Teoria da Computação

(Theoretical Computer Science)
Licenciatura em Engenharia Informática
Lecture Notes 2011-2012

Luis Caires

version of December 14, 2012

1 Review of Set Theory, Modelling with Sets

The basic goal of this chapter is to help you learn how to:

- model data spaces and data structures using basic Set Theory
- specify properties of states of a computing system and of elements within a data structures using Logic

1.1 Basic Set Theory

1. Sets, Everything is a Set and ZFC

   Set theory was invented to provide a foundation to model ALL mathematical concepts. In turn mathematical concepts can be used to model most concepts of scientific and technological disciplines. Informatics and computer science are not an exception. It turns out that set theory and mathematical logic are particularly convenient tools to model concepts in informatics and computer science.

   Set theory and logic play for informatics the same basic role as mathematical analysis (calculus) plays for disciplines such as physics or electronic engineering.

   We will base our presentation on ZFC (Zermelo-Fraenkel-Cantor) Set Theory, due to these three famous mathematicians. Set theory was also developed by pioneers of computer science, for example, John Von Neumann.

Set theory is based on the idea that "Everything is a set". Actually, this means "Everything can be modeled by a set". ZFC models things such as boolean values, natural numbers, relations, functions, databases, and even algorithms, just based on the fundamental notion of set.

2. Emptyset

   The empty set is the "simplest" set we may think of. It is the set without elements. It is represented $\emptyset$.

3. Membership

   The fundamental form of statement in set theory is

   $$x \in y$$

   which means "x is a member of y", or "x belongs to y".

4. Extensionality

   The "Extensionality Principle" of set theory means that sets are determined uniquely by their elements. If two sets (finite or infinite) have exactly the same elements, then they are actually the same set. For example, we may think of two Java vectors with exactly the same elements, without being the same vector. This is not the case with sets.

   Practically, if we want to check if two sets A and B are actually the same set, if is enough to check that every element of A also belongs to B, and that every element of B also belongs to A.

   $$A = B \Leftrightarrow (\forall x . x \in A \Leftrightarrow x \in B)$$

   Extensionality also implies that there is just one empty set.

5. Subset

   A set x is a subset of a set y if all elements of x belong to y. Formally, we have

   $$A \subseteq B \Leftrightarrow \forall x . (x \in A \Rightarrow x \in B)$$

   Note that $A \subseteq A$ for all sets A, and $\emptyset \subseteq A$ for all sets A. Sometimes we use $A \subset B$ to say that A is a strict subset of B. A strict subset of a set B is a subset that is not the trivial subset B.

   $$A \subset B \Leftrightarrow (A \subseteq B) \wedge A \neq B$$

6. Enumeration

   We can define sets in various ways.

   The simplest way is by exhaustively enumerating all the elements in the set you want to specify

$$\begin{array}{lcl} \mathit{BOOL} & \triangleq & \{\mathit{FALSE}, \mathit{TRUE}\} \\ \mathit{DWARFS} & \triangleq & \{\text{“Sneezy”}, \text{“Sleepy”}, \text{“Dopey”}, \text{“Doc”}, \text{“Happy”}, \\ & & \text{“Bashful”}, \text{“Grumpy”}\} \\ \mathit{LAMPSTATES} & \triangleq & \{\mathit{ON}, \mathit{OFF}\} \end{array}$$

   Obviously, this only works for specifying finite sets.

   When we define a set by enumerating its elements, the order or presentation does not matter! So, the following enumerations define the same set:

$$\{1, 2, 3\}$$
$$\{2, 1, 3\}$$
$$\{3, 1, 2\}$$

7. Sets, Sets of Sets, Sets of Sets of Sets, ...

   An set can also be an element of another set, and so on. This is useful to describe structured entities, with several components

$$\mathit{STACK} \triangleq \{0, \{2, \{3\}\}\}$$
$$\mathit{BOOLS} \triangleq \{\emptyset, \{\mathit{TRUE}\}, \{\mathit{FALSE}\}, \mathit{BOOL}\}$$

8. Comprehension

   We may define a new set using a logical property to select the elements we want to collect. For example

   The set of even natural numbers:

$$\mathit{EVEN} \triangleq \{n \in \mathit{NAT} \mid n\%2 = 0\}$$

   The set of non empty sets:

$$\mathit{NOTEMPTY} \triangleq \{s \mid s \neq \emptyset\}$$

   The general form of the “naive” comprehension principle allows us to define a new set given any property P expressed in the logic of set theory.

$$\{x \mid P(x)\}$$

The logic of set theory is essentially first-order logic enriched with several constants and operators that talk about sets, for example, the empty set, the membership relation, equality, etc, etc.

9. Russell's Paradox

   In 1901 Bertrand Russell discovered an inconsistency of Cantor-Frege set theory, by considering the set

   $$R \triangleq \{x \mid x \notin x\}$$

   Intuitively (so to speak), R is the set of all sets that are not members of themselves. Being not a member of itself is a property that make sense, in principle. We can think of many sets that enjoy this property, for example, the empty set is not a member of itself. The set of boolean values is not itself a boolean value.

   Since there are so many examples of sets that are not members of themselves, the set R as defined above, if it exists, must be not empty! We may even naively think that R contains all the sets that exists, since perhaps no set can be a member of itself.

   But a paradox (or inconsistency) arises! Consider the meaning of the proposition

   $$R \in R$$

   By definition of the "set" R, $R \in R$ means that $R \notin R$.

   Likewise, if we assume $R \notin R$, then it cannot be the case that $R \notin R$. So $R \notin R$ implies $R \in R$.

   Even if surprised at first, we must conclude that, according to the definition of R, we have

   $$R \in R \text{ if and only if } R \notin R$$

   which is obviously an absurd.

   Since we arrived to an absurd statement only by following the basic rules of logic and the definition of R, Russell concluded, rightly, that an expression like $\{x \mid x \notin x\}$ must be meaningless, and cannot be used to define a set. Such meaningless expressions should not be accepted by the language of set theory.

10. Separation

    To avoid confusions like Russell's paradox, we will always use Comprehension in a refined form, using the Separation principle of ZFC.

The general idea of the separation principle is that we may define a new set given any property P expressed in the logic of set theory, to select elements from some **already well defined** set S.

$$\{x \in S \mid P(x)\}$$

So, according to this principle, we have the right to write

$$\{n \in NAT \mid n\%2 = 0\}$$

a well defined set, but not an expression such as $\{s \mid s \neq \emptyset\}$.

Be careful to always use the separation principle when defining sets by comprehension in this course!

11. Union

    Besides Enumeration and Separation, we may define sets using the Union operation

    $$A \cup B$$

    Intuitively $A \cup B$ denotes the set that contains exactly the elements in A and B.

    $$\forall x.(x \in A \cup B) \Leftrightarrow (x \in A) \vee (x \in B)$$

    Given a set of sets S we also define the union $\bigcup S$ to mean the union of all sets which are elements of S. More precisely, we have

    $$\forall x.(x \in \bigcup S) \Leftrightarrow \exists y.(y \in S \wedge x \in y)$$

12. Intersection / Disjointness

    We may define sets using the Intersection operation

    $$A \cap B$$

    Intuitively $A \cap B$ denotes the set that contains exactly the elements that belong both to A and to B.

    $$\forall x.(x \in A \cap B) \Leftrightarrow (x \in A) \wedge (x \in B)$$

    We may also see that

    $$A \cap B = \{x \in A \mid x \in B\}$$

Given a set of sets S we also define the intersection $\bigcap S$ to mean the intersection of all sets which are elements of S. More precisely, we have

$$\forall x.(x \in \bigcap S) \Leftrightarrow \forall y.(y \in S \Rightarrow x \in y)$$

Two sets A and B are said to be disjoint, in symbols $A\#B$, if they do not contain any common member. We have

$$A\#B \Leftrightarrow (A \cap B) = \emptyset$$

We say that a collection S of sets is pairwise disjoint if all pairs of sets in the collection are disjoint. More precisely

$$\#S \Leftrightarrow \forall x.\forall y.(x \in S \wedge y \in S \wedge x \neq y \Rightarrow x\#y)$$

13. Relative Complement

    Given a sets A and B, the relative complement $A \setminus B$ denotes the set of all elements of A that do not belong to B. Formally

$$A \setminus B = \{x \in A \mid x \notin B\}$$

    The "absolute complement" of a set A, written $\overline{A}$ is not definable in ZFC, due to the Russell paradox.

14. Pairs

    For structuring information we need some kind of construction to aggregate data. The simplest one is the pair. We may form e.g., a pair consisting of a team and the size of the team.

$$\mathit{daltons} \triangleq (\{``\mathit{jack}", ``\mathit{joe}", ``\mathit{averell}", ``\mathit{william}"\}, 4)$$

    This corresponds to the well known notion of **ordered pair**. In set theory, everything is a set, and in fact an ordered pair such as the one above may be encoded in a set, using the scheme

$$(x, y) \triangleq \{x, \{x, y\}\}$$

    This encoding of pairs is a variant of one Kuratowski proposed in 1921.

    In practice, we will simply use the standard notation (x, y) to represent ordered pairs.

15. Products

    The product of to sets A and B, written $A \times B$ is the set of all ordered pairs whose first element belongs to A and the second element belongs to B.

    We have

    $$\forall x.(x \in A \times B) \Leftrightarrow \exists a.\exists b.(a \in A \land b \in B \land x = (a, b))$$

    This operation is also called the "cartesian" product. The name "cartesian" derives from the name of René Descartes, the mathematician-philosopher that invented the related concept of cartesian plane, where one conceive points with two coordinates (x, y) (even if it is best known by his famous punchline "I think therefore I am" :-).

16. Fixed Sequences and n-tuples.

    We may represent tuples of more than 2 elements by iterating the product. For example $STRING \times NAT \times STRING$ denotes the set of all triples (a, b, c) where $a \in STRING$, $b \in NAT$ and $c \in STRING$.

    This idea of forming sets of tuples of any fixed arbitrary length works by considering the operation $A \times B$ to be right associative, so $A \times B \times C$ is actually an abbreviation of $A \times (B \times C)$.

    In the same way a triple such as (a, b, c) is actually an abbreviation of a pair $(a, (b, c))$.

    So we can say, for example, that the first component of (a, b, c) is a and the second component of (a, b, c) is (b, c).

    Note however that a sequence such as $((a, b), c)$ is different from the sequence (a, b, c). The first is a sequence of two elements, namely the pair (a, b) and c, while the second sequence contains three elements, a, b and c.

    This reasoning applies to sequences of elements of arbitrary finite length.

17. Relations

    A (binary) relation between elements of a set A and elements of a set B is modeled as a subset of the product $A \times B$. For example, the relation $SAMEPAR$ that holds between two natural numbers if and only if they have the same parity (odd or even) is defined as follows

    $$SAMEPAR \triangleq \{(x, y) \in NAT \times NAT \mid x\%2 = y\%2\}$$

For example, $(2,8) \in \mathit{SAMEPAR}$ and $(9,1) \in \mathit{SAMEPAR}$ but $(191,256) \notin \mathit{SAMEPAR}$.

When R is supposed to denote a relation, we write $a\,R\,b$ for $(a,b) \in R$, to make it more readable. For example, we may write $2\ \mathit{SAMEPAR}\ 8$.

Here some other examples of binary relations:

$$x\ \mathit{FATHER\_OF}\ y$$

$$n\ \mathit{ANCESTOR\_OF}\ y$$

$$n\ \mathit{LINKED\_TO}\ y$$

We can also define relations between more than 2 elements. For that, we just iterate the constructions above, using products and n-tuples. For example, a phone list may be seen as a relation

$$\mathit{PHONELIST} \subset \mathit{FIRSTNAME} \times \mathit{LASTNAME} \times \mathit{PHONENUM}$$

where we may set $\mathit{FIRSTNAME} \triangleq \mathit{STRING}$, $\mathit{LASTNAME} \triangleq \mathit{STRING}$ and $\mathit{PHONENUM} \triangleq \mathit{NAT}$. For example, we may consider

$$(\text{“Luis”}, \text{“Caires”}, 218402825) \in \mathit{PHONELIST}$$

Relations are an extremely important concept in informatics and computer science. For example, it is pervasive in databases theory and practice, which are based in the so called relational data model, invented by Edgar Codd in 1970. Codd won the 1981 ACM Turing Award for this key contribution to Informatics. The relational model is the basis of most modern database systems, which use the query language SQL. You will learn more about this in the Databases course.

18. PowerSet

We often need to define the set of all subsets of a given set. For example, we may want to consider a specific phonelist, as defined above. To what set does such phonelist belong? Well, a single phonelist is a set of triples (each one representing a record) where each triple belongs to the set

$$\mathit{FIRSTNAME} \times \mathit{LASTNAME} \times \mathit{PHONENUM}$$

The set of all sets of records of these kind is denoted by the powerset

$$\wp(\mathit{FIRSTNAME} \times\ \mathit{LASTNAME} \times \mathit{PHONENUM})$$

In general, for any sets A and S we have that

$$A \in \wp(S) \Leftrightarrow A \subseteq S$$

19. Functions

A function is modeled in set theory just as a special kind of relation, a relation between arguments and the corresponding results. Since a function cannot give two different results for the same argument, we impose the following condition for a binary relation R to be considered a function

$$function(R) \triangleq \forall (x,y) \in R, \forall (x',y') \in R \,.\, (x = x') \Rightarrow (y = y')$$

This means that if F is a function such that $(\text{“}luis\text{”}, a) \in F$ and $(\text{“}luis\text{”}, b) \in F$ then $a = b$, for example $a = b = 45$. There cannot be two different pairs with the same first component!

We may think of F as the AGE function that assigns to a person its (unique) age.

Since the result b of a function relation F is unique for any given argument, we denote such result by $F(a)$ where a is the first element of the pair $(a, b) \in F$. In the example above, we have, say $F(\text{“}luis\text{”}) = 45$.

So, note that, in the end, a function in set theory is nothing but a set of ordered pairs!

To highlight the use of ordered pairs in the context of functions, we also use the following alternative notation for ordered pairs

$$x \mapsto y \triangleq (x, y)$$

The notation $x \mapsto y$ reads “x is mapped to y” (“x é aplicado em y”).

Given a function as a set (of ordered pairs) we also call such set (of ordered pairs) the **extension** of the function.

For example, the extension of the NOT function on booleans may be represented by:

$$NOT \triangleq \{TRUE \mapsto FALSE, FALSE \mapsto TRUE\}$$

Then, we have $NOT(TRUE) = FALSE$, and $(FALSE, TRUE) \in NOT$.

The set of all subsets of $A \times B$ which are functions is denoted by

$$A \to B$$

In other words,

$$A \to B \triangleq \{R \in \wp(A \times B) \mid function(R)\}$$

We may then write, as usual

$$NOT \in BOOL \to BOOL$$

$F \in A \to B$ means that F is a function that sends elements of A into elements of B.

