Regular Languages

Jeremy Bolton, PhD

Asst Teaching Professor

A VERY special thanks to the authors of THE dragon book and other contributors at Stanford

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

for	{
int	}
<<	;
=	<
(	[
)	]
++	
Identifier	
IntegerConstant	

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like `if` and `while` probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes:
  - All variable names, all possible numbers, all possible strings, etc.

Sets of Lexemes

- Idea: Associate a set of lexemes with each token.
- We might associate the "number" token with the set { **0**, **1**, **2**, …, **10**, **11**, **12**, … }
- We might associate the "string" token with the set { **""**, **"a"**, **"b"**, **"c"**, … }
- We might associate the token for the keyword **while** with the set { **while** }.

How do we describe which (potentially infinite) set of lexemes is associated with each token type?

Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have finite descriptions:
  - Define the language using an recognizer.
  - Define the language using a generator.
  - EG: Use a regular expression.
- We can use these compact descriptions of the language to define sets of strings.
- Over the course of this class, we will use all of these approaches.

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to generate certain languages (the *regular languages*).
- Often provide a compact and human- readable description of the language.
- Used as the basis for numerous software systems, including the **`flex`** .

Atomic Regular Expressions

- The regular expressions we will use in this course begin with two simple building blocks.
  - Often defined recursively.
    - Base Cases listed below
    - Recursive Cases listed on next slide
- The symbol **ε** is a regular expression matches the empty string.
- For any symbol **a in the Language**, the symbol **a** is a regular expression that just matches **a**.

Compound Regular Expressions

- If R_1 and R_2 are regular expressions, **R_1R_2** is a regular expression represents the **concatenation** of the languages of R_1 and R_2.
- If R_1 and R_2 are regular expressions, **$R_1 | R_2$** is a regular expression representing the **union** of R_1 and R_2.
- If R is a regular expression, **R*** is a regular expression for the **Kleene closure** of R.
- If R is a regular expression, **(R)** is a regular expression with the same meaning as R.

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R*$$

$$R_1 R_2$$

$$R_1 \mid R_2$$

- So **ab*c|d** is parsed as **((a(b*))c)|d**

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
1111101111100111111

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
1111101111001111

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

11110111
111111
0111
0

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

11110111
111111
0111
0

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

1*0?1*

11110111
111111
0111
0

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents “some letter.”
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

Applied Regular Expressions

- Suppose our alphabet is `a`, `@`, and `.`, where `a` represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

a^+ (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+ \quad (.a^+)^* \quad @ \quad a^+ \quad (.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+(.a^+)^*@a^+(.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0123456789]*[02468]

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0-9]*[02468]

42
+1370
-3248
-9999912

Summary

- Lexical analysis splits input text into **tokens** Which are categories of **lexemes**
- Lexemes are sets of strings often defined with **regular expressions**.

COSC252: Programming Languages:

Tokenizer

Jeremy Bolton, PhD

Adjunct Professor