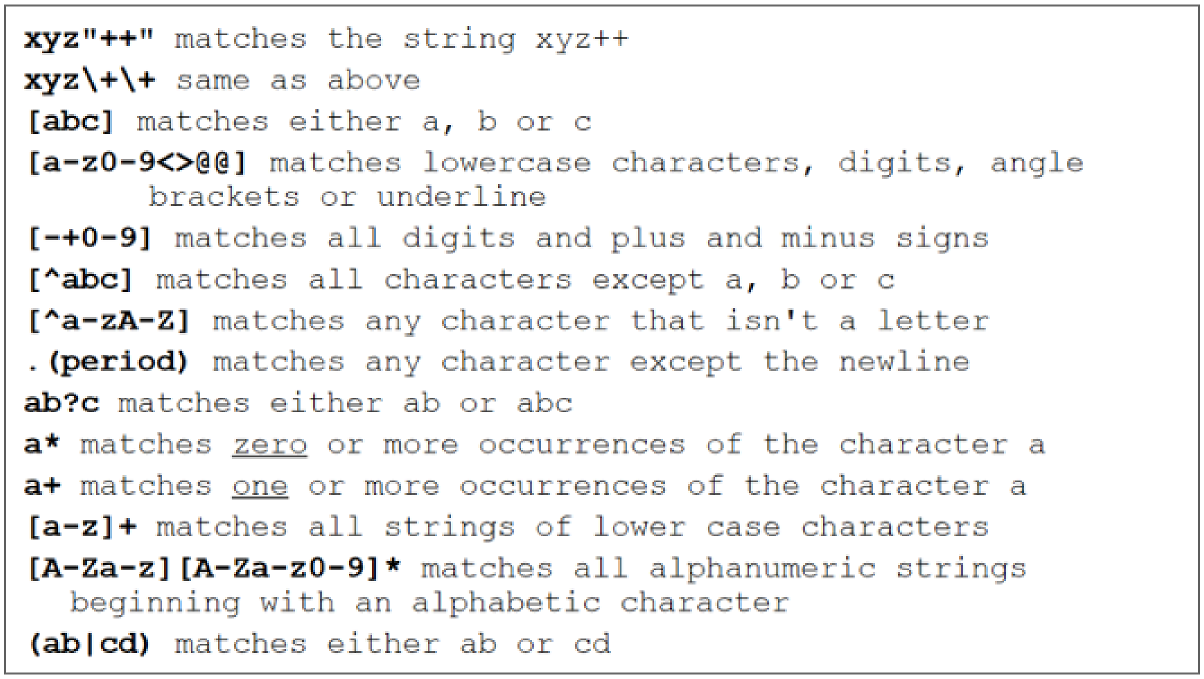
The minimal Lex program is just %%, which copies input to output.

LEX REGULAR EXPRESSIONS

Lex uses regular expressions to define all rules.
These are almost exactly the same as those used in Unix.
Lex was built for Unix, so this isn't surprising!
Lex uses the following operators:

