Introduction to the course topics

The course covers a variety of topics dealing with finite state machines, regular expressions, grammars and program verification, as well as, an introduction to the theoretical limits of algorithmic computation.

The relationship between programs and formal languages provides an example of the impact of theory on practice. Uses of formal theory include the following:

- grammars are used in lexical and parsing stages of compiler construction
- use of regular expressions in text editors
- state-charts in object-oriented modeling
- circuit-design
- DNA and protein sequence matching: regular expression matching algorithms

On the other hand, theory acts also as an "early warning system" by providing a science of the impossible:

- what should not be attempted because it is impossible (or provably too costly)

A fundamental question in computing is whether there exist tasks/problems that, even in principle, cannot be solved algorithmically and, if yes, which tasks are algorithmically solvable and which are not. In fact, it can be established that the number of different computational problems is *larger* than the number of all possible programs (in some programming language such as Java or C), which means that there must exist problems that are not solvable by any program (or by any algorithm).

Note that the number of programs is infinite, and to show that the number of computational problems is larger, we need to compare the *sizes* of different infinite numbers.

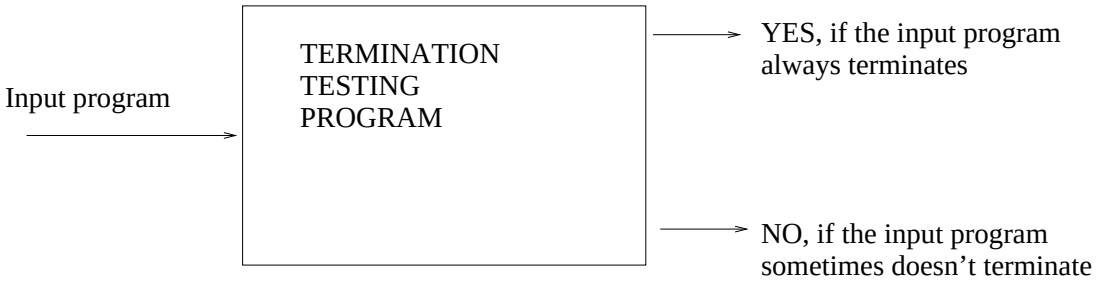

Figure 1: Example of an uncomputable problem.

In this course we use a different approach. Using a technique called *diagonalization* we establish that certain specific (and "useful") computational problems cannot be solved by any program written in the language C. The most well-known example of an unsolvable problem is the so called *halting problem* that asks whether an arbitrary program given as input terminates.

Example. A program with behavior as depicted in Figure 1 does not exist!

However, having an algorithm A for a computational problem P does not mean that P is solvable *in practice.* It might be the case that for inputs of moderate size A would need more time than the age of the universe, even when executed on modern computers ...

These observations can be summarized as a coarse classification of algorithmic problems/functions:

1. Non-computable problems (that is, even in principle impossible to solve using an algorithm/a computer)

2. Possible–with–unlimited–resources BUT impossible–with–limited–resources

3. Possible–with–limited–resources: these are the problems "solvable in practice"

Typical questions we can consider:

- Program existence: Does there exist a program for a given algorithmic problem (or function)?
- Software specification: How should programs be specified?
- Software validation: Is a given program correct, i.e., does it satisfy the specification?
- Software construction: How is a correct program obtained?
- Semantics: What does a given program do? (this is related to correctness)
- Efficiency: Is there a more efficient (faster) program for the given problem?
- Hardware comparison: Is one machine more powerful than another one?

Given a programming problem it is easier to convince someone that there is a program which solves the problem (if a program exists) than to convince someone that there is *no* program for the problem (if a program does not exist). In the former case it is sufficient to give the program and, in fact, usually it is sufficient to just outline the solution informally, or in pseudo code (if the purpose is just to convince the reader that a program exists).

Example. A program to compute the function $f(n) = n^2$.

On the other hand, if we want to show that a program for the given problem does *not* exist we need to show that none of the infinitely many possible programs solves the given problem (or computes a given function).

In order to be able to deal with negative results of this kind, we need to be precise about what constitutes a legal program! (or a legal algorithm)

Using a more practical perspective, a problem may be "uncomputable" also due to other types of reasons, for example, predicting the weather for a month in advance is impossible because the required input would be "too large" to be collected.

Instead of considering general algorithms[1], in the course we begin here with a simpler problem:

- test whether arbitrary input strings (= sequences of symbols) can be matched by a given pattern.