The set A (in $A \to B$) is called the **domain** of the function F, and B the **codomain** of the function F.

There are several ways of defining functions in set theory. An convenient way we will often use is to follow the pattern

$$F \triangleq \{x \mapsto y \in D \times C \mid P(x, y)\}$$

where $P(x, y)$ is a logical condition between the argument x and the result y, D is the domain and C is the codomain. For example,

$$DOUBLE \triangleq \{x \mapsto y \in NAT \times NAT \mid y = 2 \times x\}$$

Then $DOUBLE(2) = 4$, etc...

20. Identity Function

    For any set A there is the identity function on A, that maps each $e \in A$ into itself. The identity on A is noted Id_A. We have

    $$Id_A = \{a \mapsto b \in A \times A \mid a = b\}$$

    so that $Id_A(a) = a$ for all $a \in A$.

21. Projections

    Projections are useful functions that may be used to select elements from pairs and n-tuples.

    Given any product $A \times B$ we define the functions

    $$\pi_1 \triangleq \{((a, b) \mapsto a) \in (A \times B) \times A \mid (a, b) \in A \times B\}$$

    $$\pi_2 \triangleq \{((a, b) \mapsto b) \in (A \times B) \times B \mid (a, b) \in A \times B\}$$

    You may check that for the functions π_1 and π_2 just defined we have

    $$\pi_1 \in (A \times B) \to A$$

    $$\pi_2 \in (A \times B) \to B$$

    For example, $\pi_1((\text{“}luis\text{”}, 45)) = \text{“}luis\text{”}$, and $\pi_2((\text{“}luis\text{”}, 45)) = 45$.

    Projections generalize to n-tuples, for example, we may define the projections π_3, π_4, etc, which operate on triples, 4-tuples, etc.

1.2 Solved modeling problems

1. Model the following system with a structure.

   A lamp with two states `ON` and `OFF`.

   (a) Model the set of states of a lamp with a set $SLAMP$.

   (b) Define a function in $SLAMP \to SLAMP$ that models the "turn on" operation.

   (c) Define a function in $SLAMP \to SLAMP$ that models the "turn off" operation.

   (d) Define a function in $SLAMP \to BOOL$ that returns the current state of the lamp.

   Solution The set of states:

   $$SLAMP = \{0, 1\}$$

   The function of (b)

   $$turn\_on \triangleq \{0 \mapsto 1, 1 \mapsto 1\}$$

   The function of (c)

   $$turn\_off \triangleq \{0 \mapsto 0, 1 \mapsto 0\}$$

   The function of (d)

   $$status \triangleq \{0 \mapsto FALSE, 1 \mapsto TRUE\}$$

   The structure modeling the system:

   $$LAMP \triangleq (SLAMP, turn\_on, turn\_off, status)$$

2. Model the following system with a structure.

   A counter keeps the count of cars inside a tunnel by keeping track if cars entering the tunnel and cars exiting the tunnel.

   (a) Model the set of states of a counter with a set $SCOUNTER$.

   (b) Define a function in $SCOUNTER \to SCOUNTER$ that models the "car enter" operation.

   (c) Define a partial function in $SCOUNTER \to SCOUNTER$ that models the "car exit" operation.

(d) Define a function in $SCOUNTER \to NAT$ that yields the number of cars currently inside the tunnel.

Solution The set of states:

$$SCOUNTER \triangleq NAT$$

The function of (b)

$$car\_enter \triangleq \{n \mapsto m \in NAT \times NAT \mid m = n + 1\}$$

The function of (c)

$$car\_exit \triangleq \{n \mapsto m \in NAT \times NAT \mid n = m + 1\}$$

The function of (d)

$$cars\_inside \triangleq id_{NAT}$$

The structure modeling the system:

$$COUNTER \triangleq (SCOUNTER, car\_enter, car\_exit, cars\_inside)$$

3. Model the following data with sets

   (a) The set of all bank accounts, where each bank account includes the owner name, the account number, and the balance.

   (b) Define a function $JOIN$ that given a set of bank accounts B without repeated account numbers, and two account numbers in B, yields a set of bank accounts identical to the given one, except that the two given accounts are merged in a new account, under the number of (and owner of) smallest account number.

   (c) To what set belongs the function $JOIN$?

Solution We may first define the sets, just for convenience,

$$\begin{array}{lcl} NAME & \triangleq & STRING \\ ACCNUM & \triangleq & NAT \\ AMOUNT & \triangleq & NAT \end{array}$$

(a) The set of all bank accounts

$$ACC \triangleq NAME \times ACCNUM \times AMOUNT$$

An example of a bank account

$$(\text{“luis”}, 1024, 80000000000)$$

We have $(\text{“luis”}, 1024, 80000000000) \in ACC$

(b) Any set of bank accounts B is a subset of ACC, in other words, a member of $\wp(ACC)$.

For any set $B \in \wp(ACC)$ and account numbers n_1 and n_2 in B, we define the set

$$\begin{array}{l} \mathit{merge}(B, n_1, n_2) \\ \triangleq \\ \{c \in B \mid \pi_2(c) \neq n_1 \wedge \pi_2(c) \neq n_2\} \\ \cup \\ \{(o, n, b) \in ACC \mid \\ \qquad n = \mathit{min}(n_1, n_2) \wedge \exists b_1.\exists b_2.(o, n_1, b_1) \in B \wedge (o, n_2, b_2) \in B \wedge b = b_1 + b_2\} \end{array}$$

The first part of the union contains the accounts in B that are not the accounts with numbers n_1 or n_2.

The second part of the union contains the "joined" account.

The function JOIN can then be defined

$$\mathit{JOIN} \triangleq \{(S, n_1, n_2) \mapsto M \mid M = \mathit{merge}(S, n_1, n_2)\}$$

(c) We have

$$\mathit{JOIN} \in (\wp(ACC) \times ACCNUM \times ACCNUM) \to \wp(ACC)$$

1.3 Inductive Definitions

We have discussed several ways to define sets, for example, by enumeration, by comprehension, and by applying set operations to previously defined sets.

Another fundamental way of defining sets, particularly useful in informatics and computer science, is the so-called **inductive definition**.

Induction is an extremely powerful technique, and plays in set theory a role similar to the one recursion plays in programming (this remark is only for those of you that already know what is a recursive function or recursive procedure in a programming language).

Using induction, we define sets using an incremental construction method, by adding in stages to previously built stages, as if we were building skyscrapers from their foundations.

Actually, the basic idea is quite simple.

First, we enumerate a (finite) set of basic elements that must belong to the set we want to define. We can think of these basic elements as some kind of "seeds".

You may imagine the "seeds" as being the "basic" elements of the set. This elements will be created in stage 0.

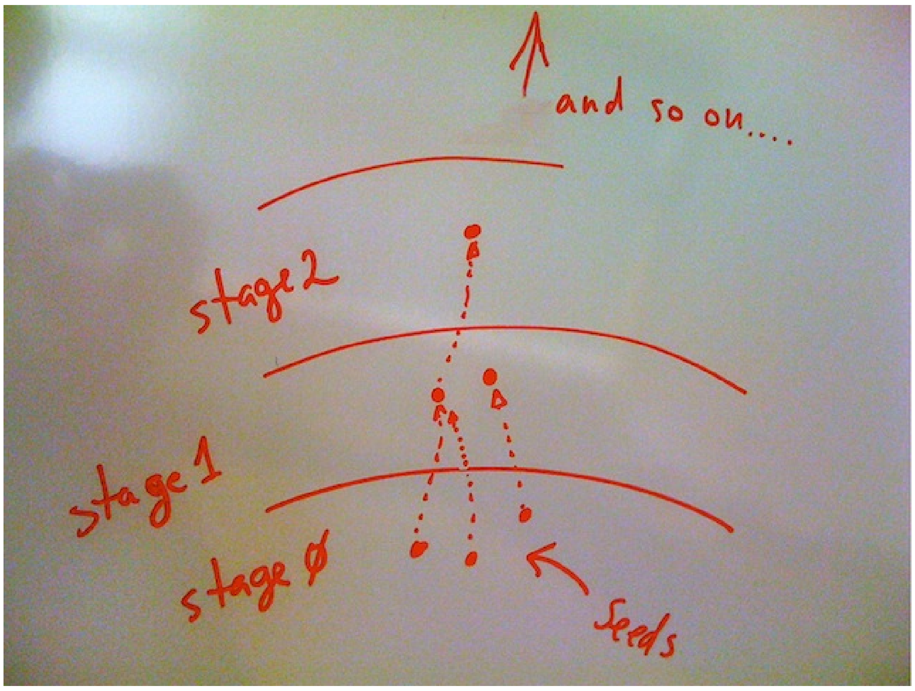


Figure 1: Building a set from the seeds, using rules to generate new elements.

Then, we add a new stage of elements to the set, these "new" elements must be calculated from the "seeds" according to some fixed rule. That will be the second stage.

Then, we add a third stage of elements to the set, calculated from the elements in level two, according to the same fixed rule.

And so on, and on ... indefinitely, to the infinite.

Obviously, we cannot in general implement the whole generation process of the complete inductive set as an algorithm. But the mechanism of induction gives us **for free** the inductive set (which in general contains an infinite number of elements) for granted automatically: we just have to say what are the seeds, and what are the generation rules. Both the seeds and the rules are finite in number, and we can easily write them down.

As a first, example, we consider an inductive definition of the set of natural numbers (supposing that it was not yet defined). First the "seed" (there is only one seed in this case, which is the simplest natural number, namely 0). We thus define:

$$\Longrightarrow 0 \in N$$

This "seed" rule asserts that 0 is in the set N, and defines the first layer of

N, which contains just 0. We then need a "construction" rule, that allows us to add new natural numbers to the set, based on elements already defined in previous layers. The rule looks like:

$$x \in N \Longrightarrow succ(x) \in N$$

We may read this construction rule as: If x is an element of the set N, then $succ(x)$ must also be an element of N. Here we have represented by $succ(x)$ the successor of x, e.g., $succ(2) = 3$.

The complete inductive definition of N is then as follows:

$$\begin{array}{rl} ZERO: & \Longrightarrow 0 \in N \\ SUCC: & x \in N \Longrightarrow succ(x) \in N \end{array}$$

It contains two rules, one seed rule and one construction rule.

This inductive definition defines a set N, the set that contains **all** the elements and **only** the elements that may be generated by the rules shown.

Note that we gave names to the rules in the inductive definition, the first rule is called *ZERO* and the second rule is called *SUCC*. To name rules in an inductive definitions, we may invent illustrative names, there is no fixed recipe to give names to rules.

In general, an inductive definition may include any number of seed rules and any number of construction rules, as we will see in forthcoming examples, although in the simple example we have only one seed rule and one construction rule.

A fundamental property of any inductively defined set S is that ANY element $e \in S$ is always justified by a finite number of applications of construction rules, always starting from one or more seed rules.

For example, we have $4 \in N$.

What is the justification of the fact $4 \in N$, according to the inductive definition given above?

It is easy:

- We know that $0 \in N$ by the ZERO (seed) rule!
- We conclude $1 \in N$ by applying the SUCC (construction) rule to $0 \in N$.
- We conclude $2 \in N$ by applying the SUCC (construction) rule to $3 \in N$.
- We conclude $3 \in N$ by applying the SUCC (construction) rule to $2 \in N$.
- We conclude $4 \in N$ by applying the SUCC (construction) rule to $3 \in N$.

We will now go through a sequence of examples of inductive definitions of sets.

Remember that in set theory, a data domain is a set, a function is a set, a relation is a set, and we can also model properties as a set. We will show below how the basic technique of inductive definitions can be used to inductively define functions, relations, properties, data domains, and so on!

1. Example: Even numbers

   Consider the set $EVENN$ of even natural numbers. We have already provided a definition of $EVENN$ using comprehension. We now provide an alternative inductive definition.

$$\begin{array}{ll} ZERO: & \Longrightarrow 0 \in EVENN \\ DUP: & x \in EVENN \Longrightarrow x+2 \in EVENN \end{array}$$

2. Example: An inductively defined data type

   Consider the set of all finite sequences of natural numbers SEQ. A sequence may be represented in set theory by an n-tuple (see Section 1(16)).

   Let us now define SEQ using an inductive definition (is is not really possible to precisely define this set using either set enumeration or set comprehension / separation).

$$\begin{array}{ll} \text{EMPTY}: & \Longrightarrow \emptyset \in SEQ \\ \text{ONEMORE}: & s \in SEQ \;;\; x \in NAT \Longrightarrow (x, s) \in SEQ \end{array}$$

   The (seed) rule EMPTY introduces the empty sequence (represented here by the empty set) in the set SEQ.

   The (construction) rule ONEMORE introduces a new sequence in the set SEQ by adding an arbitrary natural number as the new first element to an already introduced sequence.

   Notice that the ONEMORE rule constructs a new element in the set SEQ not only from some existing element $s \in SEQ$, but also from any existing element $n \in NAT$.

   For example, here is the justification that $(3, 4, 2, 4) \in SEQ$.

   - We know that $() \in SEQ$ by the EMPTY rule!
   - We conclude $(4, \emptyset) \in SEQ$ by applying the ONEMORE rule to $() \in SEQ$ and $4 \in NAT$. Notice that $(4, \emptyset) = (4)$.

- We conclude $(2,(4,\emptyset)) \in \mathit{SEQ}$ by applying the ONEMORE rule to $(4,\emptyset) \in \mathit{SEQ}$ and $2 \in NAT$. Notice that $(2,(4,\emptyset)) = (2,4)$.
- We conclude $(4,2,4) \in \mathit{SEQ}$ by applying the ONEMORE rule to $(2,4) \in \mathit{SEQ}$ and $4 \in NAT$.
- We conclude $(3,4,2,4) \in \mathit{SEQ}$ by applying the ONEMORE rule to $(4,2,4) \in \mathit{SEQ}$ and $2 \in NAT$.

3. General form of Induction Rules

   The last example shows the general format of rules in an inductive definition, which is as follows

   $$e_1 \in S_1 \; ; \; e_2 \in S_2 \; ; \; \cdots e_n \in S_n \Longrightarrow e \in U$$

   Each set S_i is either the name of the set U being inductively defined, or a set expression denoting any **already defined** set.

   The conditions $e_1 \in S_1, e_2 \in S_2, \cdots, e_n \in S_n$ are called the premises of the rule, and the $e \in U$ is called the conclusion of the rule.

4. Example: Inductively defined functions

   As we know a function is a set, a set of ordered pairs subject to the "functional" condition (see Section 1 (19)).

   We may define a function inductively as we have done above for sets.

