Languages & Automata

Part 8

Languages

Yes, we have to cover the dry stuff

Languages

- An *alphabet* is a non-empty set of symbols
- a *string* is a sequence of zero or more symbols (a tuple)
- λ (lambda) represents a string of length 0
- A *language* is a set of strings

Definitions

- If *u* and *v* are strings, then *u v* is the concatenation of *u* and *v*
- Lexicographic order:
  - strings listed in length order
  - with same-length strings listed in dictionary order (non-standard definition)

String Length Notation

- /s| is the length of string s
- If A is an alphabet…
  - A^n is the set of all strings over A of length *n*
  - A^* is the set of all strings over A of any length (zero to infinity)
  - A^+ is used to represent a set of strings of A which are one or more symbol

Regular Expressions

Specifying a Pattern

Regular Expressions

- A *regular expression* is a method of specifying a language
- Used extensively in parsing theory and language design
- Do not confuse them with the regex expressions

Regular Operations

- Regular expressions have a set of defined "Regular Operations" on sets of strings
- These are used in regular expressions to define languages

Concatenation

If A and B are non-empty sets of strings:

$$AB = \{ ab \mid a \in A, b \in B \}$$

Series of Zero or More

- Regular expressions also allow zero-to-infinity number of characters to exist
- The notation consists of an asterisk superscript following a set
- It states that the expression contains a series of zero-to-infinity characters from that set

Series of Zero or More

$$A^* = \{\} \cup \{ a_1 \mid a_1 \in A \} \cup \{ a_1\ a_2 \mid a_1, a_2 \in A \} \cup \{ a_1\ a_2\ a_3 \mid a_1, a_2, a_2 \in A \} \cup \ldots$$

Series of One or More

- Often, regular expressions contain a helpful "shortcut" notation for when one-to-infinity characters are being defined
- The notation consists of a + superscript following the set
- It is not required, since a solo occurrence can be followed by the zero-to-infinity notation

Series of Zero or More

$$A+ = \{ a_1 \mid a_1 \in A \} \cup \{ a_1 a_2 \mid a_1, a_2 \in A \} \cup \{ a_1 a_2 a_3 \mid a_1, a_2, a_2 \in A \} \cup \ldots$$

Example

Give A = {a}

A* = { λ, a, aa, aaa, aaaa, …}
A+ = { a, aa, aaa, aaaa, … }
AA* = { a, aa, aaa, aaaa, … }

Alternatives

- Regular expressions contain a version of an "or"
- Which, allows a series of characters to be accepted it they are in the first or the second set
- Several notations exist including pipe | and the logical or ∨

Example

Given A = {a}, B = {b}

A | B = a, b
A | B+ = a, b, bb, bbb, bbbb…

Parenthesis

- Just like algebraic equations, regular expressions allow the writer to specify precedence with parenthesis
- If the precedence is obvious, then they can be removed

Example

Given A = {a}, B = {b}

(A | B)+ = { a, b, ab, ba, aaa, aab, aba, abb, … }

Even Shorter Notation

- Often, we just write the character itself rather than using a set defined
- But, it is understood of being equivalent to a set containing that single character
- a* = {a}*

More Examples

```
a*b* = {λ, a, b, aa, ab, bb,
        aaa, aab, abb, bbb, ...

a* | b* = {λ, a, b, aa, bb,
           aaa, bbb, aaaa, bbbb, ...}
```


Regular Expressions in Languages

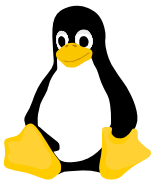
Where the rubber+ hits the road*

Regular Expressions in Languages

- Regular expressions are used extensively in patterning matching and programming languages
- Every time you write a program, they are being used, by the compiler, to parse your program

UNIX Grep

- Unix has long had tools that use regular expressions to match data
- Please note: each version of regular expressions differs in syntax



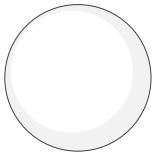
UNIX Grep

- The example below will find all of the lines in file.txt that have a substring matching ba*b
- The same feature is used by many UNIX applications

```
grep ba*b file.txt
```

Perl

- Perl (1987) may have been the first popular programming language to include their use
- Beyond the normal definition, Perl let's you specify the minimum and maximum length



