NTNU – Trondheim
Norwegian University of
Science and Technology

Lexical analysis roundup

What we have done

- Described regex
- Converted regex → NFA
- Converted NFA → DFA
- Minimized DFA
- Simulated DFA
- Suggested that creating the simulator can be left to a scanner-generator program

The original

- In the beginning, there was one called *Lex* which wrote scanners in C
- Its format and idea is sort of a template for a whole family tree of successors

  flex (still targets C, companion to GCC, we'll take it)

  JFlex (Java)

  PLY (Python)

  C# Flex (take a guess)

  Alex (Haskell)

  gelex (Eiffel)

  ...

Specification format

- Lex files are suffixed *.l , and contain 3 sections

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- Declaration and function sections can contain regular C code that makes its way into the final product
- Translation rules are compiled into a function called `yylex()`
- The output is a C file you can read if you like

Declarations

- The declaration section also admits some directives to Lex itself, so any C you wish to include is contained between %{ and %}
- The auxiliary functions section is just plain ol' source code
- The translation rules are regular expressions paired with basic blocks (actions)

As an example

- We can define some regex without attaching much of a language

    [\n\t\v\ ]

    if

    then

    endif

    end

    [0-9]+

Reacting to matched text

- We can attach actions to take on match

```
[\n\t\v\ ]    { /* Do nothing, this is whitespace */ }
if            { return IF; }
then          { return THEN; }
endif         { return ENDIF; }
end           { return END; }
[0-9]+        { return INT; }
```

NTNU – Trondheim
Norwegian University of
Science and Technology

That needs token definitions

```
%{
    #include <stdio.h>
    enum { IF, THEN, ENDIF, INT, END };
%}
%%
[\n\t\v\ ]    { /* Do nothing, this is whitespace */ }
if            { return IF; }
then          { return THEN; }
endif         { return ENDIF; }
end           { return END; }
[0-9]+        { return INT; }
```

This is plain C

NTNU – Trondheim
Norwegian University of
Science and Technology

It won’t run without a main function

```
(defs)
%%
(rules)
%%
int main () {
    int token = 0;
    while ( token != END)  {
        token = yylex();
        switch ( token ) {
            case IF: printf ( "Found if\n" ); break;
            case THEN: printf ( "Found then\n" ); break;
            case ENDIF: printf ( "Found endif\n" ); break;
            case INT: printf ( "Found integer %s\n", yytext ); break;
            case END: printf ( "Hanging up... bye\n" ); break;
        }
    }
}
```

Call the generated scan function

Do something with each token

Lex can stand alone

- If you have a simple program that just needs a scanner, and you miss regex, it can fit in a Lex specification
- I’ve put the examples online, we can run them

Lex can talk about states

- Some things are easier if you can name a sub-automaton and treat it separately
- Strings come to mind, all the things you can put between “ and ” make a foofy regex
  - Putting

    %state STRING

    in the declarations section let you talk about a state called that
  - Specifying

    \<character> let you anticipate one symbol ahead without matching it away from the input (*lookahead*)