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

LEX REGEX EXAMPLES



xyz"++" matches the string xyz++

xyz\+\+ same as above

[abc] matches either a, b or c

[a-z0-9<>@@] matches lowercase characters, digits, angle brackets or underline

[-+0-9] matches all digits and plus and minus signs

[^abc] matches all characters except a, b or c

[^a-zA-Z] matches any character that isn't a letter

.(period) matches any character except the newline

ab?c matches either ab or abc

a* matches zero or more occurrences of the character a

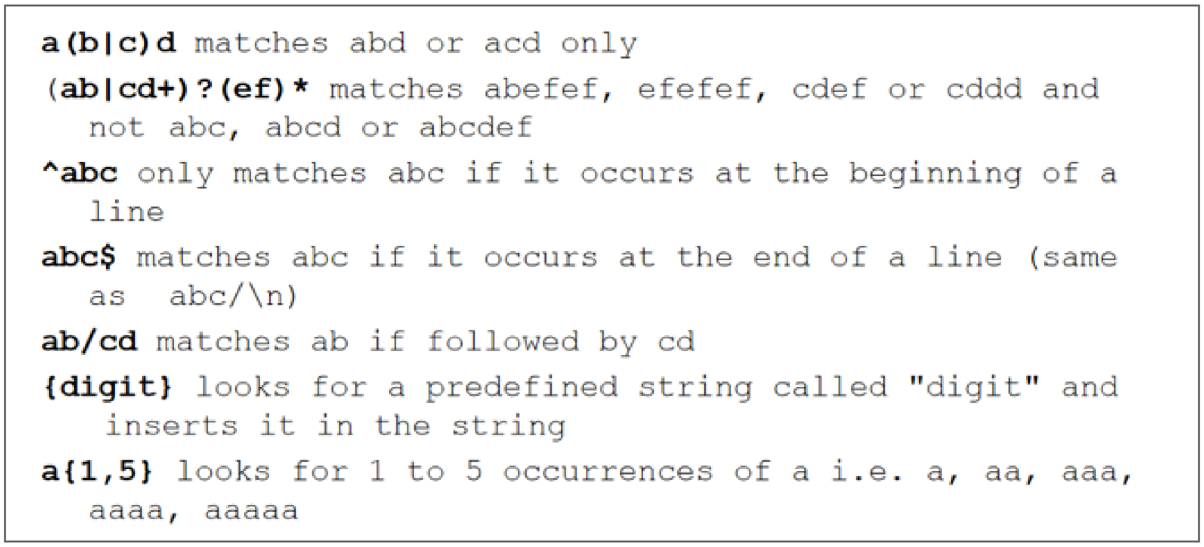
a+ matches one or more occurrences of the character a

[a-z]+ matches all strings of lower case characters

[A-Za-z][A-Za-z0-9]* matches all alphanumeric strings beginning with an alphabetic character

(ab|cd) matches either ab or cd

LEX REGEX EXAMPLES



`a(b|c)d` matches abd or acd only

`(ab|cd+)?(ef)*` matches abefef, efefef, cdef or cddd and not abc, abcd or abcdef

`^abc` only matches abc if it occurs at the beginning of a line

`abc$` matches abc if it occurs at the end of a line (same as abc/\n)

`ab/cd` matches ab if followed by cd

`{digit}` looks for a predefined string called "digit" and inserts it in the string

`a{1,5}` looks for 1 to 5 occurrences of a i.e. a, aa, aaa, aaaa, aaaaa

LEX REGEX EXAMPLES

Example: Ignoring spaces, tabs, and newlines.

```
%%
[ \t\n] ;
```

OR, using an alternative method:

```
%%
" " | \t  | \n  ;
```

LEX EXAMPLE: PRINTING MATCHED PATTERNS

You can print out what a pattern has matched using the variable `yytext`.

```
%%
[a-z]+   printf ("%s", yytext );
```

OR, using an alternative method:

```
%%
[a-z]+  ECHO;
```

LEX EXAMPLE: COUNTING CHARACTERS AND WORDS

You can output the length of a matched string using `yyleng`.
You can also use `yyless` (not pictured here) if supported to wind characters back onto the input stream one at a time.
Using both, you can actively edit the input stream!
Lex will only store 100 characters, though… so be aware.

```
%%
[a- zA -Z]+ { words++; chars +=  yyleng ; }
%%
```

AMBIGUOUS RULES IN LEX

If two rules match, Lex chooses the longer match.
If they’re the same length, Lex chooses the one that was defined first.
Example:

```
%%
integer   // it’s an int!
[a-z]+   // it’s an identifier!
%%
```

Here, "integer" is matched first because it comes before the generic identifier rule.