Perl Notations

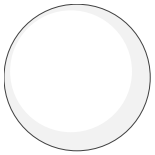
```
r?      → 0 or 1 strings from r
r+      → 1 or more string from r
r{n}    → Exactly n strings from r
r{n,}   → n or more strings from r
r{n,m}  → at least n, but not
          more than m, strings from r
```


Perl Example

```
(ab|b){2,3} = {bb, abb, bab, bbb,
               abab, ... }

… up to 12 strings
```

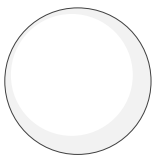

Perl Character Classes

- In Perl, you can specify set literals by using square brackets
- For shortcut notation, Perl lets you use a dash to specify a range (akin to the … in set notation)



Perl Character Classes

- Perl sets can also contain a caret (^) as the first character
- This makes the set a complement over the domain of ASCII characters
- It also has useful pre-defined sets



Character Set Descriptions

Perl	Set Notation
[abcdef]	{a, b, c, d, e, f}
[a-z]	{a .. z}
[0-9]	{0 .. 9}
[^a-z]	{a .. z}'

Java Patterns

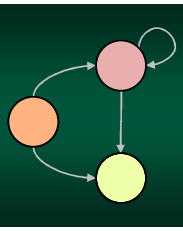
- Many languages have included Perl-style regular expressions as part of their classes
- In Java, the String class contains a "matches" method that will use a Perl pattern
- Periods match anything

Example

```
String s = "Computers are cool!";

s.matches("Computer")

... doesn't match & returns false
```


Example

```
String s = "Computers are cool!";

s.matches(".*Computer.*");

... does match.
```

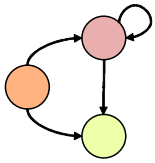



Languages & Finite Automata

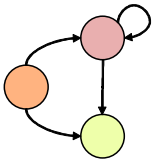
The structure of structures

Definitions

- Programming languages & patterns must be described formally
- A *finite automata* is, basically, a graph (with some additional properties) that can recognize a pattern



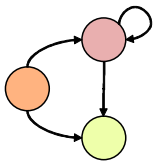
Restrictions



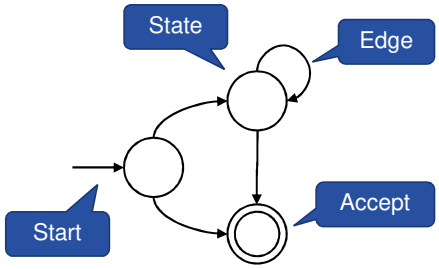
1. Each circle must have one arrow coming out for each symbol in the alphabet
2. One circle must have a start arrow (coming into it but not having an origin)
3. One or more circles must be designated "accept" (denoted with a double circle)

Languages & Finite Automata

- Each circle represents a *state*
- Each arrow, that connects states, is called an *edge*
- The double-circle represents an *accept state*



Example Finite Automata



Formal Description

- If we wanted to describe an FA to somebody without drawing a picture, how would we do it?
- What are the elements of the FA?

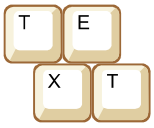
Formal Description

- The states - a set *S*
- The labels (on the arrows) - a set *A*
- The arrows – a function (F: *S* × *A* → *S*)
- Start state – a member of *S*
- Accept states – a subset of *S*

Accepting a Language

- If *M* is a FA, then the set of strings that are accepted by *M* is the language L(*M*)
- The FA "*accepts*" by…
  - repeatedly removing the front symbol from the string and following the matching arrow
  - until the string is completely gone
  - ends in "accept" state

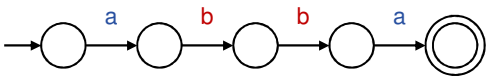
Example



- For these examples, let the alphabet A = {a,b}
- We will draw a number of FAs that will accept a string (of only a's and b's)