Talking about states

- Using those mechanisms, named states can appear in the translation rules

Set state

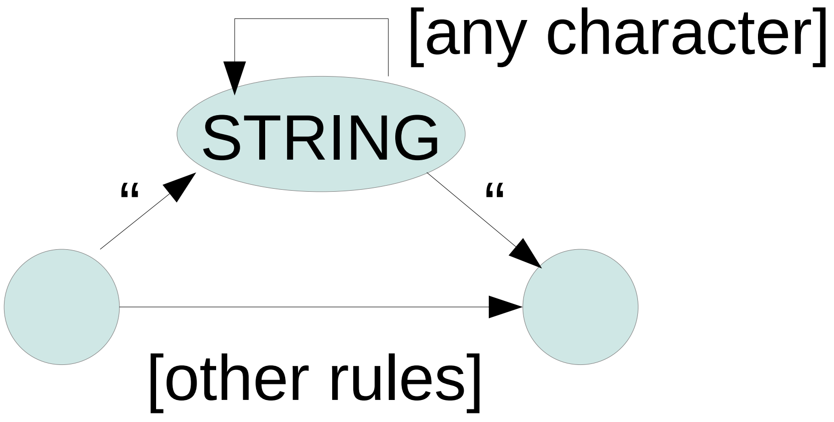
```
<INITIAL>if      { printf ( "Found 'if'\n" ); }
<INITIAL>end { printf ( "Found 'end'\n" ); return 0; }
<INITIAL>\"     { printf ( "Found string: " ); BEGIN(STRING); }
<STRING>\"     { printf ( "\n" ); BEGIN(INITIAL); }
<STRING>.      { printf ( "%c,", yytext[0] ); }
```

Stop before next “

Match any character (regex. extension ‘.’ matches anything)

This introduces a sub-automaton

- Something along these lines:



Lex can interface with other code

- Specifically, it pairs well with YACC
  (Yet Another Compiler-Compiler)
- YACC generates syntax analyzers (our next topic)
  - It can define tokens for Lex specifications to use
  - It knows to call yylex for the next token
- That is how we will make use of the two together

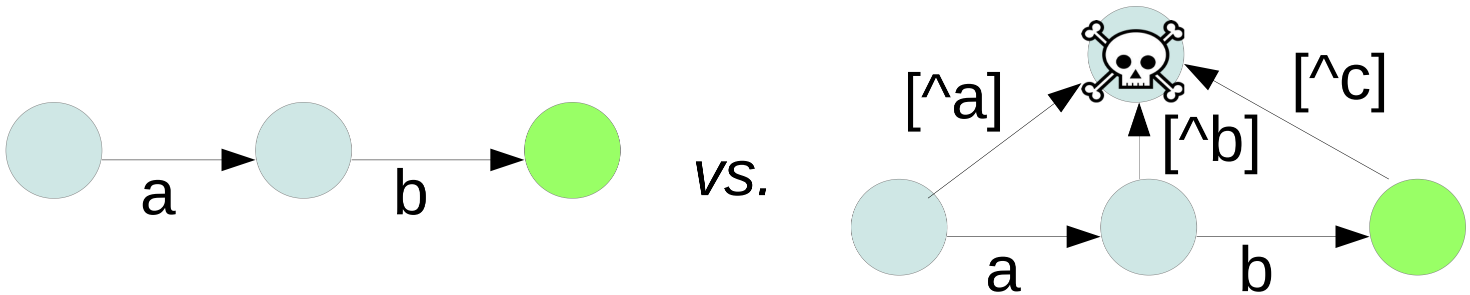
NTNU – Trondheim
Norwegian University of
Science and Technology

Bits and bobs we skipped in chapter 3: Longest match

- When there are multiple accepting states, the DFA simulation can’t guess whether to take the first match, or continue in the hope of finding another
- Common rule is that the longest match wins, and the input-recording buffer rolls back if input leads the DFA astray

Bits and bobs we skipped in chapter 3: Dead states

- Technically, every DFA state goes somewhere on every symbol
- You can trap it in a state that doesn't accept, and transitions to itself on every symbol
- It messes up the drawings (which we want because they're clear):



- It's a detail that matters more to scanner generator *authors* than to *users*, but you can read about it.

Bits and bobs we skipped in chapter 3: Direct regex → DFA translation (3.9.1-3.9.5)

- This method has a touch of syntax analysis to it
- We're going to spend quite enough time on syntax analysis, and I think the relevant principle comes through more clearly there
- You can look at it for continuity, and even return to it after we've done LL(1) parsers
  - I'm not going to bug you about the details of this algorithm
  - You should know that it exists, and converts regex to DFA

That’s a wrap

- *Onward, to the charms of syntactic analysis!*