   Let us illustrate the idea with the Fibonacci function. The Fibonacci function maps $n \in NAT$ to the n^{th} element in the Fibonacci sequence of natural numbers. Remember (this has to do with rabbits ☺) that the Fibonacci sequence is defined as follows. The first and second element in the sequence are both 1. From then on, the n^{th} element in the Fibonacci sequence is computed as the sum of the two previous ones.

   $$1, 1, 2, 3, 5, 8, \cdots$$

   The Fibonacci function *fib* then gives

   $$\begin{array}{l} fib(0) = 1 \\ fib(1) = 1 \\ fib(2) = 2 \\ fib(3) = 3 \\ fib(4) = 5 \\ etc... \end{array}$$

To define a function (such as fib) as an inductive set, we need to define a set of ordered pairs $a \mapsto b$ where a is an argument value and b is the corresponding function result.

In the case of the fib function is particularly easy to take care of with an inductive definition, because it is very clear what is the stage by stage construction rules needed!

First we need to introduce the two first values. It is clear that none of these values are computed from the other, they are both seeds, really.

$$\begin{aligned} &\Longrightarrow 0 \mapsto 1 \in fib \\ &\Longrightarrow 1 \mapsto 1 \in fib \end{aligned}$$

Recall that the function fib we are defining is a set of ordered pairs. The two rules above state that the set fib must contain the pairs $0 \mapsto 1$ and $1 \mapsto 1$. This means that $fib(0) = 0$ and $fib(1) = 1$. Actually, we could have written the rules above as

$$\begin{aligned} &\Longrightarrow fib(0) = 1 \\ &\Longrightarrow fib(1) = 1 \end{aligned}$$

since saying $a \mapsto b \in F$ is the same as saying $F(a) = b$.

Now we need a construction rule, to generate new values for the function fib from "previous" ones. For the Fibonacci function, the rule is simply

$$n \mapsto a \in fib \ ; \ n+1 \mapsto b \in fib \Longrightarrow (n+2) \mapsto (a+b) \in fib$$

This says that if we already know that (at a previous stage) $fib(n) = a$ and $fib(n+1) = b$, then we can define that $fib(n+2) = a+b$.

$$fib(n) = a; fib(n+1) = b \Longrightarrow fib(n+2) = a+b$$

We now summarize our inductive definition for the function fib, now labeling the rules with names.

$$\begin{array}{ll} \text{FIB0}: & \Longrightarrow fib(0) = 1 \\ \text{FIB1}: & \Longrightarrow fib(1) = 1 \\ \text{FIBNEXT}: & fib(n) = a; fib(n+1) = b \Longrightarrow fib(n+2) = a+b \end{array}$$

We can for example check that $fib(4) = 5$ by writing down the justification, in terms of the available induction rules.

(a) We conclude $0 \mapsto 1 \in fib$ by the FIB0 rule.

(b) We conclude $1 \mapsto 1 \in \mathit{fib}$ by the FIB1 rule.

(c) We conclude $2 \mapsto 2 \in \mathit{fib}$ by applying the FIBNEXT rule to $0 \mapsto 1 \in \mathit{fib}$ and $1 \mapsto 1 \in \mathit{fib}$ introduced in (a) and (b).

(d) We conclude $3 \mapsto 3 \in \mathit{fib}$ by applying the FIBNEXT rule to $1 \mapsto 1 \in \mathit{fib}$ and $2 \mapsto 2 \in \mathit{fib}$ introduced in (b) and (c).

(e) We conclude $4 \mapsto 5 \in \mathit{fib}$ by applying the FIBNEXT rule to $2 \mapsto 2 \in \mathit{fib}$ and $3 \mapsto 3 \in \mathit{fib}$ introduced in (c) and (d).

5. Example: The *sumupto* function

We seek an inductive definition of the *sumupto* function such that

$$\mathit{sumupto}(k) = 1 + 2 + 3 + ... + k$$

for any $k \in NAT$. Notice that this "definition" is not a precise one, and uses "hand waving" notation such as "...", etc.

We can provide a precise inductive definition as follows:

$$\begin{array}{ll} \text{SUM0}: & \Longrightarrow 0 \mapsto 0 \in \mathit{sumupto} \\ \text{SUMNEXT}: & n \mapsto s \in \mathit{sumupto} \Longrightarrow (n+1) \mapsto (n+1+k) \in \mathit{sumupto} \end{array}$$

or, perhaps more readably,

$$\begin{array}{ll} \text{SUM0}: & \Longrightarrow \mathit{sumupto}(0) = 0 \\ \text{SUMNEXT}: & \mathit{sumupto}(n) = s \Longrightarrow \mathit{sumupto}(n+1) = n + 1 + s \end{array}$$

This inductive definition defines the intended function *sumupto*. For example, we may check the justification that $\mathit{sumpupto}(4) = 10$

(a) We conclude $0 \mapsto 0 \in \mathit{sumupto}$ by the SUM0 rule.

(b) We conclude $1 \mapsto 1 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $0 \mapsto 0 \in \mathit{sumupto}$ introduced in (a).

(c) We conclude $2 \mapsto 3 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $1 \mapsto 1 \in \mathit{sumupto}$ introduced in (b).

(d) We conclude $3 \mapsto 6 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $2 \mapsto 3 \in \mathit{sumupto}$ introduced in (c).

(e) We conclude $4 \mapsto 10 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $3 \mapsto 6 \in \mathit{sumupto}$ introduced in (d).

6. Example: The *len* function

   We seek an inductive definition of the *let* function, that given a sequence of naturals, that is, an element of SEQ as defined above in 2, returns the length of the sequence, for example:

   $$len((1,2,3,4)) = 4$$

   So, we expect $len \in SEQ \to NAT$.

   Here is our inductive definition for the function *len*.

   $$\begin{array}{ll} \text{LENEMPTY :} & \Longrightarrow () \mapsto 0 \in len \\ \text{LENONEMORE :} & s \mapsto l \in len \Longrightarrow (h,s) \mapsto (l+1) \in len \end{array}$$

   or, perhaps more readably,

   $$\begin{array}{ll} \text{LENEMPTY :} & \Longrightarrow len(()) = 0 \\ \text{LENONEMORE :} & len(s) = l \;;\; h \in NAT = l \Longrightarrow len(h,s) = (l+1) \end{array}$$

   This definition inductively defines the intended function *len*. For example, we may check the justification that $len((4,2,4)) = 3$.

   Recall that $(4,2,4) = (4,(2,(4,())))$! Then

   (a) We conclude $() \mapsto 0 \in len$ by the LENEMPTY rule.
   (b) We conclude $(4) \mapsto 1 \in len$ by applying the LENONEMORE rule to $() \mapsto 0 \in len$ introduced in (a) and $4 \in NAT$.
   (c) We conclude $(2,4) \mapsto 2 \in len$ by applying the LENONEMORE rule to $(4) \mapsto 1 \in len$ introduced in (b) and $2 \in NAT$.
   (d) We conclude $(4,2,4) \mapsto 3 \in len$ by applying the LENONEMORE rule to $(2,4) \mapsto 2 \in len$ introduced in (c) and $4 \in NAT$.

7. Top-down and Bottom-up justifications.

   In the previous examples, we have given formal justifications of the fact that an element e belongs to an inductive set I by showing the sequence of rule applications that lead from the seed rules (the most basic elements) to the final construction of e.

   This style of presentation is called a **bottom-up** justification.

   For example, the justification

   (a) We conclude $() \mapsto 0 \in len$ by the LENEMPTY rule.

(b) We conclude $(4) \mapsto 1 \in \mathit{len}$ by applying the LENONEMORE rule to $() \mapsto 0 \in \mathit{len}$ introduced in (a) and $4 \in NAT$.

(c) We conclude $(2,4) \mapsto 2 \in \mathit{len}$ by applying the LENONEMORE rule to $(4) \mapsto 1 \in \mathit{len}$ introduced in (b) and $2 \in NAT$.

(d) We conclude $(4,2,4) \mapsto 3 \in \mathit{len}$ by applying the LENONEMORE rule to $(2,4) \mapsto 2 \in \mathit{len}$ introduced in (c) and $4 \in NAT$.

is bottom-up. It starts by the seed rule $\mathit{len}(()) = 0$ and then proceeds by rule application until the conclusion $\mathit{len}((4,2,4)) = 3$ of the justification is reached.

However, we can also provide justifications the other way around.

We do that by starting from the element that we want to justify, and proceeding backwards down to the seed rules.

For example, the following is the top-down version of the justification above for $\mathit{len}((4,2,4)) = 3$.

(a) We conclude $(4,2,4) \mapsto 3 \in \mathit{len}$ because we can apply LENONEMORE rule to $(2,4) \mapsto 2 \in \mathit{len}$ and $4 \in NAT$.

(b) We conclude $(2,4) \mapsto 2 \in \mathit{len}$ because we can apply LENONEMORE rule to $(4) \mapsto 1 \in \mathit{len}$ and $2 \in NAT$.

(c) We conclude $(4) \mapsto 1 \in \mathit{len}$ because we can apply LENONEMORE rule to $() \mapsto 0 \in \mathit{len}$ and $4 \in NAT$.

(d) We have $() \mapsto 0 \in \mathit{len}$ by the LENEMPTY rule.

8. Example: The *concat* function

We seek an inductive definition of the *concat* function. The concat function accepts as arguments two sequences and gives the sequence obtained by concatenating them. For example,

$$\begin{array}{l} \mathit{concat}((),(1,9)) = (1,9) \\ \mathit{concat}((3,4),(4,6)) = (3,4,4,6) \\ \mathit{concat}((1),(2)) = (1,2) \end{array}$$

Clearly, we have $\mathit{concat} \in (SEQ \times SEQ) \to SEQ$.

Here is an inductive definition for the function *concat*.

$$\begin{array}{ll} \text{CEMPTY}: & s \in SEQ \Longrightarrow ((),s) \mapsto s \in \mathit{concat} \\ \text{CSTEP}: & (s,v) \mapsto r \in \mathit{concat}\ ;\ h \in NAT \Longrightarrow ((h,s),v) \mapsto (h,r) \in \mathit{concat} \end{array}$$

or, perhaps more readably,

$$\begin{array}{ll} \text{CEMPTY}: & s \in SEQ \Longrightarrow concat((), s) = s \\ \text{CSTEP}: & concat((s, v)) = r \ ; \ h \in NAT \Longrightarrow concat(((h, s), v)) = (h, r) \end{array}$$

Let us see the justification that $concat((4, 2), (1, 4)) = (4, 2, 1, 4)$. Recall that $(4, 2) = (4, (2, \emptyset))$, $(4, 2, 1, 4) = (4, (2, (1, (4, \emptyset))))$, etc !

This time, we write a top-down justification:

(a) We conclude $concat((4, 2), (1, 4) = (4, 2, 1, 4)$ because we can apply the CSTEP rule to $concat((2), (1, 4) = (2, 1, 4)$ and $4 \in NAT$.

(b) We conclude $concat((2), (1, 4) = (2, 1, 4)$ because we can apply the CSTEP rule to $concat((), (1, 4) = (1, 4)$ and $2 \in NAT$.

(c) We conclude $concat((), (1, 4) = (1, 4)$ by the the CEMPTY rule.

1.4 Finite Sets, Infinite Sets, and Computability

The notions of finiteness and infiniteness play a central role in the study of computation. Right from the start, it allows us to separate the notion of function (as we have modeled mathematically with sets in the previous sections), which is an idealized concept, and the notion of algorithm, which is a very concrete computational method or machine.

Think of a given function, such as *concat* function defined above as a set, using induction. The *concat* function defined thus is a set of tuples, a correspondence between argument values and results, that defines the extension of the function. This extension is an infinite set, with elements such as

$$\begin{array}{l} ((1, 9, 2), (2, 3)) \mapsto (1, 9, 2, 2, 3) \in concat \\ ((), ()) \mapsto () \in concat \\ ((1, 1, 1, 1, 1), (2, 2, 2, 2, 2)) \mapsto (1, 1, 1, 1, 1, 2, 2, 2, 2, 2) \in concat \\ \ldots \end{array}$$

It is impossible to write down the full table of the concat function, obviously. But we can easily believe that such set exists at least in the world of our imagination and the current body of mathematical knowledge.

A very different thing would be an algorithm implementing the function concat. An algorithm is a mechanical process, that must be physically realizable by some kind of machine, build from a finite set of resources, consuming a finite set of energy for each run, and so on. An algorithm does not contain an infinite lookup table of correspondences between inputs and outputs, it

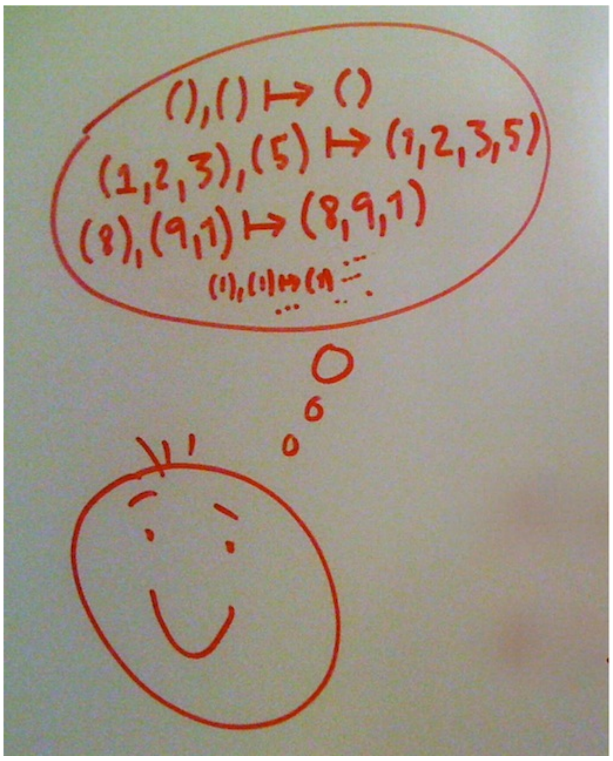


Figure 2: Me thinking about all the concat function pairs $(a, b) \mapsto ab$.

actually needs to calculate the output from the input, using a series of mechanical finite operations, for example, using the instructions of a processor inside a computer.

So an algorithm for computing *concat* must describe a very concrete and physically realizable information manipulating process, defined with some kind of machine or programming language. Such process, given two concrete sequences can laboriously compute a new sequence consisting of their concatenation. It is easy to define such a process, even in a programming language independent way:

1. pick the two input sequences s_1 and s_2
2. count the elements in s_1 and s_2 giving say l_1 and l_2
3. allocate space for a constructing a new sequence r long enough to keep the result $(l_1 + l_2)$ elements.
4. copy the elements of s_1 in sequence to the first l_1 elements of r.