Special notation and implementation techniques have been developed to specify and recognize such *patterns:*

- state-transition diagrams (finite automata): simple simulated machines
- regular expressions: rules for building patterns
- grammars: rules for generating patterns

Alphabets, strings and languages

This material is from Chapter 7 in the textbook.

We begin with an introduction to basic concepts of formal languages.

- An alphabet is a finite, nonempty set of elements. The elements of the alphabet are called symbols (or tokens, characters).
- A string over an alphabet Σ is a finite sequence of symbols of Σ. (Strings are sometimes called also *words.*)
- A language over Σ is a set of strings over Σ.

Examples.

[1]The general limits of algorithmic computability will be discussed in the last part of the course.

1. English alphabet $\{a, b, c, d, \ldots, z\}$

   Strings: cat, dog, mouse, xzrbstuph, ...

   Language: the set of all correct English sentences

   — not precisely defined ...

2. Alphabet: $\{a, b\}$

   Strings: $\varepsilon, a, b, ab, ba, aa, bb, aaa, \ldots$

   Language: $\{aaa, aab, aba, baa, abb, bab, bba, bbb\}$ - a finite language

   Another language over the same alphabet $\{a^i b^i \mid i \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$

   This is an infinite language.

3. Alphabet: Java reserved words and identifiers

   Example string: a Java program

   Language: the set of all syntactically correct Java programs

We use the following notions and definitions concerning strings:

- The empty string is denoted ε.

  ε is a string over any alphabet.

- The length of a string is the number of occurrences of symbols in it. The length of a string s is denoted $|s|$.

  Examples:

  - The length of ε is 0, that is, $|\varepsilon| = 0$.
  - The length of the string $bccb$ is 4, that is, $|bccb| = 4$.

- The concatenation of strings x and y is denoted $x \cdot y$, or xy for short. It is the string obtained by appending y to x.

Examples: If $x = abc$ and $y = de$, then $xy = abcde$ and $yx = deabc$.

As seen from the example, the concatenation operation is not commutative. Note also that ε acts as an identity for string concatenation: $x \cdot \varepsilon = \varepsilon \cdot x = x$ for all strings x, in particular, $\varepsilon \cdot \varepsilon = \varepsilon$.

Since concatenation is associative we do not need to use parentheses:
for all strings x, y, z we have $x(yz) = (xy)z$. How would you prove this?

- If x is a string, x^n denotes the concatenation of n copies of x (the power of a string). Here $n \geq 0$.

  Inductive definition of powers of a string:

  1. $x^0 = \varepsilon$
  2. $x^{i+1} = x \cdot x^i$, for $i \geq 0$.

  Example.

  $(abc)^0 = \varepsilon$

  $(abc)^3 = abcabcabc$

- If $s = xy$, we say that x is a prefix of s and y is a suffix of s. If $s = xyz$, we say that y is a substring of s. Note that here x and/or z may be the empty string.

  Examples:

  1. ab is a prefix of aba. The other prefixes of aba are ε, a and aba.
  2. ba is a suffix of aba
  3. ε is a prefix/suffix/substring of any string
  4. a string is always a prefix/suffix/substring of itself
  5. What are the substrings of $cbcc$?

Formal languages

A formal language has to be precisely defined, the word *formal* refers to the fact that we have a precise set of rules which tell us exactly which strings are in the language, respectively, are not in the language (or an algorithm that answers this question).

- A finite language can (at least in principle) be defined by listing all strings in it.

  Example: $\{00, 01, 10, 11\}$

- Infinite languages can be defined by giving some condition that exactly characterizes the strings in the language.

  Examples.

  $\{0^n \mid n \geq 0\}$

  $\{0^n 1^n \mid n \geq 1\}$

  $\{0^n 1^m \mid \sqrt{n} \leq m \leq n\}$

Note that $\emptyset$ (the empty set) is a language over any alphabet Σ. Also $\{\varepsilon\}$ (the language having only the string ε) is a language over any alphabet Σ. It is important to remember that $\emptyset \neq \{\varepsilon\}$. Why?

As indicated in the above example, languages can be speficied by "set notation". The problem with set specifications is that the notation is too powerful and, if the conditions can be specified using first-order logic, membership in the language will be an unsolvable problem.

