# Lecture 11: Regular languages and Finite State Automata

## CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

*cab203@qut.edu.au*

# Outline

Regular Languages

Regular Expressions

Finite State Automata

# Readings

This week:

- None

Next week:

- Lawson Chapter 8 and 9

# Outline

# Symbols and alphabets

Start with some set of *symbols* $\Sigma \neq \emptyset$, which we will call the *alphabet*. This can be anything you like, possibly:

- $\{0, 1\}$
- $\{a, b, c, \ldots, z\}$
- The set of all printable ASCII characters
- The set of all UNICODE characters

The alphabet is the set $\Sigma$, and the elements of $\Sigma$ are the symbols.

# Strings

A *string over* $\Sigma$ is a sequence of symbols in $\Sigma$.

- Sometimes strings are also called *words*
- We will use the notation $x_j$ to refer to the $j$th symbol in $x$, counting from the left, from 1.
- The *length* of a string is the number of symbols in the sequence
- The unique string of length 0 is called the *empty string*, with symbol $\varepsilon$
- The set $\Sigma^*$ (the *Kleene star*) is the set of all strings over $\Sigma$ of any length

Mathematically, strings are the same as tuples over $\Sigma$, but we think about and notate them differently.

# String concatenation

We can *concatenate* two strings to form a longer string.

- $x = s_1 s_2 \ldots s_j \in \Sigma^*$
- $y = t_1 t_2 \ldots t_k \in \Sigma^*$
- $xy = s_1 s_2 \ldots s_j t_1 t_2 \ldots t_k$

Example:

- $x = \text{abc}$, $y = 123$, $xy = \text{abc123}$

# Languages

A *language* over an alphabet $\Sigma$ is a set of strings over $\Sigma$.

- A language $L$ is any subset $L \subseteq \Sigma^*$
- We can specify a language by writing it out explicitly:

$$L = \{1, 11, 111, 1111\}$$

- We can also specify a language by writing rules for the strings it contains:

$$L = \{x \in \{0,1\}^* : x_1 = 1\}$$

# Language examples

- $\emptyset$
- $\{1, 11, 111, 1111 \dots\}$, the set of all strings with all 1's
- The set of binary representations of all odd natural numbers
- The set of decimal representations of prime natural numbers
- The set of all valid Python programs
- The set of all bit strings which are UNICODE encodings of a word in English
- The set of Shakespeare's plays

# Decision problems

A *decision problem* for a language $L$ is the problem of deciding whether a given string $x \in \Sigma^*$ is in $L$.

- Any computation problem with a yes/no answer can be phrased as a decision problem

> Some languages are undecidable meaning that no computer program can solve the decision problem for that language.

# Language operations

- We can perform set-theoretic operations on languages, like $\cup$ since languages are sets
- We can *concatenate* languages by pairwise concatenating all of their elements.

$$A \cdot B := \{ab : a \in A, b \in B\}$$

# Examples

- Let:

$$A = \{0, 1\}, \quad B = \{a, b\}$$

then

$$\begin{aligned} A \cdot B &= \{0a, 0b, 1a, 1b\} \\ A \cdot A &= \{00, 01, 10, 11\} \\ A \cup B &= \{0, 1, a, b\} \end{aligned}$$

# Kleene star

The *Kleene star* of a language $A$ is the set of all possible concatenations of any length of strings from $A$

- $A^0 := \{\varepsilon\}$
- $A^1 := A$
- $A^j := A^{j-1} \cdot A$
- $A^* := A^0 \cup A^1 \cup A^2 \cup \ldots$

The *Kleene plus* is like the Kleene star, but omits the empty string.

- $A^+ := A^1 \cup A^2 \cup A^3 \cup \ldots$

# Regular languages

The *regular languages* are a particular set of languages that have some nice structure, defined by:

- $\emptyset$ and $\{\varepsilon\}$ are regular languages
- For each $a \in \Sigma$, $\{a\}$ is a regular language
- If $A$ and $B$ are regular languages, then $A \cup B$, $A \cdot B$ and $A^*$ are all regular languages
- No other languages are regular

# Examples

- $A^+ = A \cdot A^*$
- $A^n = A \cdot A \cdot \cdots \cdot A$
- $\{a, aa, aaa, aaaa, \ldots\} = \{a\}^+$
- $\{abc, abcabc, abcabcabc, \ldots\} = \{abc\}^+$
- $\{aac, abc, acc, adc, \ldots\} = \{a\} \cdot \{a, \ldots, z\} \cdot \{c\}$
- $\{\varepsilon, ab, cd, abab, abcd, cdcd, \ldots\} = (\{a\} \cdot \{b\}) \cup (\{c\} \cdot \{d\})^*$