Example 1

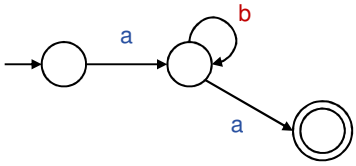
abba



Example 2

Starts with *a*, zero or more *b*, then an *a*

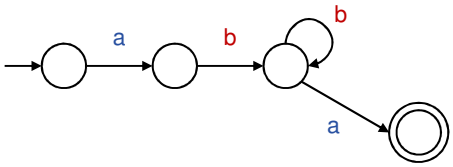
ab*a



Example 3

Start with *a*, one or more *b*, then an *a*

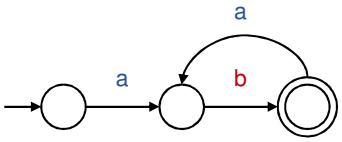
ab+a



Example 4

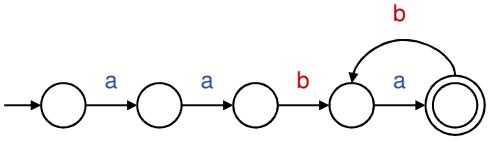
Series of *a b* (Examples: ab, abab)

(ab)+



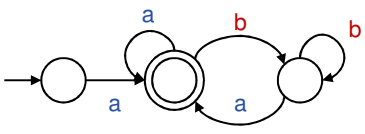
Example 5

Regular expression: a (ab)+ a



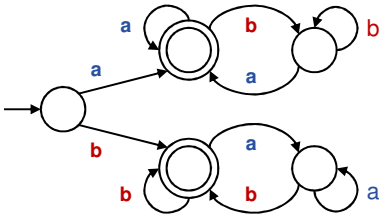
Example 6

String starting with *a* and ending in *a*



Example 7

String starting and ending with the same letter



Advice on Designing an FA

How to Write Them!

Advice on Designing an FA

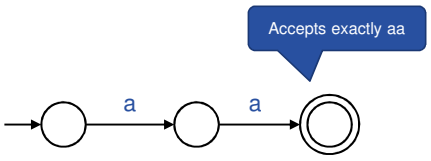
1. Have a meaning for each state – *it's the only "memory" that an FA has!*
2. First try to write just the part that accepts good strings
3. Make sure your FA is legal

Advice on Designing an FA

4. Sometimes string is in a accept/reject before finishing
5. It is easier to break big problems into sub problems
6. Try to "break" your solutions
7. Practice

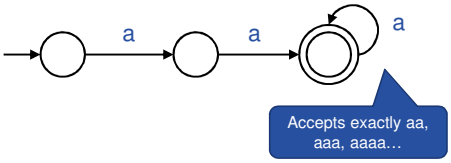
Construct Example

Strings that end in *aa*



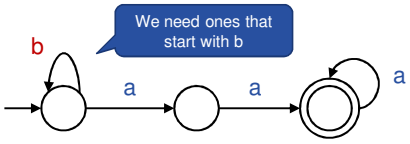
Construct Example

Strings that end in *aa* (the alpha is a b)



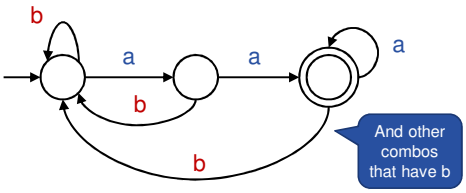
Construct Example

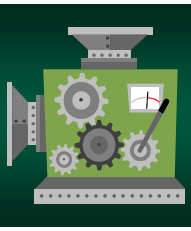
Strings that end in *aa* (the alpha is a b)



Construct Example

Strings that end in *aa* (the alpha is a b)



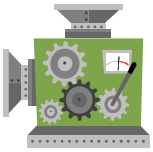


Regular Expressions to Automata

The Full Power Unleashed

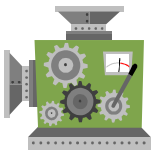
Regular Expressions to Automata

- Often, regular expressions are used to create a finite automata – which is easy to implement
- The process of this conversion is covered later in CSC 135



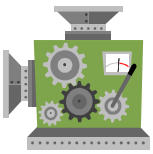
Parsers

- Parsers use finite automata to recognize the reserved words, identifiers, and other important "tokens" of a program
- They are defined using regular expressions



Parsers

- Parser generators often use regular expressions to describe the lexical structure of a language
- They use another system for syntax, but don't worry about that for this class…



Example Parser Generators

- YACC
  - C/C++ specific parser
  - created in 1970
  - uses UNIX-style expressions
- GOLD
  - multi-language parser
  - uses expressions that aren't perfect, but more closely resembles proper notation