5. copy the elements of s_2 in sequence to the elements of r in positions $l_1, l_1 + 1, \cdots, l_{1+l_2}$.
6. output the sequence r

We have then discussed two very different concepts:

- a function that specifies the concat function, which is a mathematical specification object, consisting of an infinite amount of information, and
- an algorithm that implements the concat function, which is a mechanical procedure that allows a "dummy" machine to compute its output from any given input.

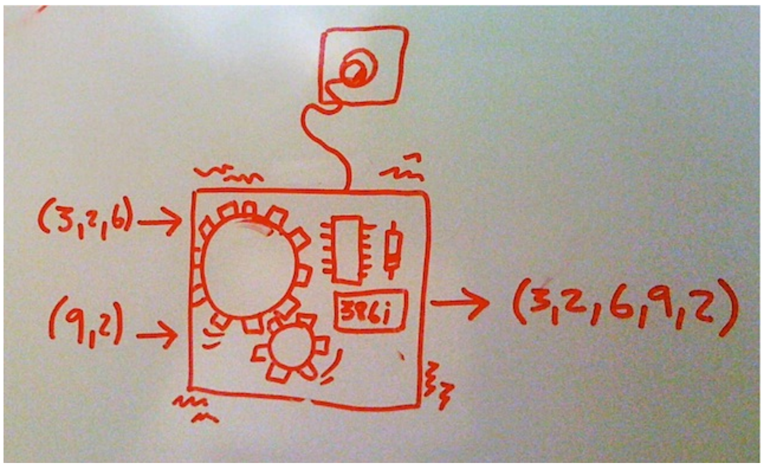


Figure 3: A machine implementing the concat function. It really needs to do some work for each input it is given, there is no place in physical reality for the machine to store an infinite table with all the pairs $(a, b) \mapsto concat(a, b)$.

Another example, may be a function *solution* that given a polynomial with integer coefficients returns *TRUE* if the polynomial has a solution or *FALSE* if it does not. We can imagine that *solution* receives its input in some textual format, and parses it (as say, excel would do).

The function *solution* is very easy to define, say

$$(\exists \vec{u_p} \sum_{i_1}^{p} k_i * {u_i}^{n_i} = v) \Rightarrow (\text{“}\sum_{i_1}^{p} k_i * {x_i}^{n_i} = v\text{”}) \mapsto \mathit{TRUE} \in \mathit{solution}$$

$$(\forall \vec{u_p} \sum_{i_1}^{p} k_i * {u_i}^{n_i} \neq v) \Rightarrow (\text{“}\sum_{i_1}^{p} k_i * {x_i}^{n_i} = v\text{”}) \mapsto \mathit{FALSE} \in \mathit{solution}$$

Every polynomial of the kind considered either has a solution or it does not have a solution, so this is a perfectly well defined total function.

But is there a well defined algorithm implementing this function ?

Indeed, may we find an algorithm actually able to compute the solution function, and effectively check, in a finite number of steps of computation, if any polynomial given as input as a solution or not?

This kind of question may be answered in several ways.

On possibility is to really give a description of an algorithm, as we did above for the concat function, or in some programming language. We may then convince ourselves that the algorithm is correct, and that's all.

Other possibility is that the current state of knowledge does not yet help us to define the algorithm, but it may be the case that some algorithm may be found, if we get smart enough.

Other possibility is that it will never be possible to give an algorithm, just because the solution function cannot be computed by algorithmic means. In this case, we are not thinking of any limitation of the current state of the knowledge, or that we are not smart enough currently to come up with an algorithm, but of the absolute impossibility of solving the problem using any finite method of computation whatsoever!

There are necessarily many functions we can reasonably conceive that cannot be implemented by algorithmic means.

The simplest way to see this is just by counting: there are many more possible functions that algorithms!

Hum .. wait?! How can that be?

Both the set of possible functions and of algorithms is infinite!

How can we say that there are more possible functions than algorithms?

To answer this questions, we need to review a bit the theory of infinite cardinals, and get a better grasp of what does it mean to be "finite" and "infinite". We will also learn that there are several levels of infinity, and that the cardinal of the set of natural numbers is but the "smallest" infinite, in an infinitely increasing sequence of infinite cardinal numbers.

Understanding these basis concepts of infinity is very important to recognize whether a problem is computable by an algorithm, or if it is not (in which case we call the problem "undecidable").

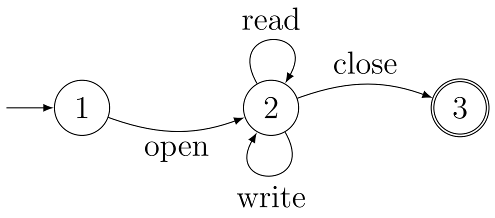
2 Computational Machines and Specifications

We have seen in the previous chapter how various kinds of computational systems may be modeled by set-theoretical structures.

Typically, such structures model a set of states, and a collection of functions and relation that model the dynamics of the system, namely, the operations that change the state, and cause the system to evolve in time.

In particular, a computation of the system is modeled by a sequence of applications of functions or relations between states.

In this chapter we will study a very general way of modeling computational systems of the above kind, but that only possess a finite number of states. As we will see these models, known as **finite automata**, are already interesting enough and can be used to model many useful computational systems. For example, consider the following graphical representation of a finite automaton, representing the correct sequence of operations one may perform in a file.



It contains three states 1, 2 and 3. The state 1, represented with a simple incoming arrow, in the initial state. The state 3, represented with two concentric circles is a final state. There is also an intermediate state 2, which is neither initial nor final. The labeled arrows represent transitions of the automaton, the transitions are labeled with elements called actions (or symbols).

A computation of this automaton is represented by the trace (or word), a sequence of actions (or symbols). An example of a valid computation is

`open read read write close`

Indeed starting from the initial state 1, it is possible to reach the final state 3, by performing each one of the given actions in the sequence, from left to right. On the other hand, the trace

`open open`

does not constitute a correct computation of the same automaton. Indeed, after going from the initial state to the second by performing `open`, we reach a state where no transition labeled by `open` exists. Thus, we say the the automaton rejects (or does not accept) the trace `open open`. This may be understood as meaning that a file cannot be opened twice.

[to be completed]

2.1 Deterministic Finite Automata

1. Deterministic Finite Automata (DFA) are very simple models of computation. Each DFA represents a computational system (software or hardware) characterized by a finite set of states, and that evolves from state to state by performing actions (also called transitions), also selected from a finite set of possible transitions.

   Formally, a DFA A is a structure

   $$A = \langle S, \Sigma, s, \delta, F \rangle$$

   where

   (a) S is the finite set of states of A

   (b) Σ is the finite set of actions (or symbols) of A

   (c) s is state in S, the initial state of A

   (d) δ is the transition function of A, that given a current state in S and an action/symbol in Σ, indicates the (unique) next state in S to which the automata should transition.

   So we have $\delta \in S \times \Sigma \to S$ where δ is in general a partial function.

   (e) F is a subset of S, the set F of final states of A.

   For example, the DFA sketched in Figure above may be formally represented, as explained in the previous definition, as the following structure:

   $$\begin{array}{lcl} AFILE & = & \langle S, \Sigma, s, \delta, F \rangle \\ \\ S & = & \{s_1, s_2, s_3\} \\ \Sigma & = & \{\texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}\} \\ s & = & s_1 \\ \delta & = & \{(s_1, \texttt{open}) \to s_2, (s_2, \texttt{read}) \to s_2, \\ & & (s_2, \texttt{write}) \to s_2, (s_2, \texttt{close}) \to s_3\} \\ F & = & \{s_3\} \end{array}$$

It is some times practical to represent the transition function of an DFA by a matrix or array, as we illustrate below:

	`open`	`write`	`read`	`close`
s_1	s_2	$-$	$-$	$-$
s_2	$-$	s_2	s_2	s_3
s_3	$-$	$-$	$-$	$-$

The entries where $\delta(s, a)$ is undefined are marked $-$.

2. A computation of a DFA is any sequence of actions it may perform, starting from the initial state s, and leading to any final state in F.

   For example, the DFA $AFILE$ above has, among many others, the following computations:

   ```
   open close
   open read read close
   open read write read close
   ```

   We may represent the system configuration of a DFA by a pair consisting of a marked sequence of symbols, and the current state. In a marked sequence of symbols we mark with a vertical bar | the separation between the actions / symbols already performed and the actions / symbols still to be performed, and where the first symbol to the right of | is the next to be processed. We may imagine that the current state is the analogous of the program counter, in a standard processor, and the marked sequence of actions some kind of input buffer.

   So, for our current example with the DFA AFILE, we may consider the initial system configuration

   $$(|\texttt{open close}, s_1)$$

   After one transition, because $\delta(s_1, \texttt{open}) = s_2$, the configuration evolves to

   $$(\texttt{open}|\texttt{close}, s_2)$$

   After one more transition, because $\delta(s_2, \texttt{close}) = s_3$, the configuration evolves to

   $$(\texttt{open close}|, s_3)$$

   Since we have reached the end of the sequence and s_3 is a final state ($s_3 \in F$), we can say that the sequence of symbols `open close` is a computation of $AFILE$. There are several different ways to say this, namely

- the sequence `open close` is a computation of $AFILE$.
- the sequence `open close` is accepted by $AFILE$.

We may use any of these ways, depending on the context.

On the other hand, the following sequences of symbols / actions are not computations of $AFILE$.

```
open read
open read open close
```

In the first case, we may try to obtain a computation starting from the configuration

$$(|\texttt{open read}, s_1)$$

After on step, we get to

$$(\texttt{open}|\texttt{read}, s_2)$$

After other step, we get to

$$(\texttt{open read}|, s_2)$$

We have reached the end of the sequence of symbols, but did not reach a final state. Indeed, the computation attempt did not terminate, it got stuck in the non-final state s_2. So, the sequence `open read` is not accepted by the DFA AFILE.

Likewise, consider the sequence `open read open close`. We start off as

$$(|\texttt{open read open close}, s_1)$$

After one step, we get to

$$(\texttt{open}|\texttt{read open close}, s_2)$$

After one more step, we get to

$$(\texttt{open read}|\texttt{open close}, s_2)$$

Now the problem is that the next symbol to process in `open`, but unfortunately the transition function δ does not define any valid transition. Indeed $\delta(s_2, \texttt{open})$ is not defined! So, this computation atempt did not properly terminate as well, and we conclude that the sequence `open read open close` is not accepted by the DFA $AFILE$.

3. DFAs are very convenient also because they lead to very efficient and simple implementations of trace recognizers. Given an array representation of the transition function, we may simply implement the Java like pseudo-code below, to check if an input sequence *input* is accepted by the given DFA.

   ```
   boolean accept(Word input) {
     State currentState = initialState;
     while input.hasNext() {
        State nextState = delta[ currentState, input.next() ];
        if (nextState.undefined()) return false;
        currentState = nextState;
      }
     return (currentState.isFinal())
     }
   ```

   We have here represented a word by an iterator of symbols. Of course, other representations are possible.

4. For any DFA $A = \langle S, \Sigma, s, \delta, F \rangle$, it makes sense to talk about the set of all computations of Am, or, equivalently, about the set of all sequences of symbols accepted by A. This set of sequences, is called the language accepted by the DFA A, noted $\mathcal{L}(A)$.

   $$\mathcal{L}(A) = \{w \in \mathit{Words}(\Sigma) \mid w \text{ is accepted by the DFA } A\}$$

   This is not a precise definition, since we did not yet explain what is the set $Words(\Sigma)$ and what does it really mean to "be accepted". For that, we need to introduce some few concepts.

2.2 Alphabets, words, traces, and (formal) languages

We have talked about sets of actions, sequences of actions, symbols, etc, when discussing DFAs.

All these notions involving sequences of symbols, are abstracted and studied in theoretical computer science by the notion of "formal language". The concept of formal language is very general and useful, and is not just connected to "languages" in the usual sense. We will give some examples of applications below. But before, we introduce a few useful definitions.

1. Alphabet

An alphabet is just a finite set of symbols (also actions). We may define alphabets as we wish, typically by enumeration. For example,

$$\begin{aligned} \Sigma_{FILE} &= \{\texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}\} \\ \mathit{DIGITS} &= \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \} \end{aligned}$$

2. Word (or trace)

   A word (or trace) over an alphabet Σ is a finite, possibly empty, sequence of symbols taken from Σ.

   We write () for the empty word. Some texts also represent the empty word by λ or ϵ.

   Here are some examples:

   `close close write write` is a word over Σ_{FILE}.

   `open read close close` is a word over Σ_{FILE}.

   `open 1 2 close` is not a word over Σ_{FILE}.

   () is a a word over Σ_{FILE}.

   `1 1 1 1` is a word over $DIGITS$.

   () is a word over $DIGITS$.

   `1 a b 1` is not a word over $DIGITS$.

3. Extending one word by one symbol.

   Given a word u over Σ and a symbol $a \in \Sigma$, we denote by au the word over Σ obtained by adding a before the first symbol of u.

   For example, if u is `1 1 1 1` and a is `3` then au is `3 1 1 1 1`.

   For example, if u is () and a is `2` then au is `2`.

   Note that in rigor one thing is a symbol, and other is a word with just one symbol (like one thing is an integer, and other thing is a list with length one containing the same integer). We shall never make confusions, though.

4. The set $\mathit{Words}(\Sigma)$ of all words (or traces) over the alphabet Σ.

   Given an alphabet Σ, we may easily define the set $\mathit{Words}(\Sigma)$ of all words over the alphabet Σ, using induction, as discussed in the first chapter.

   $$\begin{aligned} &\Rightarrow () \in \mathit{Words}(\Sigma) \\ &w \in \mathit{Words}(\Sigma) \land a \in \Sigma \Rightarrow aw \in \mathit{Words}(\Sigma) \end{aligned}$$

It should be pretty clear that whenever the alphabet Σ is non empty, the set $\mathit{Words}(\Sigma)$ of all words over Σ is countably infinite.

We may now define what is a (formal) language over an alphabet.

5. Formal language over an alphabet Σ

   Given an alphabet Σ, a formal language over Σ is any subset L of $Words(\Sigma)$. So the set $\mathit{Lang}(\Sigma)$ of all languages over Σ is

   $$\mathit{Lang}(\Sigma) = \wp(\mathit{Words}(\Sigma))$$

   For a simple example, we may consider the set PRIMES of all words in $\mathit{Words}(\mathit{DIGITS})$ that represent prime numbers. This is the example of what we cal a "formal language". Then, we know that

   $$\texttt{11} \in \mathit{PRIMES}$$

   but

   $$\texttt{22} \notin \mathit{PRIMES}$$

   although $\texttt{11} \in \mathit{DIGITS}$ and $\texttt{22} \in \mathit{DIGITS}$.

   As another example, we may consider the set $\mathcal{L}(AFILE)$ of all words over Σ_{FILE} that are accepted by the DFA FILE, presented above. Then, we know that

   $$\texttt{open read close} \in \mathcal{L}(AFILE)$$

   but

   $$\texttt{open open} \notin \mathcal{L}(AFILE)$$

   even if clearly $\texttt{open read close} \in \Sigma_{FILE}$ and $\texttt{open open} \in \Sigma_{AFILE}$.

