LEX1: Intro to Regexps

Lexical Analysis

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

Lexical Analysis

Also called *lexing* or *scanning*, take input program *string* and convert into *tokens*

Example

```
double f = sqrt(-1);
```

```
T_DOUBLE        (“double”)
T_IDENT         (“f”)
T_OP            (“=”)
T_IDENT         (“sqrt”)
T_LPAREN        (“(”)
T_OP            (“-”)
T_INTCONSTANT   (“1”)
T_RPAREN        (“)”)
T_SEP           (“;”)
```

Token Attributes

- Some tokens have attributes:
  - `T_IDENT (“sqrt”)`
  - `T_INTCONSTANT (“1”)`
- Other tokens do not:
  - `T_WHILE`
- Source code location for error reports
- A token is defined using a **pattern**.
- The **pattern** for identifiers: a sequence of one or more letters, digits and underscores which starts with a letter or underscore.

`T_IDENT`	`(“sqrt”)`
Token	Lexeme

Lexical errors

The lexer does not check for syntax errors!

- What if user omits spaces: `doublef=sqrt(-1);`
  - No lexical error!
  - Single token is produced: `T_IDENT`(“`doublef`”)
  - Not two tokens: `T_DOUBLE`, `T_IDENT`(“`f`”)
- Typically few lexical error types
  - Illegal chars
  - Unclosed string constants
  - Comments that are not terminated correctly

Lexical errors

- Lexical analysis should not disambiguate tokens
  - e.g. unary operator – (minus) versus binary operator – (minus)
  - Use the same token `T_MINUS` for both
  - It’s the job of the parser to disambiguate based on the context

Q: Using the same token definitions as before, provide the sequence of token(s) that will be produced for input `double(-1)`

Ad-hoc Lexer

Implementing Lexers: Loop and switch scanners

- Big nested switch/case statements
- Lots of getc()/ungetc() calls
  - Buffering and streams; Sentinels for push-backs
- Can be error-prone
- Changing or adding a keyword is problematic

Read source of an ad-hoc lexer: LexTokenInternal in clang

Implementing Lexers: Loop and switch scanners

- Does the implementation exactly capture the language specification?
- How can we show correctness?
- **Key idea**: separate the definition of tokens from the implementation
- **Problem**: we need to reason about patterns and how they can be used to define tokens (recognize strings).

Specifying Patterns using Regular Expressions

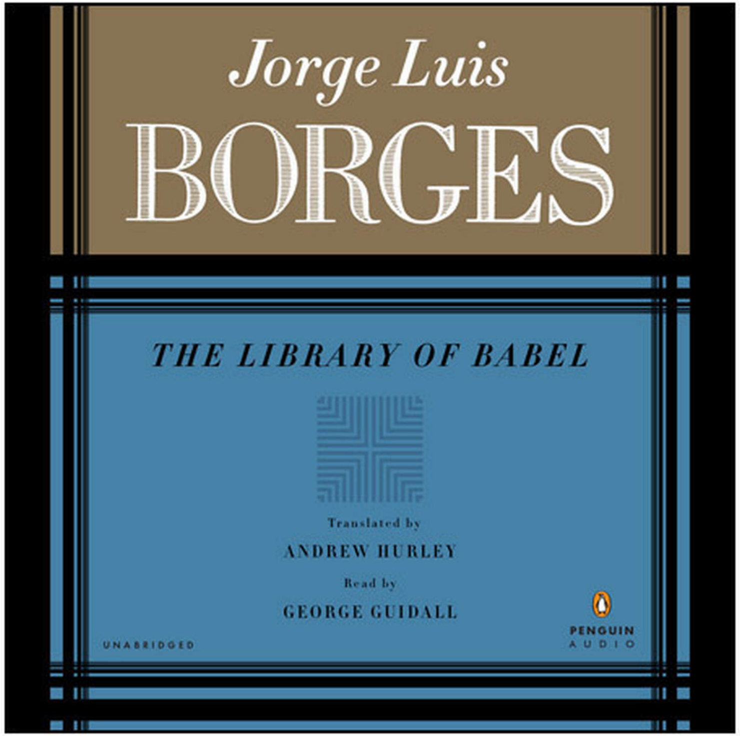
Formal Languages: Recap

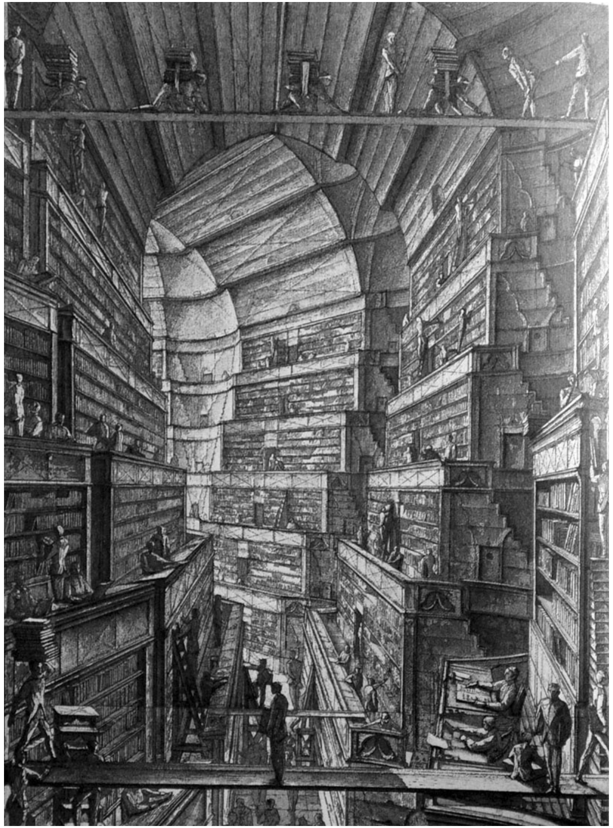
- Symbols (each of length one): a, b, c
- Alphabet : finite set of symbols $\Sigma = \{a, b\}$
- String: sequence of symbols (length = #symbols) bab or $a^2 = aa$
- Empty string (has zero length): ε
- Define: $\Sigma^\varepsilon = \Sigma \cup \{\varepsilon\}$
- Define: $\Sigma^0 = \{\varepsilon\}, \Sigma^1 = \{a, b\}, \Sigma^2 = \{aa, ab, bb, ba\}$
- Set of all strings: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots \cup \Sigma^n : n \to \infty$
- (Formal) Language: a set of strings $\{ a^n b^n : n > 0 \}$

