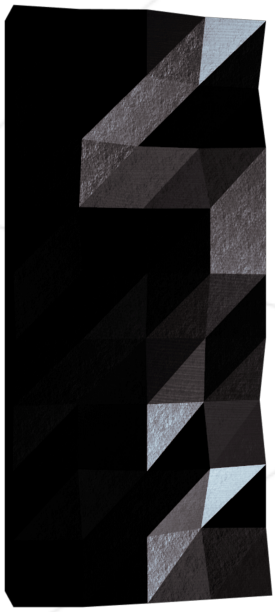
β

This project is currently under beta-test, until it is validated by a fair number of students in the 42 community. Please report any typo, incoherence, inconsistency, error, using the form https://tally.so/r/3lVDJo



ft_lex

easy lexing

Summary: You probably developed lexers (or tokenizers) in previous projects. Now, let's create a generic lexer to use in all your projects!

Version: 1.00

Contents

Chapter I

Forewords

— Would you like a drink? asked Colin. My pianocktail is finished, you could try it out.
— It works? asked Chick.
— Perfectly. I had trouble getting all the bugs out but the results go beyond my expectations. I got a truly astounding mix out of Black and Tan Fantasy.
— How did you make it work? asked Chick.
— With every note, said Colin, I've matched a spirit, liqueur or flavoring. The loud pedal corresponds to whipped egg and the soft pedal to ice. For seltzer water, you need to do a trill in the upper register. The quantities are in direct proportion with the duration: the 64th note equals a 16th part, a quarter note one part and a whole note four parts. When playing a slow tune, a leveling system is put to work so that the quantity is not increased—that would make for too abundant a cocktail—only the alcohol content. And, depending on the length of the tune, the part's valence can be changed, reducing it for example to one one-hundredth to get a drink that takes into account all of the harmonies by means of a lateral regulator.
— That's complicated, said Chick.
— Everything is controlled by electrical contacts and relays; I won't give you the details, you know all that. And besides, what's more, the piano really works.
— That's marvelous! said Chick.
— There's only one problem, said Colin. The loud pedal for the whipped egg. I had to put in a special system of interlocking parts because when you play a tune that's too "hot," pieces of omelet fall into the cocktail and it's hard to swallow. I'll modify that. For the time being, you just need to be careful. For the sour cream, it's low G.
— I'm going to make myself one on Loveless Love, said Chick. It'll be great.
— It's still in the junk room that I turned into a workshop, said Colin, because the protection plates aren't screwed in. Come on, let's go. I'll set it for two cocktails of about twenty centiliters, to start off with.
Chick sat down at the piano. At the end of the tune, part of the front panel opened up with a clap and a row of glasses appeared. Two of them were filled to the rim with an appetizing mixture.
— You scared me, said Colin. At one point you hit a wrong note. Luckily, it was in harmony.
— It accounts for the harmony? said Chick.
— Not all the time, said Colin. That would be too complicated. There are only a few constraints. Drink and come eat.

Boris Vian *L'Écume des jours*

The pianocktail could be an example of a finite-state transducer, its input alphabet is the set of all the key of the piano and its output alphabet is the set of all the cocktail ingredients.

Chapter II

Objectives

In this project, you will implement the POSIX lex utility.

During this project, you will learn about automata theory, formal language theory, Chomsky hierarchy, dragons and a lot of weird mathematical symbols and Greek letters.

II.1 Description

Lex is a program that generates lexical analyzers (aka scanners or lexers or tokenizers).

It takes a .l file (a lex-specific format) as input and outputs a lex.yy.c file containing the C source code that represent the lexer.

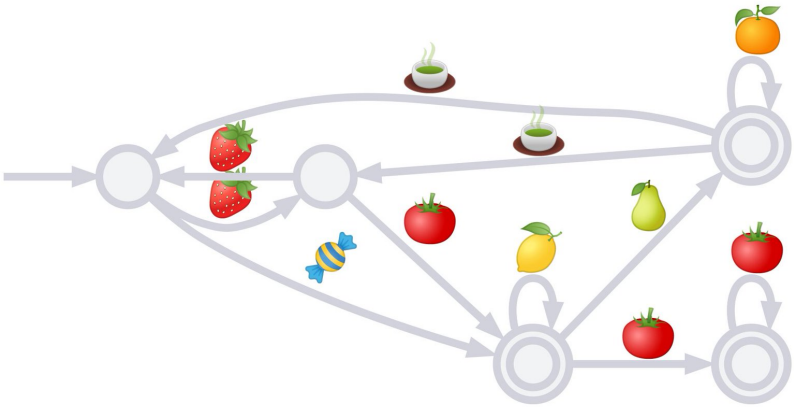
Lexical analysis is the process of converting a sequence of characters into a sequence of lexical tokens.