# More interesting examples

- The set of IP addresses in the usual format (e.g. 192.168.1.1)
- The set of legal email addresses
- The set of integers, in Base-10 representation (e.g. -1234)
- The set of valid dates in DD-MM-YYYY format
- Any finite language

# Non-regular languages

- Most programming languages are not regular (e.g. Python)
- The language $L \subseteq \{a, b\}^*$ consisting of all strings over $\{a, b\}$ that have an equal number of $a$'s as $b$'s
- The language of matched parentheses $\{\varepsilon, (), ()(), (()), (()()), \ldots\}$

These examples all require an unbounded amount of memory to keep track of things, for example to count the number of $a$'s and $b$'s.

# Python tuples

While Python has data structures for strings, these are intended for Unicode characters only. For strings over arbitrary elements, we use can use *tuples*.

```
>>> t = ( 'one', 'two', 3)     # tuple literals
>>> len(t)                     # number of elements
3
>>> t[0]                       # access specific element
'one'
>>> t[0] = 'blah'              # can't change tuples!
TypeError: 'tuple' object does not support item assignment
>>> a,b,c = t                  # tuple unpacking
>>> print(a,b,c)
one two 3
>>> t + ( 4, 5)                # tuple concatenation
('one', 'two', 3, 4, 5)
>>> v = ( 1, )                 # notation for a tuple with one element
>>> v
(1,)
```

# Python and languages

Pragramatically, languages are usually dealt with by building a parser, rather than sets of strings. But the mathematical definitions can still be implemented. The Kleene star is infinite, though, so we won't look at that here.

```
>>> S = { (0,1), (1,1) }; T = { (0,0), (1,) }
>>> { s + t for s in S for t in T }           # concat two languages
{(0, 1, 1), (1, 1, 0, 0), (0, 1, 0, 0), (1, 1, 1)}
>>> S | T                                     # union of languages
{(0, 1), (0, 0), (1, 1), (1,)}
>>> A = { 0, 1 }
>>> def strLenN(A, n):                # strings of length n over A
...     if n == 0:
...         return { () }
...     else:
...         return { s + (a,) for s in strLenN(A, n-1) for a in A }
>>> strLenN(A, 2)
{(0, 1), (1, 0), (0, 0), (1, 1)}
```

# Outline

Regular Languages

Regular Expressions

Finite State Automata

# Regular expressions

*Regular expressions* are a way of specifying a regular language over an alphabet $\Sigma$. They are tightly related to the definition of regular languages.

# Regular expressions

Regular expressions over an alphabet $\Sigma$ are strings over $\Sigma \cup \{(, ), |, *\}$

- the empty string $\varepsilon$ is a regular expression
- `x` is a regular expression for an $x \in \Sigma$ (literals)
- if `x` is a regular expression then `(x)` is a regular expression
- if `x` and `y` are regular expressions then so is `xy`
- if `x` and `y` are regular expressions then so is `x|y`
- if `x` is a regular expression then so is `x*`.

The set of regular expressions over some alphabet $\Sigma$ is itself a language, and it is not regular.

# Regular expression, order of operations

Order of operations for regular expressions

- `( )`
- Kleene star
- concatenation
- `|`

So `ab*` is the same as `a(b*)`, and `abc|def` is the same as `(abc)|(def)`.

# Matching

Regular expressions are specifications for strings following a certain pattern. If a string follows the pattern for a regular expression then we say that the regular expression *matches* the string. The rules for matching are:

- $\varepsilon$ matches the string $\varepsilon$
- for any $x \in \Sigma$, $x$ matches the string $x$
- for any regular expressions $x$, $y$, `xy` matches a string $z$ if $z = uv$, $x$ matches $u$ and $v$ matches $y$
- for any regular expressions $x$, $y$, `x|y` matches a string $z$ if $x$ matches $z$ or $y$ matches $z$
- for a regular expression $x$, `x*` matches $z$ if $z = z_1 z_2 \ldots z_j$ and $x$ matches each of $z_1$, $z_2$, $z_j$.

# Examples

- `(ab)+c` matches abc, ababc, abababc etc.
- `(a|b)*c` matches c, ac, bc, aac, abc, bac, bbc, aaac, etc.

# Regular languages and regular expressions

- For any regular expression $x$, the set of strings that $x$ matches is a regular language.
- For any regular language $L$, there is a regular expression that matches exactly the set of strings in $L$.

To make this strictly true, we need to introduce one more character, $\emptyset$, which is a regular expression and matches nothing. Then the regular expression $\emptyset$ matches exactly the regular language $\emptyset$.