In the first part of the course we want to consider language specification tools that allow the development of efficient algorithms for the language. The first idea is that we can define new languages from "simpler" ones using operations on languages. Three important operations we consider are

- union
- concatenation
- closure

Later we will see that all regular languages can be built from elements of Σ, the empty string ε and the empty set $\emptyset$ using these operations.

Union

If R and S are languages over Σ, their union is denoted $R + S$. It consists of all strings that are in R *or* in S. (Thus $R + S$ is just a different notation for the union of sets, $R \cup S$.)

Concatenation

If R and S are languages, their concatenation is defined as

$$R \cdot S = \{rs \mid r \in R, s \in S\},$$

usually written simply as RS.

Examples.

If $R = \{a, ab\}$, $S = \{bc, c\}$, what is their concatenation RS? Note: the concatenation consists of 3 different strings.

If $R_1 = \{a, \varepsilon\}$ and $S_1 = \{ab, b\}$, what is $R_1 S_1$?

Here it is useful to consider how the empty set and the set consisting of the empty string behave with respect to set concatenation. What are the following languages:

$\emptyset \cdot R = \ldots\ldots ?$

$\{\varepsilon\} \cdot R = \ldots\ldots ?$

$\{\varepsilon\} \cdot \emptyset = \ldots\ldots ?$

When union or concatenation is applied to finite languages, the result is always finite. The third operation allows us to specify infinite languages.

Closure of languages

The set of all strings over alphabet Σ is denoted Σ^*. The closure operation can be extended for any language S:

$$\begin{aligned} S^* &= \{s_1 \cdot \ldots \cdot s_n \mid s_i \in S, i = 1, \ldots, n, n \geq 0\} \\ &= \{\varepsilon\} + S + S^2 + S^3 + \ldots \end{aligned}$$

Example. Let $S = \{01, 1\}$. Then

$S^0 = \{\varepsilon\}$

$S^1 = S = \{01, 1\}$

$S^2 = \{0101, 011, 101, 11\}$

$S^* = \{\varepsilon, 1, 01, 11, 011, 101, 111, 0101, \ldots\}$

Here S^* is an infinite language and we cannot explicitly write down all strings of S^*.

Example. $\emptyset^* = \{\varepsilon\}$. This can be verified using the definition of the closure operator.

We denote also

$$\begin{aligned} S^+ &= \{s_1 \cdot \ldots \cdot s_n \mid s_i \in S, i = 1, \ldots, n, n \geq 1\} \\ &= S + S^2 + S^3 + \ldots \end{aligned}$$

Note that $S^* = S^+ + \{\varepsilon\}$ for any language S.

Now we come to a definition that will be central in the first part of the course.

Definition. A language S over alphabet Σ is said to be regular if S can be defined from elements of Σ and $\emptyset$ using the operations *union, concatenation* and *closure.* The description of the language S in this form is called a regular expression for S.

- Note that, as observed before, $\emptyset^* = \{\varepsilon\}$. Sometimes the definition of regular languages includes ε as one of the "basic building blocks", but ε can be produced using the closure operator.
- In particular, all finite languages are regular. Why?

As we will see, the regular languages have "nice" properties and can be easily implemented. However, regular languages form only a "small" family of languages and we will develop techniques for showing that a language is not regular.

Application: *Sequence matching problem*

Each DNA molecule is composed of two strands that are made up of a sequence of nucleotides. Each nucleotide has one of four bases, represented by symbols A, T, C, G (and other parts). Proteins are large molecules that are composed of a sequence of amino acids. There are 20 amino acids that occur in proteins, denoted by standard one-letter symbols. In this way, DNA or protein molecules can be represented as strings. Analyzing DNA or protein sequences can help to determine which function they perform, or what parts of the sequence are important for a particular function. Comparing different DNA sequences tells us which organisms are related.

Related sequences are not necessarily identical. When comparing DNA or protein sequences we want to *align* them in a way that minimizes some type of distance between the sequences.

Regular expressions are used to specify *patterns* that describe a related set of strings (sequences). In this way we can compare the pattern to an individual sequence or to a database of sequences to find good matches. The applications often use *extended regular*

expressions that allow operations other than union, concatenation and closure. (Examples in class.)

The tools used to solve sequence matching problems typically involve also *finite state machines* that are discussed in out next topic. The BLAST family of search engines use heuristic techniques to build a large *deterministic finite automaton* that, for a given query string, finds from a database of known sequences the most closely related ones.