   These are two simple examples of formal languages. Obviously, a language may be finite or infinite. It is easy to see that both PRIMES and $\mathcal{L}(AFILE)$ are infinite languages.

   We may now get back to AFDs, and to the precise definition of what does it mean for a word / trace to be accepted by an automaton.

2.3 Language Accepted by a DFA

6. Acceptance of a word by a AFD A

   Given any DFA $A = \langle S, \Sigma, s, \delta, F\rangle$, we may define a relation

   $$\mathit{step}_A \subseteq S_A \times Words(\Sigma_A) \times S_A$$

that represents the multistep transition relation for the DFA A.

Intuitively, we want the relation $step_A$ to be defined so that $(s, w, s') \in step_A$ if and only if the DFA A can transition from state s to state s' by processing the symbols in w in the indicated sequence.

For example, getting back to our running example DFA $AFILE$, we would have

$$\begin{array}{l}
(s_2, (), s_2) \in step_{AFILE} \\
(s_1, \texttt{open close}, s_3) \in step_{AFILE} \\
(s_1, \texttt{open close}, s_2) \notin step_{AFILE} \\
(s_2, \texttt{read read}, s_2) \in step_{AFILE} \\
(s_2, \texttt{read read write}, s_2) \in step_{AFILE} \\
(s_2, \texttt{open}, s_2) \notin step_{AFILE}
\end{array}$$

Given any DFA A, it is easy to define the relation $step_A$ inductively as follows

$$\begin{array}{l}
s \in S_A \Rightarrow (s, (), s) \in step_A \\
(s', u, s'') \in step_A \wedge a \in \Sigma_A \wedge \delta_A(s, a) = s' \Rightarrow (s, au, s'') \in step_A
\end{array}$$

7. Language accepted by an AFD A. Given the concepts presented above we may now give a precise definition for the language $\mathcal{L}(A)$ accepted by an AFD $A = \langle S, \Sigma, s, \delta, F \rangle$, intuitively described above by

$$\mathcal{L}(A) = \{w \in Words(\Sigma) \mid w \text{ is accepted by the DFA } A\}$$

Indeed, we may simply set

$$\mathcal{L}(A) = \{w \in Words(\Sigma) \mid \exists f. f \in F_A \wedge (s_A, w, f) \in step_A\}$$

Reading off this precise definition, $\mathcal{L}(A)$ is the set of words that can induce a sequence of computation steps from the initial state s_A to some final state $f \in F_A$, as conveniently expressed by the $step_A$ relation.

2.4 Regular Languages

A language for which it is possible to construct a DFA that accepts it is called a **regular language**.

DFAs have limited expressive power, mainly because they only possess a finite number of states. There are many languages which can not be accepted by DFAs. For example, the language *PRIMES* mentioned above is not regular.

Other simple example of non-regular languages are the so-called "parenthesis" languages. A "parenthesis" language is a language with a nested block structure, like the language with arithmetic expressions with parentheses $(1 + ..(...)... * 3)$, or a programming language with nested blocks, such as Java $\{\{...\}\}$. Intuitively, to make sure that all opening parenthesis are matched by corresponding closing parenthesis, a DFA would need to be prepared to "know" in every state how many parenthesis are currently open. But since a DFA only contains a finite number of states N, it could never handle an expression with say $N + 10$ nested blocks of parenthesis. We will study later in this course more powerful (stack-based) machines, which will be able to accept non-regular languages as the ones illustrated here.

Still, there many interesting computational systems that may be modeled by DFAs, and the simplicity of DFAs are attractive and convenient.

For example, the language of all words over *DIGITS* that represent numbers divisible by k, where k is any natural number, is regular.

The language over the UNICODE character set that represents all valid descriptions of floating numbers in IEEE754 format is regular.

You will see many other examples in the exercises.

2.5 Specifying Languages with Regular Expressions

We have seen how to define machines that can recognize formal languages, namely deterministic finite automata. While DFAs have a simple operational interpretation, as some kind of "low level machines", they are not so convenient to construct or specify in a user friendly way. The situation is similar to what happens with low level hardware processors. While a processor can easily compute the value of a complex arithmetic expression, or insert a node in a data structure, it is much more convenient to express the latter with an algebraic expression such as "(a*b)+2" than with a sequence of machine code instructions, even if both the algebraic expression and the sequence of machine code instructions will denote the same computation.

It turns out (perhaps surprisingly) that regular languages can be expressed by a kind of algebraic expressions, known for very good reasons as **regular expressions**.

8. Regular Expressions

Regular expressions are a specification (meta-)language for defining formal languages.

There are the following forms of regular expressions:

- The expression () denotes the (one word) language that only contains the empty word.
- The expression a, where a is a symbol / action of some alphabet, denotes the language that only contains the one-symbol word a.
- The expression $\emptyset$, denotes the empty language (that is the language without any word, the empty set).

  These are the basic regular expressions. To denote more complex languages, we now introduce three operators to construct new languages from existing ones.
- Given expressions E and F, the expression (EF) denotes the set of words that result from concatenating some word in the language denoted by E with some word in the language denoted by F.
- Given expressions E and F, the expression $(E + F)$ denotes the set of words that are either some word in the language denoted by E or some word in the language denoted by F.
- Given an expression E, the expression (E^*) denotes the set of words that are the concatenation of zero or more words in the language denoted by E.

We may use parenthesis to disambiguate expressions. To eliminate redundant parenthesis, we assume that E^* is stronger that EF, and EF is stronger that $E + F$. So that for example

$$a * b + cd^*(a + b)$$

means

$$((a*)b) + ((c(d^*)(a + b))$$

We may easily describe the language accepted by the DFA $AFILE$ presented above with a regular expression over the alphabet Σ_{FILE}:

$$\texttt{open}(\texttt{read} + \texttt{write})^*\texttt{close}$$

Notice how this regular expression is much more concise than the description of the DFA $AFILE$. On the other way, a regular expression does not give us any hint on how to check if a word belongs to the language it denotes, unlike a DFA, which immediately provides an algorithm. We will see below how to close this gap.

In particular, we will see that any regular language (accepted by a DFA) can be represented by a regular expression, and conversely that any language denoted by a regular expression can be implemented by a DFA.

But before that, we will now define the set $\mathit{RegExp}(\Sigma)$ of regular expressions over an alphabet Σ in a precise way, and characterize the language denoted by any such regular expression.

9. Language specified by a regular expression

   The set $\mathit{RegExp}(\Sigma)$ is defined inductively as follows:

$$\begin{array}{l}
\Rightarrow () \in \mathit{RegExp}(\Sigma) \\
a \in \Sigma \Rightarrow a \in \mathit{RegExp}(\Sigma) \\
E \in \mathit{RegExp}(\Sigma) \wedge F \in \mathit{RegExp}(\Sigma) \Rightarrow (EF) \in \mathit{RegExp}(\Sigma) \\
E \in \mathit{RegExp}(\Sigma) \wedge F \in \mathit{RegExp}(\Sigma) \Rightarrow (E + F) \in \mathit{RegExp}(\Sigma) \\
E \in \mathit{RegExp}(\Sigma) \Rightarrow (E^*) \in \mathit{RegExp}(\Sigma)
\end{array}$$

   We know that any regular expression $E \in \mathit{RegExp}(\Sigma)$ specifies a language over Σ. In fact, we will denote by $L(E)$ the language denoted by the expression E.

   For example

$$L(\mathtt{read} + \mathtt{write}) = \{\mathtt{read}, \mathtt{write}\}$$

   and

$$L((\mathtt{read} + \mathtt{write})^*) = \mathit{Words}(\{\mathtt{read}, \mathtt{write}\})$$

   We can define the set $L(E)$ for any regular expression $E \in \mathit{RegExp}(\Sigma)$ by induction.

   First, given a language L, we may define the set $\mathit{loop}(L)$ of all words obtained by concatenating zero or more words from L. This is useful for defining the meaning of a regular expression of the form E^*.

$$\begin{array}{l}
\Rightarrow () \in \mathit{loop}(L) \\
u \in L \wedge v \in \mathit{loop}(L) \Rightarrow uv \in \mathit{loop}(L)
\end{array}$$

   Given this, we may provide the inductive definition of $L(E)$ for any

$E \in \mathit{RegExp}(\Sigma)$ as follows:

$$\begin{array}{l}
\Rightarrow L(()) = \{()\} \\
a \in \Sigma \Rightarrow L(a) = \{a\} \\
L(E) = L_E \wedge L(F) = L_F \Rightarrow L(EF) = \{uv \mid u \in L_E \wedge v \in L_F\} \\
L(E) = L_E \wedge L(F) = L_F \Rightarrow L(E+F) = L_E \cup L_F \\
L(E) = L_E \Rightarrow L(E^*) = \mathit{loop}(L_E)
\end{array}$$

2.6 Compiling any Regular Expression to a DFA

We have claimed that any regular expression can be "compiled" or translated to a DFA that recognizes the language denoted by it. In fact, we can present an algorithm (a "compiler"), that given a regular expression E generates a equivalent DFA in this sense.

In fact, the translation process is very simple and intuitive. How shall we proceed? In general, a regular expression may be very complex! A typical way to deal with this situation is to proceed by composition of parts, exactly as compilers for programming languages work. To cope with the structural complexity of programming languages, a compiler translates each form of the source language in a way that does not depend on the parts of each given construct. For example, consider a if the else construct:

$$\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}, C_2$$

When a compiler translates such a command into machine code, it does not really need to look on the translation of the sub expressions or sub commands B, C_1 and C_2. Essentially, it assumes that somehow these parts will be translated into appropriate sequences of machine code instructions, according to the general translation procedure, recursively.

$$\begin{array}{l}
B \mapsto \mathit{code}(B) \\
C_1 \mapsto \mathit{code}(C_1) \\
C_2 \mapsto \mathit{code}(C_2)
\end{array}$$

It then assembles the various pieces, by adding the "glue" code to put together the final code for the given if then else command. Typically this would involve adding machine code instructions to check the condition, and jump to the appropriate block of code. For example:

$$\begin{array}{rl} & code(B) \\ & \texttt{jmpnz}\ LabelElse \\ & code(C_1) \\ & \texttt{jmp}\ Done \\ & code(C_2) \\ Done: & \texttt{nop} \end{array}$$

The idea behind our translation of regular expressions will follow the same recipe. We first provide a way of translating the basic regular expressions, and then a systematic pattern of translating the complex constructs of concatenation (EF), union ($E + F$) and iteration E^*.

We then seek a function *Compile* that given a regular expression E yields a DFA $Compile(E)$. We proceed by looking to some simple cases.

- Case of ().

  If the regular expression we want to translate is (), what would be the corresponding DFA?

  The answer is simple, we may set:

  More precisely,

  $$Compile(\,()\,) = \langle S, \Sigma, s, \delta, F, \rangle$$

  where

  $$\begin{array}{rcl} S & = & \{1\} \\ \Sigma & = & \emptyset \\ s & = & 1 \\ \delta & = & \emptyset \\ F & = & \{1\} \end{array}$$

- Case of a, with $a \in \Sigma$

  If the regular expression we want to translate is a for some symbol a, what would be the corresponding DFA?

  The answer is simple, we may set:

  More precisely,

  $$Compile(a) = \langle S, \Sigma, s, \delta, F, \rangle$$

where

$$\begin{array}{rcl} S & = & \{1,2\} \\ \Sigma & = & \{a\} \\ s & = & 1 \\ \delta & = & \{(1,a) \mapsto 2\} \\ F & = & \{2\} \end{array}$$

- Case of (EF), where E and F are some regular expressions.

  If the regular expression we want to translate is of the form (EF) for some regular expressions E and F, what would be the corresponding DFA?

  Reasoning as sketched above, by composition of elementary pieces, we may assume that we have already produced DFAs for E and F, say

  $$\begin{array}{l} \mathit{Compile}(E) = \langle S_E, \Sigma_E, s_E, \delta_E, F_E, \rangle \\ \mathit{Compile}(F) = \langle S_F, \Sigma_F, s_F, \delta_F, F_F, \rangle \end{array}$$

  The question now is how to produce a DFA for the expression EF, by assembling the DFAs generated for E and F. Intuitively the solution is simple, since a word of $L(EF)$ is a word of $L(E)$ concatenated with a word of $L(F)$, we may try to compose the DFAs in sequence, merging the final states of $\mathit{Compile}(E)$ with the initial state of $\mathit{Compile}(F)$. In this way, the resulting DFA would work as follows: it would start in the initial state of $\mathit{Compile}(E)$, and continue until a state in F_E is reached. It would then have accepted the first part of a given word. Then it would somehow skip to the initial state of $\mathit{Compile}(F)$ and continue. If a final state of $\mathit{Compile}(F)$ is reached, the second part of the given word would have been accepted.

  This idea is fine, but the difficulty is: how do we express the jump from the final state of $\mathit{Compile}(E)$ to the initial state of $\mathit{Compile}(F)$? This would require some kind of special transition in the DFA, a transition that would allow it to move from a state to another state without consuming any symbol.

  This need suggests the introduction of a special kind of Finite Automaton, where such moves are allowed. They are called Non-Deterministic Finite Automata (NFA), and we will introduce them in greater detail in the next item.

2.7 Non Deterministic Finite Automata

A Non-Deterministic Finite Automata is a finite automata that besides the simple transitions that we have discussed for DFAs also permits transitions from one state to other state without consuming any symbol from the input stream.

We call to such transitions ()-transitions or ϵ-transitions.

Moreover, we also allow several transitions from one state to other states labelled by the same label (either a symbol or (). This is from where the "non-determinism" comes from.

Let us look to an example. Consider the NFA in Figure.

We check that the word `a b` is accepted. We start off as

$$(|\texttt{a b}, s_1)$$

After one step, we get to

$$(\texttt{a}|\texttt{b}, s_2)$$

After one more step, we get to

$$(\texttt{a b}|, s_3)$$

This was pretty similar to the DFA case. Now, for something different, we check that the word `a c d` is accepted. We start off as

$$(|\texttt{a c d}, s_1)$$

After one step, we get to

$$(\texttt{a}|\texttt{c d}, s_2)$$

But now, we move to

$$(\texttt{a}|\texttt{c d}, s_4)$$

without skipping any symbol in the input. We do that because there is a transition from s_2 to s_4 labeled with the empty word (). We can now continue, and get to

$$(\texttt{a c}|\texttt{d}, s_5)$$

and, finally, reach a final state

$$(\texttt{a c d}|, s_3)$$

For another example, consider now the following NFA.

It accepts the language {`talk mary`, `talk joe`}.