All strings of length 0, 1, 2 using symbols from the alphabet Σ

Q: How many strings in Σ^n if the alphabet Σ has m elements.

The Library of Babel: Visualizing Σ^*





Regular Languages

Recursively defining the set of all regular languages:

1. The empty set and $\{a\}$ for all a in Σ^{ε} are regular languages
2. If L_1 and L_2 and L are regular languages, then:

   $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ (concatenation)

   $L_1 \cup L_2$ (union)

   $L^* = \cup_{i=0}^{\infty} L^i$ (Kleene closure)

   are also regular languages

3. There are no other regular languages

$\{a\}^*$ $\{\varepsilon, a, aa, \ldots$

L^*

$L^0 \cup L^1 \cup L^2 \ldots$

Regular Languages

- The set of regular languages: each element is a regular language
  - $R = \{R_1, R_2, \dots, R_n, \dots\}$
- Each regular language is an example of a (formal) language, i.e. a set of strings

  e.g. $\{a^m b^n \colon m > 0, n > 0\}$

Formal Grammars

- A formal grammar is a concise description of a formal language using a specialized syntax

- For example, a **regular expression** is a concise description of a regular language

    `(a|b)*abb` is the set of all strings over the alphabet {*a*, *b*} which end in *abb*

- We will use regular expressions (regexps) in order to define tokens in our compiler,
    - e.g. Python integers are defined as the pattern `[+-]?([1-9][0-9]*|0)`

any number from 1 to 9

zero or more numbers from 0 to 9

Regular Expressions: Definition

- Every symbol of $\Sigma \cup \{ \varepsilon \}$ is a regular expression (regexp)
  - If $\Sigma = \{a, b\}$ then a, b are regexps
- If r_1 and r_2 are regular expressions, combine them using:
  - Concatenation: r_1r_2, e.g. ab or aba
  - Alternation: $r_1|r_2$, e.g. a|b
  - Repetition: r_1*, e.g. a* or b*
- No other core operators are defined
- But other operators can be defined as combinations of the basic operators, e.g. a+ = aa*

Lex regular expressions

<table>
<tr><th>Expression</th><th>Matches</th><th>Example</th><th>Using core operators</th></tr>
<tr><td>c</td><td>non-operator character c</td><td>a</td><td></td></tr>
<tr><td>\c</td><td>character c literally</td><td>*</td><td></td></tr>
<tr><td>"s"</td><td>string s literally</td><td>"**"</td><td></td></tr>
<tr><td>.</td><td>any character but newline</td><td>a.*b</td><td></td></tr>
<tr><td>^</td><td>beginning of line</td><td>^abc</td><td>used for matching</td></tr>
<tr><td>$</td><td>end of line</td><td>abc$</td><td>used for matching</td></tr>
<tr><td>[s]</td><td>any one of characters in string s</td><td>[abc]</td><td>(a|b|c)</td></tr>
<tr><td>[^s]</td><td>any one character not in string s</td><td>[^a]</td><td>(b|c) $\Sigma = \{a, b, c\}$</td></tr>
<tr><td>r*</td><td>zero or more strings matching r</td><td>a*</td><td></td></tr>
<tr><td>r+</td><td>one or more strings matching r</td><td>a+</td><td>aa*</td></tr>
<tr><td>r?</td><td>zero or one r</td><td>a?</td><td>$(a|\varepsilon)$</td></tr>
<tr><td>r{m,n}</td><td>between m and n occurences of r</td><td>a{2,3}</td><td>(aa|aaa)</td></tr>
<tr><td>r_1r_2</td><td>an r_1 followed by an r_2</td><td>ab</td><td></td></tr>
<tr><td>$r_1|r_2$</td><td>an r_1 or an r_2</td><td>a|b</td><td></td></tr>
<tr><td>(r)</td><td>same as r</td><td>(a|b)</td><td></td></tr>
<tr><td>r_1/r_2</td><td>r_1 when followed by an r_2</td><td>abc/123</td><td>r_1r_2 used for matching</td></tr>
</table>

Limitations(?) of Regular Expressions

- Regexps can be used only if the language definition is sane
  - Should not permit crazy long-distance effects (e.g. Fortran)

```
DO 5 I = 1,5   ➡   T_DO T_INT(5) T_ID(I) T_EQ …
DO 5 I = 1.5   ➡   T_ID(DO 5 I) T_EQ T_FLOATCONST(1.5)
```