This step is generally combined with syntactic analysis: the process of converting a sequence of tokens into a data structure (often an Abstract Syntax Tree).

This is why lex is typically combined with yacc, another utility that will generates a parser (this is the goal of the next project in this branch `ft_yacc`).

II.2 Example

```
$> cat scanner.l
%%

[0-9]+ printf("NUMBER: %s\n", yytext);

"+" |
"-" |
"*" |
"/" printf("OPERATOR: %s\n", yytext);

"(" printf("OPEN PARENTHESIS\n");

")" printf("CLOSED PARENTHESIS\n");

"\n" printf("NEWLINE\n");

[[:blank:]] // ignore whitespaces

.   printf("Invalid character: %c\n", *yytext);

$> lex scanner.l && clang -o scanner lex.yy.c -ll
$> echo "42+1337+(21*19)" | ./scanner
NUMBER: 42
OPERATOR: +
NUMBER: 1337
OPERATOR: +
OPEN PARENTHESIS
NUMBER: 21
OPERATOR: *
NUMBER: 19
CLOSED PARENTHESIS
NEWLINE
```



A NFA recognizing the regex ((🍓🍓)*(🍓🍎|🍬)🍋*🍋🍊*🍵🍓?)*(🍓🍓)*(🍓🍎|🍬)🍋*(🍎*|🍋🍊*)
Generated by the bot @happyautomata made by @thingskatedid

Chapter III

General rules

- This project will only be evaluated by actual human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements listed below.
- You are allowed to write your program in any of the following languages:
  - C
  - C++
  - Java
  - Zig
  - Rust
  - Caml/OCaml
  - Any **purely functional** language
  - Assembly (please don't)
- In all cases, you are only allowed to use the standard library.
- The executable file must be named `ft_lex`.
- The libl must be named libl.
- If you use a compiled language, you must submit a Makefile or a configuration for the native build system of your language.
- You have to handle errors in a sensitive manner. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- If a user error occurs (e.g., invalid regex in the .l file), your program must provide a detailed error message (e.g., `lexer.l:12 unexpected token ')'`)

Chapter IV

Mandatory part

IV.1 The program

- You must implement the lex utility as described by POSIX.1-2024 with all its options and functionalities.
- Therefore, you must also implement the libl (a small utility library) as specified by POSIX and write a makefile to compile it.
- You may choose any language from the list above, but the target language must be C.
- The output C file must contain a compiled **deterministic** finite automaton.
- In the output file (and in the libl), you may only use the following functions:
  - all the functions from `<stdio.h>`
  - all the functions from `<string.h>`
  - `malloc(3)`
  - `realloc(3)`
  - `calloc(3)`
  - `free(3)`

Flex can be useful if you don't understand the documentation and need to experiment by yourself.

Sometimes, POSIX say that something have an unspecified behaviour, (for example using another locale than POSIX) This means that you are not required to handle these cases, but your program must not crash. I advise you to read the POSIX terminology here

```
man dot
```

IV.2 Testing

You must provide example .l files, as well as files to parse, that cover all functionalities of your ft_lex.

Chapter V

Bonus part

V.1 Polyglotism

You've done the difficult part, now let's add another target language to your ft_lex, this can be any language!
For the input file format you can take inspiration from the existing implementations (alex, golex, jlex, ocamllex...) or invent a new one if you prefer.
You must add a flag to your program to switch the target language.

If you plan to realize cc1 later, think about the language that you will use to make it.

Obviously, if you create a C++ version, you can't simply reuse the C code be creative.

V.2 Compression

As you can see, a deterministic finite automaton can be very very heavy (If you are not convinced, try this regex: '`(a{0,10000}){0,10000}`').

For this bonus you must add some sort of compression algorithm to your program that must really lighten your DFA, we're talking about a 2x difference (for example try with `flex -t lexer.l | wc` and `flex -t -Cf lexer.l | wc`).

Add a flag to your program that will enable this feature.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VI

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.