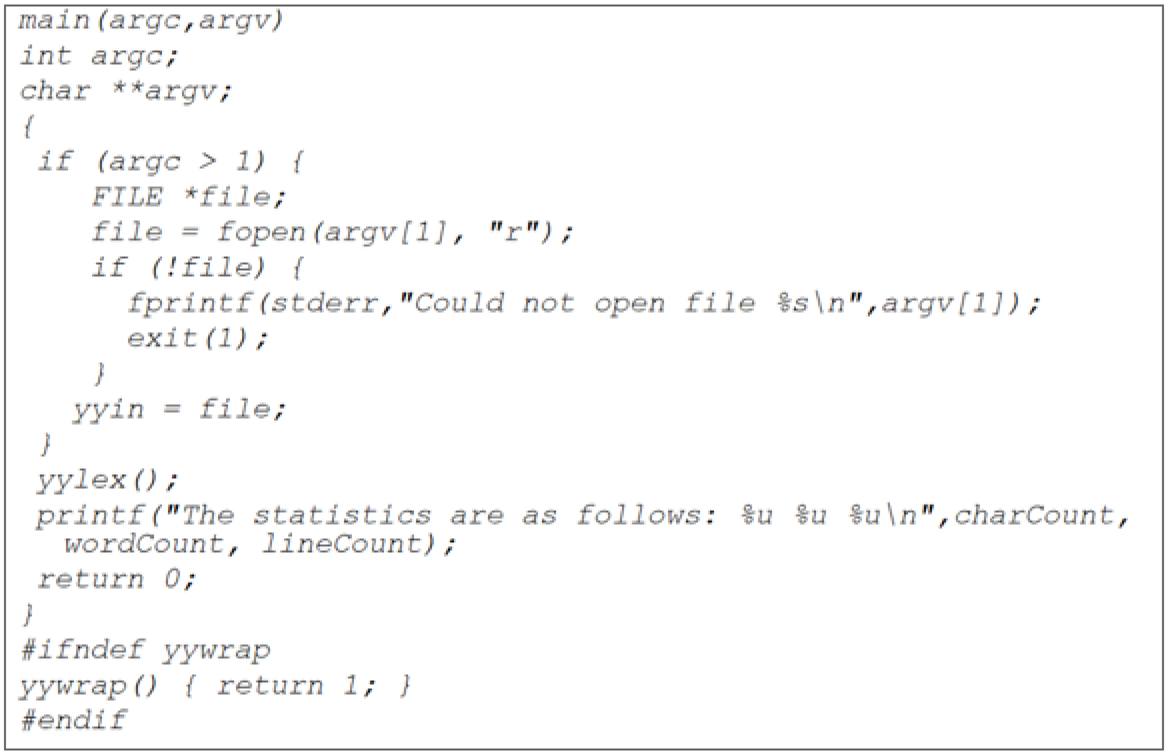
LAST WORDS ON LEX

- Lex will convert your code into C.
- Since Lex is context-sensitive analysis, preprocessor statements need to be handled differently than in C.
- We can’t just use `#define` statements.
  Two methods to handle this:
- **Use of Flags:** When only a few rules change from one context to another – user-defined.
- **Use of Start Conditions:** Similar but allows Lex to define the code required.

EXAMPLE LEX PROGRAM

```
Example lex program to extract basic word statistics for a user
   specified file - based on example from “Unix programming
   tools: Lex and Yacc”, John Levine, 1992, O’Reilly, pages 32-
   35.

%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

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

%%

{word} { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
. charCount;

%%
```

EXAMPLE LEX PROGRAM



```
main(argc,argv)
int argc;
char **argv;
{
 if (argc > 1) {
    FILE *file;
    file = fopen(argv[1], "r");
    if (!file) {
      fprintf(stderr,"Could not open file %s\n",argv[1]);
      exit(1);
    }
   yyin = file;
 }
 yylex();
 printf("The statistics are as follows: %u %u %u\n",charCount,
   wordCount, lineCount);
 return 0;
}
#ifndef yywrap
yywrap() { return 1; }
#endif
```

EXAMPLE LEX PROGRAM

This Lex program scans input and performs specific actions.

```
%%
[a-z]+   printf("Matched: %s\n", yytext);
[0-9]+   printf("Number: %s\n", yytext);
%%
```

EXAMPLE LEX PROGRAM (PT.2)

This Lex program replaces spaces and tabs with a single space.

```
%%
[\t ]+  printf(" ");
%%
```

INTRODUCTION TO YACC

- YACC: Yet Another Compiler Compiler
- Converts user-defined grammar into a parser
- Generates a function `yyparse()` (written in C)
- Works together with Lex

YACC GRAMMAR OVERVIEW

- Uses LALR(1) parsing (Look-Ahead Left-Right, one symbol lookahead)
- Specification includes:
  - Input processing
  - Rules describing input structure
  - Code execution when rules match
- YACC processes tokens identified by Lex

YACC GRAMMAR

Rules in YACC invoke actions when a match is found. Example:

```
date: month_name day ',' year;
```

This rule matches the format "March 3, 1992".