Notice that a NFA somehow seems to need to "guess" what is the correct path it must choose, in order to accept a word.

But that should not worry us for now. We say that a NFA accepts a word just in case there is some sequence of transitions from the initial state to some final state. We may imagine that the NFA will explore all paths somehow, and will always find an acceptance path, if there is one.

We may think that this would be terribly inefficient.

But in fact, it is not! We will later see that any NFA can be converted to a DFA that recognizes the same language, and that, of course, is no longer non-deterministic! This means that we may use NFAs to specify languages, with much more freedom than with DFAs, but without loosing efficiency or practical use.

So, NFAs are not introduced to complicate things, but actually to make things easier! In particular, they help us quite a lot to define the translation from regular expressions to finite automata: as we have seen, being able to define ()-transitions is very useful, for example, to construct an automaton to recognize (EF) given an automata to recognize E and another automaton to recognize F.

We we will see, ()-transitions will also be very helpful in the cases of $(E + F)$ and E^*. Before continuing with that, we give the formal definition of NFA. Formally, a NFA A is a structure

$$A = \langle S, \Sigma, s, \Delta, F \rangle$$

where

(a) S is the finite set of states of A

(b) Σ is the finite set of actions (or symbols) of A

(c) s is state in S, the initial state of A

(d) Δ is the transition relation of A, that given a current state in S and an action/symbol in Σ or the empty word () indicates a next state in S to which the automata should transition.
So we have $\Delta \subseteq S \times (\Sigma \cup \{()\}) \times S$.

(e) F is a subset of S, the set F of final states of A.

The key difference between a DFA and a NFA is obviously in the way transitions are defined. While in the case of DFA, the transitions are given by a partial function $\delta \in S \times \Sigma \to S$, in the case of NFAs, we must provide a relation $\Delta \subseteq S \times (\Sigma \cup \{()\}) \times S$.

The relation Δ is a set of triples (s, u, s') where s and s' are states, and u is either a symbol in Σ or the empty word. These triples define the possible transitions in the NFA.

Notice that in a NFA there can be more than transition from a given state labeled by the same symbol or by the empty word. Look to the following formal definitions of the NFAs sketched above

- $$Sample = \langle S, \Sigma, s, \Delta, F \rangle$$

  where

  $$\begin{array}{rcl} S &=& \{1,2,3,4,5\} \\ \Sigma &=& \{a,b,c,d\} \\ s &=& 1 \\ \Delta &=& \{(1,a,2),(2,b,3),(2,(),4),(4,c,5),(5,d,3)\} \\ F &=& \{3\} \end{array}$$

- $$Talkie = \langle S, \Sigma, s, \Delta, F \rangle$$

  where

  $$\begin{array}{rcl} S &=& \{1,2,3,4,5,6,7\} \\ \Sigma &=& \{talk, mary, joe\} \\ s &=& 1 \\ \Delta &=& \{(1,(),2),(2,talk,3),(3,joe,4), \\ && (1,(),5),(5,talk,6),(6,mary,7)\} \\ F &=& \{4,7\} \end{array}$$

As we have done for DFAs, for any NFA A, it is easy to define the multi-step transition relation $step_A$ inductively as follows:

$$\begin{array}{l} s \in S_A \Rightarrow (s,(),s) \in step_A \\ (s',u,s'') \in step_A \wedge a \in \Sigma_A \wedge (s,a,s') \in \Delta_A \Rightarrow (s,au,s'') \in step_A \\ (s',u,s'') \in step_A \wedge (s,(),s') \in \Delta_A \Rightarrow (s,u,s'') \in step_A \end{array}$$

Given the concepts presented above we may now give a precise definition for the language $\mathcal{L}(A)$ accepted by an NFA $A = \langle S, \Sigma, s, \Delta, F \rangle$, intuitively described above by

$$\mathcal{L}(A) = \{w \in Words(\Sigma) \mid w \text{ is accepted by the NFA } A\}$$

Indeed, we may simply set

$$\mathcal{L}(A) = \{w \in \mathit{Words}(\Sigma) \mid \exists f. f \in F_A \wedge (s_A, w, f) \in \mathit{step}_A\}$$

Reading off this precise definition, $\mathcal{L}(A)$ is the set of words that can induce a sequence of computation steps from the initial state s_A to some final state $f \in F_A$, as conveniently expressed by the step_A relation.

We see that the definition of $\mathcal{L}(A)$ for a NFA A is superficially identical to the definition of language accepted by a DFA. The additional expressiveness comes from the fact that step_A is now non-deterministic, that is, there may be several states s' such that $\mathit{step}_A(s, w, s')$, for some fixed s and w. While for a DFA, there is only at most one state s' such that $\mathit{step}_A(s, w, s')$, for some fixed s and w.

2.8 Compiling any Regular Expression to a DFA

We now continue our presentation of the translation between a regular expression and a DFA. But, we will now take a very useful intermediate step. Rather than translating into a DFA, we will translate into a NFA. This will make our life easier, and without any inconvenience. We will show later how to convert a NFA into an equivalent DFA. This whole method will then provide us a way of translating (or compiling) a regular expression into a DFA, by going through a NFA as an intermediate step. Sometimes in good science and engineering, we need to back up a bit, so to better jump ahead !

We start by summarizing what we have already found, but recasting to the setting of NFAs.

- Case of ().
  If the regular expression we want to translate is (), what would be the corresponding NFA?
  The answer is simple, we may set:
  More precisely,

  $$\mathit{Compile}(\ ()\ ) = \langle S, \Sigma, s, \delta, F, \rangle$$

  where

  $$\begin{array}{rcl} S & = & \{1\} \\ \Sigma & = & \emptyset \\ s & = & 1 \\ \Delta & = & \emptyset \\ F & = & \{1\} \end{array}$$

- Case of a, with $a \in \Sigma$

  If the regular expression we want to translate is a for some symbol a, what would be the corresponding NFA?

  The answer is simple, we may set:

  More precisely,

$$Compile(a) = \langle S, \Sigma, s, \Delta, F, \rangle$$

  where

$$\begin{array}{lcl} S & = & \{1, 2\} \\ \Sigma & = & \{a\} \\ s & = & 1 \\ \Delta & = & \{(1, a, 2)\} \\ F & = & \{2\} \end{array}$$

- Case of (EF), where E and F are any regular expressions.

  We continue our reasoning, and assume that we have already produced NFAs for E and F, as follows

$$\begin{array}{l} Compile(E) = \langle S_E, \Sigma_E, s_E, \Delta_E, F_E, \rangle \\ Compile(F) = \langle S_F, \Sigma_F, s_F, \Delta_F, F_F, \rangle \end{array}$$

  We assume that the sets S_E and S_F are disjoint. We can always assume that, even if we need to change the name of some states.

  Now, to define a NFA that accepts (EF) we let

$$Compile(EF) = \left\langle S_{(EF)}, \Sigma_{(EF)}, s_{(EF)}, \Delta_{(EF)}, F_{(EF)}, \right\rangle$$

  where

$$\begin{array}{lcl} S_{(EF)} & = & S_E \cup S_F \\ \Sigma_{(EF)} & = & \Sigma_E \cup \Sigma_F \\ s_{(EF)} & = & s_E \\ \Delta_{(EF)} & = & \Delta_E \cup \Delta_F \cup \{(s, (), s_F) \mid s \in F_E\} \\ F_{(EF)} & = & F_F \end{array}$$

- Case of $(E + F)$, where E and F are any regular expressions.

  Again, assume that we have already produced NFAs for the regular expressions E and F, as follows

$$\begin{array}{l} Compile(E) = \langle S_E, \Sigma_E, s_E, \Delta_E, F_E, \rangle \\ Compile(F) = \langle S_E, \Sigma_E, s_E, \Delta_E, F_E, \rangle \end{array}$$

Again, we assume that the sets S_E and S_F are disjoint.

Now, to define a NFA that accepts $(E+F)$ we let

$$Compile(E+F) = \left\langle S_{(E+F)}, \Sigma_{(E+F)}, s_{(E+F)}, \Delta_{(E+F)}, F_{(E+F)}, \right\rangle$$

where $s_{(E+F)}$ is a new state, not in $S_E \cup S_F$, and

$$\begin{array}{lcl} S_{(E+F)} & = & \{s_{(E+F)}\} \cup S_E \cup S_F \\ \Sigma_{(E+F)} & = & \Sigma_E \cup \Sigma_F \\ \Delta_{(E+F)} & = & \Delta_E \cup \Delta_F \cup \{(s_{(E+F)}, (), s_E), (s_{(E+F)}, (), s_F)\} \\ F_{(E+F)} & = & F_E \cup F_F \end{array}$$

- Case of $(E)^*$, where E is any regular expression.

  Again, assume that we have already produced a NFA for the regular expression E as follows

  $$Compile(E) = \langle S_E, \Sigma_E, s_E, \Delta_E, F_E, \rangle$$

  Now, to define a NFA that accepts (E^*) we let

  $$Compile(E^*) = \left\langle S_{(E^*)}, \Sigma_{(E^*)}, s_{(E^*)}, \Delta_{(E^*)}, F_{(E^*)}, \right\rangle$$

  where $s_{(E^*)}$ is a new state, not in S_E, and

  $$\begin{array}{lcl} S_{(E^*)} & = & \{s^*\} \cup S_E \cup S_F \\ \Sigma_{(E^*)} & = & \Sigma_E \\ \Delta_{(E^*)} & = & \Delta_E \cup \{(s_{(E^*)}, (), s_E)\} \cup \{(f, (), s_{(E^*)}) \mid f \in F_E\} \\ F_{(E^*)} & = & \{s_{(E^*)}\} \end{array}$$

The function $Compile(-)$ we just defined above translates (compiles) any regular expression E into a NFA that recognizes the language $L(E)$.

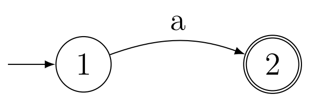
Lets see how it works in a concrete example. Let

$$E \triangleq (a+b)(ab+cd)^*$$

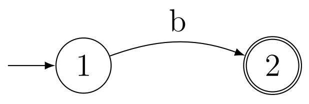
be a regular expression over the alphabet $\Sigma = \{a, b, c, d\}$. We illustrate the various automata in graphical form, by applying the function $Compile(-)$ presented above to the expression E.

It is clear that in order to compute $Compile(-)$ for a complex expression we need to compute it for subexpressions. For example, to compute $Compile(E)$ we need to compute $Compile(a+b)$ and $Compile((ab+cd)^*)$, and so on. So we start from most elementary subexpressions and compose the result, bottom up.

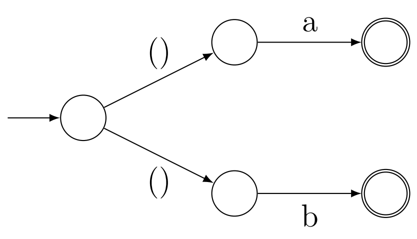
We first compute the NFA for a. It gives



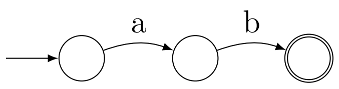
Then the NFA for b



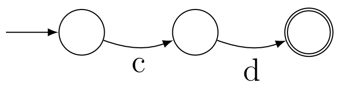
Then the NFA for $a + b$



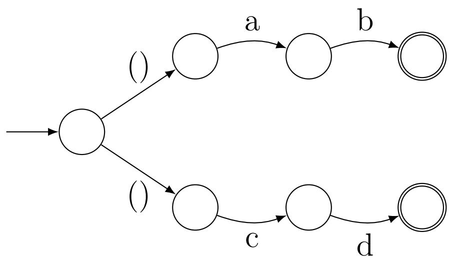
Then the NFA for ab



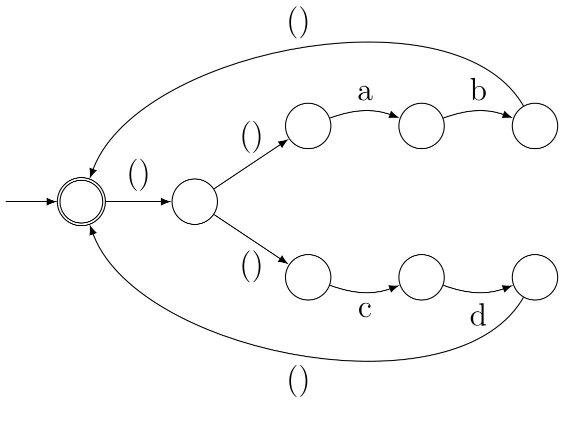
Then the NFA for cd



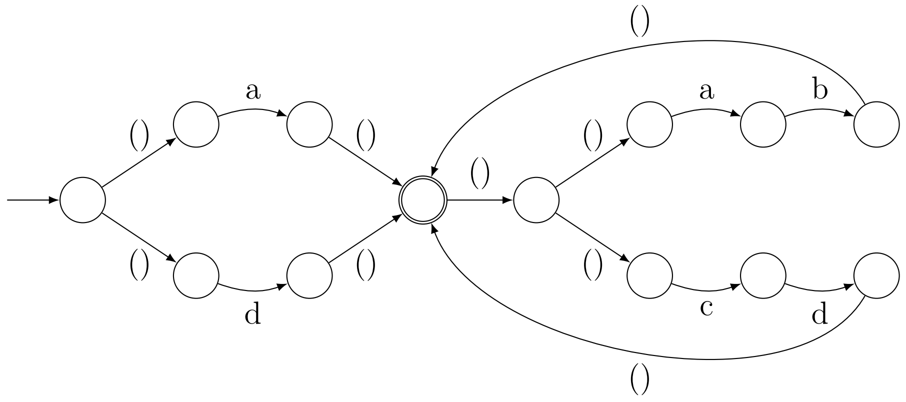
Then the NFA for $ab + cd$



Now the NFA for $(ab + cd)^*$



Finally, we get the whole NFA for the regular expression $(a+b)(ab+cd)^*$



2.9 Compiling a NFA to an equivalent DFA

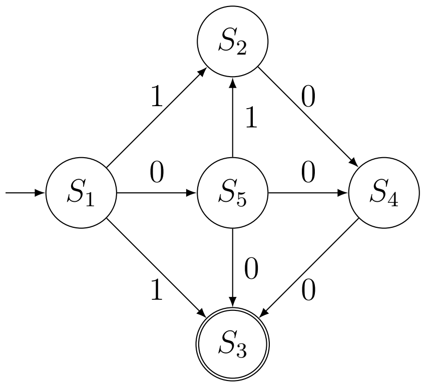
We promised in a previous section to show how given any non-deterministic finite automaton one can construct a deterministic finite automaton that recognizes exactly the same language.

At first sight, this may be really surprising! After all, NFAs seem to have in general much more degrees of freedom to accept a word, by magically guessing the right path to follow.