# Extensions

We can add some *syntactic sugar* to make regular expressions easier to use

- `x+` means the same thing as `xx*`
- `[xyz]` means the same as `(x|y|z)` and similar
- `.` means the same thing as [ *every symbol in* $\Sigma$ ]

One standard for regular expressions is POSIX regular expressions which includes more syntactic sugar.

# Using regular expressions

Most programming languages, and many special tools, allow you to use regular expressions:

- Test if some string matches a regular expression
- Search a string for substrings that match a regular expression
- Searching through entire files, directories for substrings that match regular expressions (`grep`)
- Rewriting substrings based on regular expressions and editing rules (`sed`)

# Applications of regular expressions

- Specifying and implementing parts programming languages (e.g. recognising legal variable names)
- System administration (searching for files with certain types of file names)
- Programming (e.g. searching source tree for definitions of functions)
- Checking user input (sanitising strings)
- Database admin (sanitising searches, ensuring data has correct format)
- Network admin (recognise DNS names with certain forms, etc.)
- Superhero scenarios (XKCD "Regular Expressions")

# Regular expressions in Python

Regular expressions are handled by thet re module. Regular expressions are just strings.

```
>>> import re                  # use the re module
>>> r = 'a+b+c'                # define our regular expression
>>> s = 'blahabcddeeaabbccaa'     # some string
>>> re.findall(r,s)            # get a list of all substrings matching r
['abc', 'aabbc']
>>> re.split(r,s)              # split the string on substrings matching r
['blah', 'ddee', 'caa']
>>> re.sub(r,'HERE',s)         # substitute matches of r with text
'blahHEREddeeHEREcaa'
>>> m = re.search(r,s)         # search for first match of r
>>> m.span()                   # location in string of first match
(4, 7)
>>> m.group()                  # matching substring
'abc'
```

# Outline

Regular Languages

Regular Expressions

Finite State Automata

# Finite state automata

A *finite state automaton* (plural *automata*) is one model of computation. It consists of:

- An alphabet $\Sigma$
- A set $S$ of *states*
- A *starting state* $s_0 \in S$
- A set of *accepting states* $A \subseteq S$
- A *state change function* $\delta : S \times \Sigma \to S$

As the name suggests, $S$ is a finite set.

If you give an FSA an infinite linear memory, you get a Turing machine, which is the standard model of computing.

# Using FSA

We can feed an input $x = x_1 x_2 \dots x_n \in \Sigma^*$ into a FSA like so:

$$
\begin{aligned}
t_0 &= s_0 \\
t_j &= \delta(t_{j-1}, x_j) \text{ for } j = 1 \dots n
\end{aligned}
$$

- If $t_n \in A$ then the FSA *accepts* $x$
- If $t_n \notin A$ then the FSA *rejects* $x$

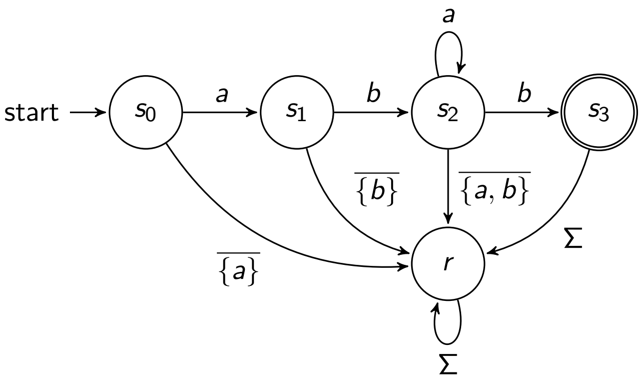
FSAs are sometimes called deterministic finite automata, in contrast with non-deterministic finite automata, where the state change function outputs a *set* of possible states to transition to.

# State change diagrams

A *state change diagram* allows us to depict a FSA using a graph-like diagram

- Each state is a vertex
- We draw an edge $(s, t)$ with label $x$ if $\delta(s, x) = t$
- If $\delta(s, x) = t$ for multiple $x$,then we just draw one edge with multiple labels
- We allow *loops* which is an edge from a vertex to itself
- Mark the accepting states (we'll use a double circle)

# State change diagram example




State change diagrams are not quite directed graphs since we allow loops. Once you allow loops in directed graphs you can depict any binary relation.

# Recognising languages

A FSA $M = (\Sigma, S, s_0, \delta, A)$ *recognises* a language $L \subseteq \Sigma^*$ if

- For every $x \in L$, $M$ accepts $x$
- For every $x \notin L$, $M$ rejects $x$

# Kleene's theorem

Kleene's theorem states that the set of languages recognisable by finite state automata is the same as the set of regular languages.