DEFINING TOKENS IN YACC

YACC allows defining tokens to match Lex definitions.

```
%token name1 name2 ...
%start symbol
```

Tokens represent lexical units like identifiers, numbers, operators, etc.

YACC FORMAT

The structure of a YACC file:

```
{declarations}      // Optional
%%
{rules}             // Required
%%
{programs}          // Optional
```

ESCAPE SEQUENCES IN YACC

YACC supports C-style escape sequences.

```
'\n' - newline
'\r' - carriage return
'\t' - tab
'\b' - backspace
'\x01' - hex-defined character
```

Note: The null terminator `'\0'` is not allowed.

DEFINING RULES IN YACC

Rules describe how tokens combine to form higher-level structures.

```
A: B C D;
A: E F;
A: G;
```

OR, using an alternative syntax:

```
A: B C D | E F | G;
```

A rule may contain something, or be empty. Exactly one empty can exist in the entire rule set.

```
example: ;
```

DEFINING ACTIONS IN YACC

Actions define what happens when a rule matches.

```
A: '(' B ')' { hello(1, "abc"); }
X: Y Z { printf("Message\n"); flag = 25; }
```

EXAMPLE: A CALCULATOR IN YACC

This YACC program defines basic arithmetic operations.

```
%token NUMBER
%%
expr: expr '+' term | expr '-' term | term;
term: term '*' factor | term '/' factor | f
factor: '(' expr ')' | NUMBER;
%%
```

EXAMPLE: A CALCULATOR IN LEX

Lex rules to tokenize numbers and operators.

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

COMPILING AND RUNNING

To compile and run a YACC and Lex program:

```
yacc -d calc.y
lex calc.l
gcc -o calc y.tab.c lex.yy.c -lfl
./calc
```

HOW YACC PARSES INPUT

- Uses a **stack** to track symbols
- Parses left-to-right
- Supports four key operations:
    - **Shift** - Pushes symbol onto stack
    - **Reduce** - Applies a rule and reduces
    - **Accept** - Successfully parsed input
    - **Error** - Invalid syntax

RECURSIVE PARSING EXAMPLE

YACC can handle recursive grammars:

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

AMBIGUITY IN YACC

Ambiguous rules require additional handling.
Example: Parsing 2+3*4

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

This ensures multiplication has higher precedence than addition.

AMBIGUITY: REDUCE/REDUCE CONFLICT

Occurs when a phrase can be reduced in multiple ways. Example:

```
word: sequence | maybeword;
```

Solution: Ensure only one empty rule exists.

FIXING SHIFT/REDUCE CONFLICTS

YACC allows specifying operator precedence:

```
%left '+' '-'
%left '*' '/'
%right '='
```

This ensures correct evaluation order.

LEX AND YACC TOGETHER

- Lex scans input and returns tokens.
- YACC parses and processes tokens.
- They work together to build compilers and interpreters.

DEBUGGING LEX AND YACC

- Use **verbose mode** in YACC to debug parsing issues.
- Add **printf statements** in Lex to track tokens.
- Check YACC's generated parse table for conflicts.

EXAMPLE: PARSING AN IF STATEMENT

A simple if-statement in YACC:

```
if_stmt: 'if' '(' condition ')' '{' stateme
```

HANDLING LOOPS IN YACC

Example of while and for loops:

```
loop_stmt: 'while' '(' condition ')' '{' st
         | 'for' '(' init ';' condition ';'
```

ADVANCED YACC FEATURES

- Multiple start conditions
- Context-sensitive parsing
- Error handling with `yyerror()`

OPTIMIZING LEX & YACC CODE

- Use **efficient regex patterns** in Lex.
- Define **precise precedence rules** in YACC.
- Eliminate **ambiguous rules** to avoid conflicts.

REAL-WORLD APPLICATIONS OF LEX & YACC

- Used in compilers like GCC.
- Common in interpreters and parsers.
- Also used in configuration file processing.

THANK YOU!

Any questions?