But actually, it is quite intuitive for a informatician / computer scientist to see why any NFA can be simulated by some DFA. The idea is to let the DFA simulate all alternative computations simultaneously, as if they were done in parallel (think, e.g., of a multiprocessor).

To grasp the general idea, it is easier to look to a concrete example. Consider then the following NFA over the alphabet $\Sigma = \{0, 1\}$, depicted in the figure below.

$$\begin{tikzcd}
 & S_2 \arrow[dr, "0"] & \\
\arrow[r] S_1 \arrow[ur, "1"] \arrow[r, "0"] \arrow[dr, "1"'] & S_5 \arrow[u, "1"'] \arrow[r, "0"] \arrow[d, "0"] & S_4 \arrow[dl, "0"] \\
 & S_3 &
\end{tikzcd}$$



Now, let us think on how the NFA BOB works to accept some word w.

Now assume that BOB is in its initial state, and starts a computation with the word $w = 100$.

Thus, the first symbol of w is 1. To what states may BOB transit ?

Looking to the possible transitions we see that BOB may transit to either s_2 or to s_3, non-deterministically, in one step. In fact, it may also transit to s_4, since it can silently move from s_3 to s_4 without consuming any other input symbol, due to the () labeled transition. Now, rather than picking a determined choice for BOB's step, let us just record the simple fact that after having consumed the symbol 1 the NFA BOB must have reached some state in the set $\{s_2, s_3, s_4\}$ and continue from then on.

The second symbol to process is 0. To what states may BOB transit, given that it must be in some of the states $\{s_2, s_3, s_4\}$?

Well, we see that if the given current possible state is s_2, there will be no possible next move, since BOB does not get out of s_2 by consuming 0. And likewise for s_3, since there is also no possible 0 move out of s_3.

On the other hand, if the given current possible state is s_4, we see that BOB may move to s_3 by consuming 0.

We conclude that from the information that BOB is in one of the states in $\{s_2, s_3, s_4\}$, after consuming 0, the only possible next state is s_3. In other words, from the set of states $\{s_2, s_3, s_4\}$, after consuming 0 , the only possible set of next states in $\{s_3\}$.

Now, the next symbol of w is 0. Since there is no possible move out of s_3 consuming 0, we conclude that $w = 110$ is not accepted by BOB.

Notice that we have explored all possible non-deterministic moves of BOB when processing the word 100, by keeping track of all accessible states in "parallel", as follows:

First, we have the configuration

$$|100, \{s_1\}$$

After consuming 1, BOB may get to any of the states

$$1|00, \{s_2, s_3, s_4\}$$

After consuming 0, BOB may get to any of the states

$$10|0, \{s_3\}$$

Now, BOB gets stuck since there is no 0 move out of s_3.

If you think about the way we have reasoned about BOB's possible moves, you may have noticed that we have reduced various non-deterministic computations to a single deterministic computation, expressed in terms of sets of accessible states, rather than in terms of isolated accessible states.

By thinking about the **sets of accessible states** rather than in terms of simple accessible states is the key insight for constructing a DFA equivalent to a given NFA. Each state of the constructed DFA will somehow represent sets of states of the given NFA.

We illustrate the technique with BOB, and construct an equivalent DFA, to which we call say dBOB.

First, the initial state of dBOB. This set must represent the set of BOB states that are accessible from the initial state of BIB without consuming any input symbol.

We may then set $R_1 = \{s_1\}$. In this case, the state R_1 only contains the initial state of BOB, since BOB does not contain any ()-labeled transitions from s_1.

Now, lets find the right transitions from R_1. We have already seen that consuming 1 we will get us to any of the states $\{s_2, s_3, s_4\}$. We may then set $R_2 = \{s_2, s_3, s_4\}$ and $\delta_{dBOB}(R_1, 1) = R_2$.

Now, exploring a possible 0-move from $R1$, will lead us to $R_3 = \{s_5\}$: there is no other possibility. We can then set $\delta_{dBOB}(R_1, 0) = R_3$.

We now continue exploring the possible transitions, from R_2 and R_3.

From R_2 by a 0-move we go to $R_4 = \{s_3\}$, as seen above. So $\delta_{dBOB}(R_2, 0) = R_3$

We now analyse the possible transitions from $R_2 = \{s_2, s_3, s_4\}$ by a 1-move. We see that there are no 1-labeled moves from any one of s_3, s_3 and s_4. So the set of 1 accessible states from $\{s_2, s_3, s_4\}$ is the empty set $\emptyset$. We may then leave $\delta_{dBOB}(R_2, 1)$ undefined, as there is no valid transition in this case.

Let us now consider possible transitions from $R_3 = \{s_5\}$.

For a 0-labeled move, we see that we get to $R_5 = \{s_3, s_4\}$, and we set $\delta_{dBOB}(R_3, 0) = R_5$.

For a 1-labeled move, we see that we get to $R_6 = \{s_2, s_4\}$, and we set $\delta_{dBOB}(R_3, 1) = R_6$.

We continue exploring the possible transitions, from R_5 and R_6.

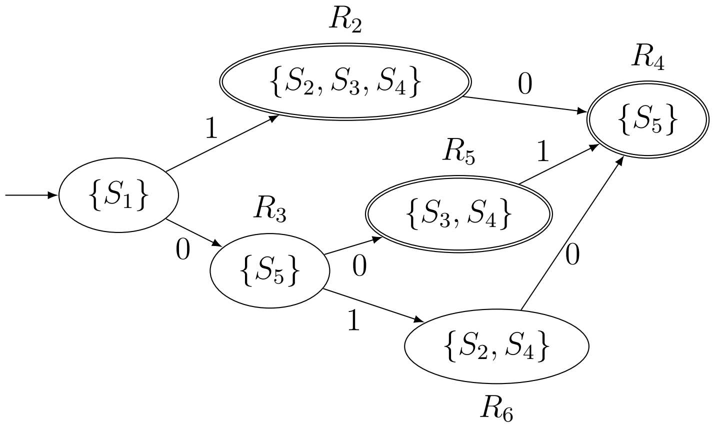
From $R_5 = \{s_3, s_4\}$, by a 0-labeled move we get to $\{s_3\} = R_4$, so $\delta_{dBOB}(R_5, 0) = R_4$.

From $R_5 = \{s_3, s_4\}$, there are no possible 1-labeled moves.

From $R_6 = \{s_2, s_4\}$, by a 0-labeled move we get to $\{s_3\} = R_4$, so $\delta_{dBOB}(R_6, 0) = R_4$.

From $R_6 = \{s_2, s_4\}$, again there is no possible 1-labeled move.

We have explored all possible transitions from reachable states, so we get to the following DFA, to which we have called dBOB.



Notice that the final states of dBOB are obviously the states that contain some final state of BOB, because when any one of such states are reached, that means that BOB could have reached a final state.

In the discussion above, we have illustrated the construction process of a DFA equivalent to a given NFA, in the sense that it recognizes the same language. This process is called the “Rabin-Scott powerset construction” (for finite automata), due to their developers, the famous computer scientists Michael Rabin and Dana Scott, who have introduced the main results on finite automata theory in a paper published in 1959.

We summarize now the construction in detail:

Assume given a NFA $A = \langle S_A, \Sigma_A, s_A, \Delta_A, F_A \rangle$.

Define a DFA $D = \langle S_A, \Sigma_D, s_D, \delta_D, F_D \rangle$ as follows:

- $S_D = \wp(S_A)$. So, the set of states of D is the set of all subsets of states of A (hence the name “powerset construction”). In practice we do not need to consider all the elements of $\wp(S_A)$, but only the ones reachable from the initial state, as we have illustrated in the example above.

- $\Sigma_D = \Sigma_A$.
- $s_D = \ \mathit{closeempty}(\{s_A\})$.

  Here we have introduce the notion of empty-closure of a set of states P, noted by $\mathit{closeempty}(P)$. Given a set of states $P \subseteq S_A$, we define its empty-closure as the set of all states that can be reached from some state in P just by following ()-moves.

  $$\mathit{closeempty}(P) = \{s \in S_A \mid \exists p.p \in P \wedge \mathit{step}_A(p, (), s)\}$$

  Recall that step_A is the multi-step transition relation associated to the NFA A, as defined in the previous section on NFAs.

  So the initial state s_D represents the set of all states of A which are accessible from the initial state s_A of A without consuming any symbol.

- $\delta_D = \{(s, a) \mapsto s' \in S_D \times \Sigma_D \times S_D \mid s' = \mathit{closeempty}(\mathit{move}(\{s\}, a))\}$ where $\mathit{move} \in S_D \times \Sigma_D \to \Sigma_D$ is the function defined by

  $$\mathit{move}(P, a) = \{s \in S_A \mid \exists p.p \in P \wedge (p, a, s) \in \Delta_A\}$$

  Intuitively, $\mathit{move}(P, a)$ gives the set of all states of A which are accessible from some state in P by following some a-move.

  So, $\delta_D(P, a)$ gives the set of states accessible from some state in P by following a-moves, and then closing under ()-moves. For example, as we have seen in the example BOB above

  $$\delta(\{s_1\}, 1) = \{s_2, s_3, s_4\}$$

  because

  $$\mathit{move}(\{s_1\}, 1) = \{s_2, s_3\}$$

  and

  $$\mathit{closeempty}(\{s_2, s_3\}) = \{s_2, s_3, s_4\}$$

- $F_D = \{s \in S_D \mid s \cap F \neq \emptyset\}$, the set of all states of D that contain at least one final state of A.

And that is it!

It can be proved that D recognizes exactly the same language as A recognizes. We will not do that now, although the proof is simple, we just have to show that A and D can simulate each other in every input word.

Notice that the powerset construction gives (in the worst case) an exponential blow up of the number of states when going from a non-deterministic a from deterministic automaton (as $\#S_D = 2^{\#S_A}$).

So there is this tension: one the one hand we can express a language very compactly with a NFA (which is essentially of the same size as the corresponding regular expression). But NFAs and regular expressions are not so easy to implement, while DFAs are! On the other hand DFAs are potentially much larger in terms of memory usage.

As usual, there is no free lunch !

2.10 Expressing a DFA by a Regular Expression

We have seen that every language specified by a regular expression can be implemented by a DFA. What about the other way around? Can every language implementable by a DFA be expressed by a regular expression? Or there are some finite automata that accept languages that escape the expressive power of regular expressions ?

In this section we will see that regular expressions are indeed powerful enough to specify all the languages recognized by DFAs. We thus find here a perfect match between specifications and implementations, a very convenient scenario, desirable for a good relationship between specifications and programming languages.

[to be concluded, please consult the course text book]

3 Turing Machines and Universality

[Missing discussion; to be done]

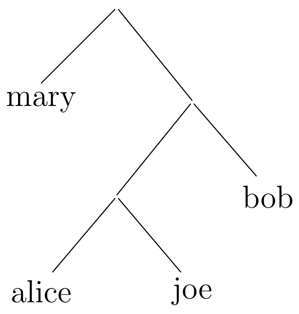
3.1 Structure of a Stack Based Turing Machine

3.1.1 Data

We first describe the data manipulated by a SBTM. Essentially data elements are binary trees, whose leaves are atomic data elements, that we may think of as symbols of a given alphabet Σ. We assume that Σ contains the special symbol `null` than stands for the empty tree. The set *DATA* is defined inductively as follows:

$$a \in \Sigma \Longrightarrow a \in \mathit{DATA}$$
$$t_1 \in \mathit{DATA}; t_2 \in \mathit{DATA} \Longrightarrow (t_1, t_2) \in \mathit{DATA}$$

A pair (t_1, t_2) represents a tree where t_1 is the left subtree and t_2 is the right subtree. As an example of a tree the one in the figure



which is represented as $(\texttt{mary}, ((\texttt{alice}, \texttt{joe}), \texttt{bob})) \in \mathit{DATA}$.

We also use trees to represent lists of data elements. For example, the list $[t_1, t_2, t_3]$ is represented by the three $(t_1, (t_2, (t_3, \texttt{null})))$. This is similar to the encoding of sequences as ordered pairs we have already seen before in the course.

3.1.2 Components of the Machine

The components of the machine are the following ones

- The Stack

  This is a simple, plain, regular stack of DATA elements, there is no special thing to say about it. The stack is manipulated by special machine instructions, in a way that will be made precise below.

  When a machine starts operation, the stack is assumed to be empty.

- The Memory

  The machine is equipped with a potentially infinite random access memory, where memory locations are names $M_1, M_2, M_3, \ldots$. Each memory cell can hold a data element (e.g. some value $v \in DATA$).

  Memory cells are read and written by special machine instructions, in a way that will be made precise below.

  When a machine starts operation, all memory cells are assumed to contain the value `null`. We represent such a cleared memory by M_{null}.

- The Program Counter

  The program counter is a special register that holds the control state in which the machine is in, at each moment during a computation. The machine works by following a state transition relation (or program), as we have previously seen with DFAs and stack machines, going through various states $q \in S$. When a machine starts operation, the program counter holds the initial control state s.

- The control transition function

  The transition function specifies the behavior of the machine. It is based on S, a finite set of states. As usual we assume a distinguished initial state $s \in S$ and a set $F \subseteq S$ of final states.

  The transition function has type $\tau \in S \times OP \to S$. Given a state s, a machine configuration, and an operation $op \in OP$, it specifies next state s', after the operation is performed on the configuration.

  Thus, τ is a set of triples $(q, OP, q') \in S \times OP \times S$.

  The possible operations are:

  - `push` t

    push the data value t on the top of the stack
  - `left`

    if the value on the top of the stack is a non-empty tree, replace such value by the tree's left branch.
  - `right`

    if the value on the top of the stack is a non-empty tree, replace such value by the tree's right branch.
  - `cons`

    pops two values from the top of the stack, first t_1 and then t_2, and finally pushes the tree (t_1, t_2).

- `eq`

  pops two values from the top of the stack, t_1 and t_2, and then pushes `true` if the values are equal, and `false` otherwise.

- $?a$

  executes if the value on the top of the stack is a, otherwise the operation is not applicable (and so the transition will not take place).

- ?`cons`

  executes if the value on the top of the stack is a non-empty tree, otherwise the operation is not applicable (and so the transition will not take place).

- `load` k

  pushes the content of memory cell M_k on the top of the stack.

- `store` k

  pops the value on the top of the stack and stores it on memory cell M_k.

A machine configuration is represented by a triple

$$(M, s, q)$$

where M is the memory, s is the stack, and q is the program counter.

We denote by $M[k]$ the contents of the memory cell M_k.

We also write $M[k/v]$ to represent the memory that coincides with M in all memory cells except in M_k, that contains v.