- Every regular language is recognised by some finite state automaton
- Every finite state automaton recognises some regular language

# Recognising some simple languages

We'll see how to recognise a simple subset of regular languages given by regular expressions consisting of:

- Single symbol literals
- $*$
- $+$

# Recognising a simple sequence of literals

Suppose we have a regular expression which is just literals: $x_1 x_2 x_3 \dots x_n$. We can recognise this with a simple FSA:

- States $\{s_0, s_1, \dots s_n, r\}$
- State change function

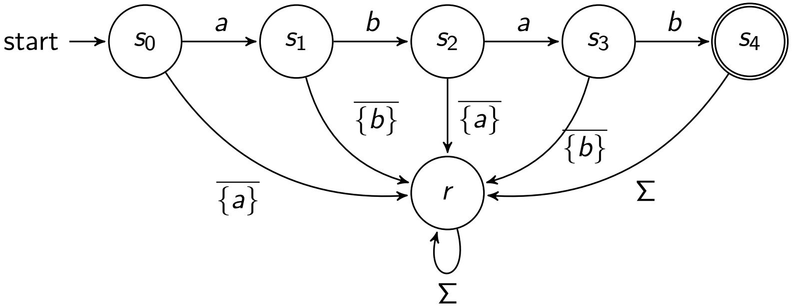
$$\delta(s, x) = \begin{cases} s_{j+1} & s = s_j \wedge x = x_{j+1} \\ r & x \neq x_{j+1} \vee s = s_n \vee s = r \end{cases}$$

- $A = \{s_n\}$

Basically, we move to a new state if we see the correct next letter in the sequence, otherwise we move to a special rejecting state. Also, we move to the rejecting state if the sequence is too long.

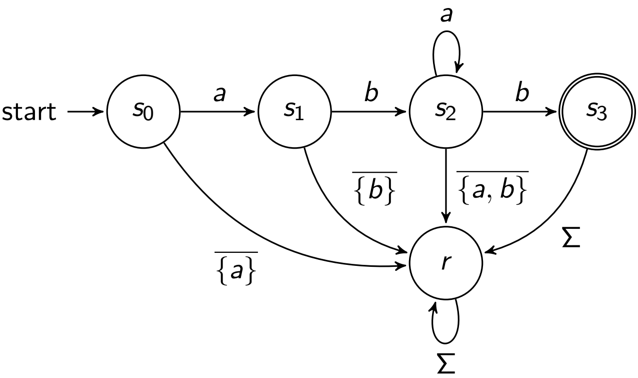
# Simple literal example

Let's build a FSA to recognise the regular expression `abab`:



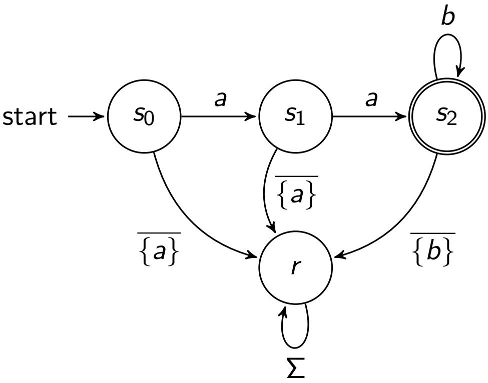
# Adding Kleene stars

If we have a Kleene star then we add a loop instead of a state for the literal before the *. For example, to recognise `aba*b`:



# Kleene star at the end

to recognise `aab*`:




Here we added a loop on the accepting state

# Kleene plus and special cases

There are some special cases that break our simple method, but we can fix them up:

- The expression `a+` is the same as `aa*`
- The expression `a*a` is the same as `aa*`

Using these expressions, we can deal with some more cases, including the Kleene plus. There are other special cases that we won't cover:

- `a*b*`

# Common gotchas

- In our construction we use a "reject" state $r$, but it is *not requirement* in general
- Make sure there is exactly *one* state to transition to for each symbol

# Uses of FSA

- Model of computation
- Model of protocols (network connections)
- Modelling aspects of systems programming

# FSA's in Python

We can build FSA's using a combination of Python structures

```
>>> delta = { (1,0): 2, (1,1): 1,
...           (2,0): 2, (2,1): 3,
...           (3,0): 2, (3,1): 3 }
>>> def runFSA(start, delta, accepting, input):
...     state = start
...     for i in input:
...         state = delta[(state, i)]
...     if state in accepting:
...         return True
...     return False
...
>>> runFSA(1, delta, { 3 }, (1,0,1,1))
True
>>> runFSA(1, delta, { 3 }, (1,0,0))
False
```