COMP4141 Theory of Computation

Lecture 1 Introduction, Sets, and Words

Ron van der Meyden

CSE, UNSW

February 29, 2016

(Credits: David Dill, Thomas Wilke, Kai Engelhardt, Peter Höfner, Rob van Glabbeek)

Organisation

Classes

- lectures :
  Mon 12:00 - 13:00 Mathews 310 (K-F23-310)
  Wed 16:00 - 18:00 Mathews 310 (K-F23-310)
- slides + white board

Organisation

Homework

- weekly exercise
- return Wed before the lecture
- working in pairs OK

Assessment

- 30% homework (12×2.5)
- 70% exams:
  a mid-session test (in Week 6- date tba) (1 hour), worth up to 20 marks
  a final exam (date tba) (2 hours) worth up to 50 marks

Organisation

http://www.cse.unsw.edu.au/~cs4141/

Consultation

- questions before/after/during lectures
- tutorials
- office hour: immediately after lectures
- email

COMP4141

What is it?

- a computer science subject
- timeless
- moderately challenging
- fun

What is it not?

- a programming subject
- a skills subject
- buzzword-compliant
- a trivial source of 6UoC

Value of the course

In 20 years, computers and programming will be vastly different. But this material will be very much the same—and will still be useful.

Provides insight into fundamental questions

- defines the questions
- answers some
- many are open!
- very close connection with logic, algorithms, linguistics, others.

Provides advanced problem-solving tools.

- springboard for more advanced courses
- research
- applications

Practice with mathematics and proofs.

Main Questions

How can we represent the inputs and outputs of computations?

What sorts of things can be computed?

How hard is it to compute?

How do we express computation?

Set theory, developed as a foundation for all of mathematics, provides a very useful formal framework in which to express the answers to such questions.

Set Theory

- union: $S \cup T$
- intersection: $S \cap T$
- empty set: $\emptyset$
- set difference: $S \setminus T$ or $S - T$
- complement: $\overline{S}$
- distributivity: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$
  $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$
- subset: $S \subseteq T$
- element of: $x \in S$
- comprehension: $\{x \in S \mid \phi(x)\}$ or $\{x \in S : \phi(x)\}$
  the set of elements of S satisfying ϕ

Representing Sets (discussion)

Suppose a programmer needs to represent a small, *finite*, set S.

- What does "*represent*" mean?
  Answer: You can answer questions about it.
  Simple common question: Is $x \in S$?
  Other questions: Is $S = \emptyset$? Is $S \cap T = \emptyset$? Etc.
- What representations would be appropriate?

Suppose you want to represent *infinite* sets. How do you do it?

- Same question: What does "represent" mean?
  Same answer: You can answer questions about it.
  Same simple common question: Is $x \in S$?
- What representations would be appropriate?
  That's what this part of the course is about.

What is a representation?

Suppose you have devised a notation for sets, that is a *representation* that can be stored in a computer.
Can all sets be represented?

This raises profound questions: Which sets can be represented on a computer and which can't?

One view of formal language theory

Automata and complexity theory is concerned with properties of *formal languages*.

In formal language, automata, and complexity theory, a *language* is just a set of strings.

(Like many mathematical definitions, this leaves behind most of what we think of as "languages," but can be made precise. And it leads to very profound results.)

Basically, any object or value that is of interest to computer science can be represented as a string.

So a set of anything can be considered a language.

Questions from formal language theory

What (infinite) sets are representable?

What can a computer do with the representations, in theory?

What cannot be done with the representations, in theory?

What problems are easy, hard, or impossible to solve computationally?

Another view of formal language theory

For practical purpose, a language is the same thing as a Boolean function. Such a function is also called a *property* or a *predicate*.

For example, the predicate **even**(x), which returns "true" iff x is (string representation of) an even number, can be considered to represent the set of even numbers (think of it as an "implicit set lookup").

So, if we can answer questions about languages, we are also answering questions about properties of objects.

Application: Computer languages

Basis for tools and programming techniques.

- Lexical analysis
- Parsing
- Program analysis

Many interesting problems in programming language implementations are hard or impossible to solve in general. Examples:

- Equivalence of grammars.
- Almost any exact analysis.

Application: Formal Verification

Formal verification attempts to prove system designs (e.g. programs) correct, or to find bugs.

Methods are generally from logic and automata theory. Many of the constructions in this course are used in practical tools.

- Automata constructs (e.g. product construction)
- Reductions to SAT (an NP-completeness proof technique).
- "Bounded model checking"—the idea is from Cook's theorem

It is also important to know a little about complexity theory, since many problems in this area are hard or impossible to solve, in general.

Basic concept

Definition

An *alphabet* is a non-empty finite set. The members of the alphabet are called *symbols*.

Examples

Binary alphabet $\{0, 1\}$

ASCII character set—the first 128 numbers, many of which are printed as special characters. Also, any other finite character set.

The capital Greek sigma (Σ) is often used to represent an alphabet.

Strings

Informally: A string is a finite sequence of symbols from some alphabet.

Examples

- ϵ—the empty string (the same for every alphabet). (Leaving a blank space for the empty string is confusing, so we use the Greek letter "epsilon"). ϵ is not a symbol! It is the string with no symbols; the string of zero length.
- 000, 01101 are strings over the binary alphabet
- "String" is a string over the ASCII character set, or the English alphabet.

Strings cont.

Definition (strings over alphabet Σ)

Base: ϵ is a string over Σ

Induction: If x is a string over Σ and a is a symbol from Σ, then xa is a string over Σ.

(Think of xa as appending a symbol to an existing string.)

Notation: The set of all strings over an alphabet Σ is written Σ^*.

Length of a string

Many functions are defined recursively on the structure of strings, and many proofs are done by induction on strings.

Informally: The *length* of a string is the number of occurrences of symbols in the string (the number of different positions at which symbols occur).

The length of string x is written $|x|$.

Definition (length)

Base: $|\epsilon| = 0$

Induction: $|xa| = |x| + 1$

Concatenation of strings

Informally: The concatenation of strings x and y over alphabet Σ is the string formed by following x by y. It is written $x \cdot y$, or (more often) xy.

Examples

- $abc \cdot def = abcdef$
- $\epsilon \cdot abc = abc$

Definition (concatenation)

The definition is recursive on the structure of the second string:
Base: $x \cdot \epsilon = x$ if x is a string over Σ.
Induction: If x and y are strings over Σ and $a \in \Sigma$ then

$$x \cdot (ya) = (x \cdot y)a$$

Note: The parentheses are not symbols, they are for grouping, so $x \cdot (ya)$ is x concatenated with ya.

Proof by Induction

- Show that for arbitrary strings x, y, z over Σ concatenation is associative, i.e.,

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Languages

Definition

A *language* over Σ is a subset of Σ^*.

NB

Of course, this omits almost everything that one intuitively thinks is important about a language, such as meaning. But this definition nevertheless leads to incredibly useful and important results.

Examples

- $\emptyset$ (the empty language)
- $\{\epsilon\}$ (the language consisting of a single empty string).
- The set of all strings with the same number of as as bs.
- The set of all prime numbers, written as binary strings.
- The set of all strings representing C programs that compile without errors or warnings.
- The set of all first-order logic formulas.
- The set of all theorems of number theory, in an appropriate logical notation.
- The set of all input strings for which a given Boolean C function returns "true."