The possible transitions of the SBTM are then the following ones. We represent the stack as a list, with the topmost element as first element. We write

$$(M, s, q) \longrightarrow (M', s', q')$$

to say that the machine in a configuration with control state q, stack s and memory M transitions to a new configuration where the memory is now M', the stack becomes s, and the state in the PC is q'. We use

$q, q' \in S$ for denoting control states, $a, b \in \Sigma$, and $t, l, r, u \in DATA$.

$$\begin{array}{ll}
(M, s, q) \longrightarrow (M, (t, s), q') & \text{if } (q, \texttt{push } t, q') \in \tau \\
(M, ((l, r), s), q) \longrightarrow (M, (l, s), q') & \text{if } (q, \texttt{left}, q') \in \tau \\
(M, ((l, r), s), q) \longrightarrow (M, (r, s), q') & \text{if } (q, \texttt{right}, q') \in \tau \\
(M, (l, (r, s)), q) \longrightarrow (M, ((l, r), s), q') & \text{if } (q, \texttt{cons}, q') \in \tau \\
(M, (t, (u, s)), q) \longrightarrow (M, (\texttt{true}, s), q') & \text{if } (q, \texttt{eq}, q') \in \tau \text{ and } t = u \\
(M, (t, (u, s)), q) \longrightarrow (M, (\texttt{false}, s), q') & \text{if } (q, \texttt{eq}, q') \in \tau \text{ and } t \neq u \\
(M, (a, s), q) \longrightarrow (M, (a, s), q') & \text{if } (q, ?a, q') \in \tau \\
(M, ((a, b), s), q) \longrightarrow (M, ((a, b), s), q') & \text{if } (q, ?\texttt{cons}, q') \in \tau \\
(M, s, q) \longrightarrow (M, (M[k], s), q') & \text{if } (q, \texttt{load } k, q') \in \tau \\
(M, (u, s), q) \longrightarrow (M[k/u], s, q') & \text{if } (q, \texttt{store } k, q') \in \tau
\end{array}$$

A computation of the SBTM is a finite sequence of transitions

$$(M_{initial}, \emptyset, s) \xrightarrow{*} (M_{final}, s, q)$$

such that $q \in F$ is a final control state. $M_{initial}$ is the initial state of the memory. M_{final} is the final state of the memory. Initially, the stack is empty ($\emptyset$) but the end it may contain some information. That depends on the “programmer” intentions.

3.2 Example of a SBTM program

As a first example of a SBTM program, we consider a procedure to reverse a list of symbols.

The idea is to insert the input to the program in memory position X_2 and get the output on X_1. If we give as input to the program the list $[X, Y]$, we expect it to compute the list $[Y, X]$, if we give it as input the list $[l, u, c, y]$, we expect it to compute the list $[y, c, u, l]$, etc.

In this case, the code will leave X_2 set to `null` at the end, but of course it would be easy to change the code to avoid that.

We list the program by defining the transition relation τ, in a way that it looks similar to an assembly language program. Each line is of the form

$$\textit{state\_label}, \textit{instruction}, \textit{next\_state}$$

The program now follows (you may also find an almost identical one in the green board above):

start	`load 2`	*ch*
ch	?`null`	*end*
ch	?`cons`	*s1*
s1	`store 2`	*s2*
s2	`load 1`	*s3*
s3	`load 2`	*s4*
s4	`left`	*s5*
s5	`cons`	*s6*
s6	`store 1`	*s7*
s7	`load 2`	*s8*
s8	`right`	
s9	`store 2`	*start*

The initial state is *start*, and the only final state is *end*.

The program works by assuming that X_1 is initially `null`. At each step of the cycle starting at control state *start*, the machine removes the first element in the list stored in X_2 and adds it at the head of the list stored at X_1, when the list stored in X_2 becomes empty (contains `null`), the procedure stops.

If we want, we may list the program in more succinct form, by eliding the names of the states that just follow on a consecutive sequence. This way, the program looks pretty similar to an assembly language program for a modern processor.

```
start  load 2   ch
ch     ?null    end
ch     ?cons    s1
s1     store 2
       load 1
       load 2
       left
       cons
       store 1
       load 2
       right
       store 2  start
```

We illustrate how the programs works on a simple example, but precisely listing the transitions between configurations using the transition rules defined above. We give as input to the program the list $[X, Y]$ and expect it to compute $[Y, X]$.

The program just uses two memory cells M_1 and M_2, so we will represent the memory state as $\langle v_1 \rangle \langle v_2 \rangle$ where v_1 is the content of memory cell M_1 and v_2 is the content of memory cell M_2.

We consider then the initial memory $\langle \texttt{null} \rangle \langle [X, Y] \rangle$, meaning the $M[1]$ contains *null* ($M[1] = \mathit{null}$) and $M[2]$ contains the list $[X, Y]$ ($M[2] = [X, Y]$).

Remember that a list such as $[X, Y]$ actually is a tree $(X, (Y, \texttt{null}))$, lists are just a special case of trees, for which we have this special notation.

We thus consider the initial configuration for our SBTM

$$(\langle \texttt{null} \rangle \langle [X, Y] \rangle, \texttt{null}, \mathit{start})$$

We start off with the empty stack (`null`) and in the initial control state (*start*), and now, start running the program, applying an appropriate transition at each step. To make it readable, we annotate each transition with the operation executed (according to the transition relation τ - the program). Here we go.

$$
\begin{array}{l}
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,\texttt{null},\mathit{start})\overset{\texttt{load 2}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[[X,Y]],\mathit{ch})\overset{\texttt{?cons}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[[X,Y]],\mathit{s1})\overset{\texttt{store 2}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,\texttt{null},\mathit{s2})\overset{\texttt{load 1}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[\texttt{null}],\mathit{s3})\overset{\texttt{load 2}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[[X,Y],\texttt{null}],\mathit{s4})\overset{\texttt{left}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[X,\texttt{null}],\mathit{s5})\overset{\texttt{cons}}{\to}\\
(\langle\texttt{null}\rangle\,\langle[X,Y]\rangle\,,[[X]],\mathit{s6})\overset{\texttt{store 1}}{\to}\\
(\langle[X]\rangle\,\langle[X,Y]\rangle\,,\texttt{null},\mathit{s7})\overset{\texttt{load 2}}{\to}\\
(\langle[X]\rangle\,\langle[X,Y]\rangle\,,[[X,Y]],\mathit{s8})\overset{\texttt{right}}{\to}\\
(\langle[X]\rangle\,\langle[X,Y]\rangle\,,[[Y]],\mathit{s8})\overset{\texttt{store 2}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,\texttt{null},\mathit{start})\overset{\texttt{load 2}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[[Y]],\mathit{ch})\overset{\texttt{?cons}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[[Y]],\mathit{s1})\overset{\texttt{store 2}}{\to}\\
(\langle]\rangle\,\langle[Y]\rangle\,,\texttt{null},\mathit{s2})\overset{\texttt{load 1}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[[X]],\mathit{s3})\overset{\texttt{load 2}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[[Y],[X]],\mathit{s4})\overset{\texttt{left}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[Y,[X]],\mathit{s5})\overset{\texttt{cons}}{\to}\\
(\langle[X]\rangle\,\langle[Y]\rangle\,,[[Y,X]],\mathit{s6})\overset{\texttt{store 1}}{\to}\\
(\langle[Y,X]\rangle\,\langle[Y]\rangle\,,\texttt{null},\mathit{s7})\overset{\texttt{load 2}}{\to}\\
(\langle[Y,X]\rangle\,\langle[Y]\rangle\,,[[Y]],\mathit{s8})\overset{\texttt{right}}{\to}\\
(\langle[Y,X]\rangle\,\langle[Y]\rangle\,,[\texttt{null}],\mathit{s9})\overset{\texttt{store 2}}{\to}\\
(\langle[Y,X]\rangle\,\langle\texttt{null}\rangle\,,\texttt{null},\mathit{start})\overset{\texttt{load 2}}{\to}\\
(\langle[Y,X]\rangle\,\langle\texttt{null}\rangle\,,[\texttt{null}],\mathit{start})\overset{\texttt{?null}}{\to}\\
(\langle[Y,X]\rangle\,\langle\texttt{null}\rangle\,,[\texttt{null}],\mathit{end})
\end{array}
$$

The computation have thus reached a final configuration, since we assumed *end* to be a final control state. The result of reversing the list initially given at M_2 is not computed and delivered at M_1. Please read each step in the computation above carefully, and make sure you understand what is going on. It should be clear that the program works for lists of any length.

Before concluding the section we summarize the definition of what is a SBTM. Formally, a Stack Based Turing Machine T is a structure

$$T = \langle S, \Sigma, s, M, \tau, F \rangle$$

where

1. S is the finite set of control states of T
2. Σ is the finite set of actions (or symbols) of T.

   Based on Σ we define the set $DATA$ of values manipulable by the machine, as shown before.
3. s is distinguished state in S, the initial state of T
4. M is the initial memory configuration.
5. τ is the transition relation of T, that given a current state in S and a machine operation indicates the next state in S to which the machine should transition after performing the operation (if the operation is possible). So we have $\tau \in S \times OP \times S$.

   The relation τ is deterministic in the sense that τ must satisfy the following condition:

   $(s, op_1, s') \in \tau$ and $(s, op_2, s'') \in \tau$ then $op_1 = ?\alpha$ and $op_2 = ?\beta$ and $\alpha \neq \beta$.

   This means that "non-deterministic transitions" must be labeled (different) by top of stack query operations, so that the actual value on the top of stack will be used to pick the unique appropriate alternative.
6. F is a subset of S, the set F of final states of T.
7. The set of operations OP is given by

   $$OP = \{\texttt{push}\ t, \texttt{left}, \texttt{right}, \texttt{cons}, \texttt{eq}, ?t, ?\texttt{cons}, \texttt{load}\ k, \texttt{store}\ k\}$$

   where t is any value $t \in \texttt{DATA}$, and k any natural number $k \in NAT$.

3.3 Universality

3.3.1 Programs as Data

We may represent the transition relation of any SBTM as data, as follows. A natural number k is represented by a list of length k built only with the symbol `1`. Each element of the transition relation is represented by a four element list:

$$TRANS\_STEP = [state_1, opcode, arg, state_2]$$

For example, the triple

$$(s3, \texttt{load}\ 2, s4)$$

will be encoded as

$$[s3, load, [\texttt{1}, \texttt{1}], s4]$$

We will need to consider an alphabet Σ containing, besides all the symbols of the given SBTM, special symbols to represent all the control states, all the operation codes, the symbol `1`, etc. A whole program will then be represented by the list of all triples, in some arbitrary order.

$$Program = [trans\_step_1, trans\_step_2, trans\_step_2, trans\_step_3, \ldots]$$

The memory state is also represented as a list of values, where the k^{th} element represents the memory location M_k. For every program to be simulated, the number of used memory locations is always finite, so we may initially construct the memory list with the length we wish.

3.3.2 Universal Program

The program for the universal SBTB will make special use of the following reserved memory locations

- $M_{\texttt{PC}}$, containing the value of the program counter (a control state, represented by the corresponding symbol)
- $M_{\texttt{STACK}}$, containing the data representation of the stack.
- $M_{\texttt{CODE}}$, containing the data representation of the program to be simulated.
- $M_{\texttt{CODEL}}$, containing the search list for the next instruction.
- $M_{\texttt{TRIPLE}}$, containing the current triple under execution.

- $M_{\texttt{OP}}$, containing the current operation under execution.
- $M_{\texttt{MEM}}$, containing the current simulated memory.
- $M_{\texttt{FINALS}}$, containing the list of final states.
- Other temporary locations, local to subsidiary operations and not so important as the main locations above are discussed below.

The initial state of the the universal program is *pre_fetch*, the sole final state is *halt*. The machine is initialized with the initial state of the program to be simulated in the M_{PC} register. The following code implements the "fetch" part of each machine cycle. It inspects the current state, and checks if it belongs to the set of final states. If that is the case, the simulator halts, otherwise proceeds to decode the instruction and jump to the appropriate implementation.

```
pre_fetch   load FINALS

other       store TMP_FINALS   chk1
            load TMP_FINALS    chk1
chk1        ?null
sel         store D            fetch
chk1        ?cons
            left
            load PC
            eq                 at_end?
at_end?     ?true              halt
at_end?     ?false
            store D
            load TMP_FINALS
            right              other

fetch       load CODE

retry       store CODEL
            load CODEL         sel
sel         ?null              ERROR
sel         ?cons              go
go          left
            left
            load PC
            eq                 cmp
cmp         ?false
            load CODEL
            right              retry
cmp         ?true
            load CODEL
            left
            store TRIPLE       dispatch
```

The next snippet is executed after each instruction simulator terminates, initializes M_{PC} with the next state as defined in the current triple, and loops again to the initial state *pre_fetch*.

```
next  load TRIPLE
      right
      right
      right
      left
      store PC          pre_fetch
```

This is the dispatch code, it picks the current triple, gets the operation code out of it, and branches to the appropriate operation implementation.

```
dispatch     load TRIPLE
             right
             left
             store OP
             load OP
             push push
             eq
s1           ?true          is_push_op
is_push_op   storeD         push_impl
s1           ?false
             store D
             load OP
             push right
             eq             s2
s2           ?true          is_right_op
is_right_op  store D        right_impl
s2           ?false
             store D
             load OP
             push left
             eq             s3
s3           ?true          is_left_op
is_left_op   store D        left_impl
s3           ?false
             store D
             load OP
             push cons
             eq             s4
s4           ?true          is_cons_op
is_cons_op   store D        cons_impl
s4           ?false
             store D
             load OP
             push load
             eq             s5
s5           ?true          is_load_op
is_load_op   store D        load_impl
s5           ?false
...                         identical for the remaining operation codes
```

We now present the implementation of the various instructions.

```
push_impl   load STACK
            load TRIPLE
            right
            right
            left
            cons
            store STACK   next

right_impl  load STACK
            right
            load STACK
            left
            right
            cons
            store STACK   next

left_impl   load STACK
            right
            load STACK
            left
            left
            cons
            store STACK   next

cons_impl   load STACK
            right
            right
            load STACK
            right
            left
            load STACK
            left
            cons
            cons
            store STACK   next
```

```
eq_impl      load STACK
             right
             right
             load STACK
             right
             left
             load STACK
             left
             eq
             cons
             store STACK   next

a?_impl      load STACK
             left
             load OP
             right
             right
             left
             eq            tst
tst          ?true
             store D       next
tst          ?false
             store D       retry

cons?_impl   load STACK
             left          s1
s1           ?cons
             store D       next
             push null
s1           ?null
             store D       retry
```

```
load_impl   load OP
            right
            right
            left
            store MEM_POS
            load STACK
            load MEM
next_pos    store TMP_MEM
            load MEM_POS     chm
chm         ?null
            store D
            load TMP_MEM
            left
            cons
            store STACK      NEXT
chm         ?cons
            right
            store MEM_POS
            load MEM_TMP
            right            next_pos


store_impl  load OP
...                          left as exercise :-)
```