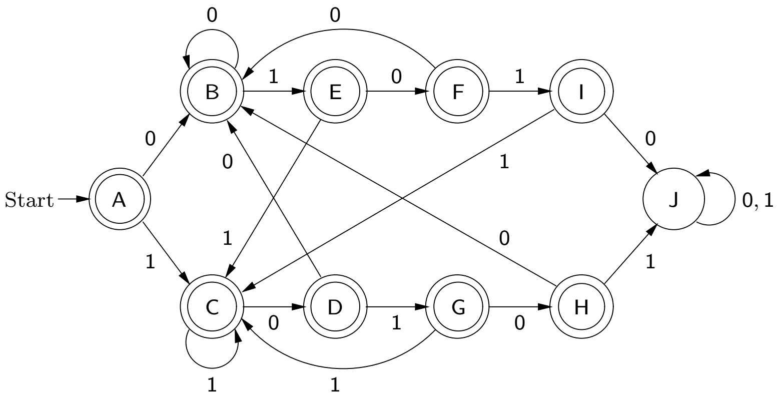
Formal Language Theory

Integrating Experimentation and Proof

Alley Stoughton

Draft of Fall 2018



Contents

List of Figures

Preface

Background

Since the 1930s, the subject of formal language theory, also known as automata theory, has been developed by computer scientists, linguists and mathematicians. Formal languages (or simply languages) are sets of strings over finite sets of symbols, called alphabets, and various ways of describing such languages have been developed and studied, including regular expressions (which "generate" languages), finite automata (which "accept" languages), grammars (which "generate" languages) and Turing machines (which "accept" languages). For example, the set of identifiers of a given programming language is a formal language—one that can be described by a regular expression or a finite automaton. And, the set of all strings of tokens that are generated by a programming language's grammar is another example of a formal language.

Because of its applications to computer science, most computer science programs offer both undergraduate and graduate courses in this subject. Perhaps the best known applications are to compiler construction. For example, regular expressions and finite automata are used when specifying and implementing lexical analyzers, and grammars are used to specify and implement parsers. Finite automata are used when designing hardware and network protocols. And Turing machines—or other machines/programs of equivalent power—are used to formalize the notion of algorithm, which in turn makes possible the study of what is, and is not, computable.

Formal language theory is largely concerned with algorithms, both ones that are explicitly presented, and ones implicit in theorems that are proved constructively. In typical courses on formal language theory, students apply these algorithms to toy examples by hand, and learn how they are used in applications. Although much can be achieved by a paper-and-pencil approach to the subject, students would obtain a deeper understanding of the subject if they could experiment with the algorithms of formal language theory using computer tools.

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize a deterministic finite automaton that accepts some language, L. With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (hopefully) prove it correct. But, given the right computer tools, another approach would be possible. First, the student could try to express L in terms of simpler languages, making use of various language operations (e.g., union, intersection, difference, concatenation, closure). The student could then synthesize automata accepting the simpler languages, enter these machines into the system, and then combine these machines using operations corresponding to the language operations used to express L. Finally, the resulting machine could be minimized. With some such exercises, a student could solve the exercise in both ways, and could compare the results. Other exercises of this type could only be solved with machine support.

Integrating Experimentation and Proof

To support experimentation with formal languages, I designed and implemented a computer toolset called Forlan [Sto05, Sto08]. Forlan is implemented in the functional programming language Standard ML (SML) [MTHM97, Pau96], a language whose notation and concepts are similar to those of mathematics. Forlan is a library on top of the Standard ML of New Jersey (SML/NJ) implementation of SML [AM91]. It's used interactively, and users are able to extend Forlan by defining SML functions.

In Forlan, the usual objects of formal language theory—finite automata, regular expressions, grammars, labeled paths, parse trees, etc.—are defined as abstract types, and have concrete syntax. Instead of Turing machines, Forlan implements a simple functional programming language of equivalent power, but which has the advantage of being much easier to program in than Turing machines. Programs are also abstract types, and have concrete syntax. Although mainly *not* graphical in nature, Forlan includes the Java program JForlan, a graphical editor for finite automata and regular expression, parse and program trees. It can be invoked directly, or via Forlan.

Numerous algorithms of formal language theory are implemented in Forlan, including conversions between regular expressions and different kinds of automata, the usual operations (e.g., union) on regular expressions, automata and grammars, equivalence testing and minimization of deterministic finite automata, a general parser for grammars, etc. Forlan provides support for regular expression simplification, although the algorithms used are works in progress. It also implements the functional programming language used as a substitute for Turing machines.

This undergraduate-level textbook and Forlan were designed and developed together. I have attempted to keep the conceptual and notational distance between the textbook and toolset as small as possible. The book treats most concepts and algorithms both theoretically, especially using proof, and through experimentation, using Forlan.

In contrast to some books on formal language theory, the book emphasizes the concrete over the abstract, providing numerous, fully worked-out examples of how regular expressions, finite automata, grammars and programs can be designed and proved correct. In my view, students are most able to learn how to write proofs—and to see the benefit of doing so—if their proofs are about things that they have designed.

I have also attempted to simplify the foundations of the subject, using alternative definitions when needed. E.g., finite automata are defined in such a way that they form a set (as opposed to a proper class), and so that more restricted forms of automata (e.g., deterministic finite automata (DFAs)) are proper subsets of that set. And finite automata are given semantics using labeled paths, from which the traditional δ notation is derived, in the case of DFAs. Furthermore, the book treats the set theoretic foundations of the subject more rigorously than is typical.

Readers of this book are assumed to have significant experience reading and writing informal mathematical proofs, of the kind one finds in mathematics books. This experience could have been gained, e.g., in courses on discrete mathematics, logic or set theory. The book assumes no previous knowledge of Standard ML. In order to understand and extend the implementation of Forlan, though, one must have a good working knowledge of Standard ML, as could be obtained by working through [Pau96] or [Ull98].

Drafts of this book were successfully used at Kansas State University in a semester long, undergraduate course on formal language theory.

Outline of the Book

The book consists of five chapters. Chapter 1, *Mathematical Background*, consists of the material on set theory, induction and recursion, and trees and inductive definitions that is required in the remaining chapters.

In Chapter 2, *Formal Languages*, we say what symbols, strings, alphabets and (formal) languages are, show how to use various induction principles to prove language equalities, and give an introduction to the Forlan toolset. The remaining three chapters introduce and study more restricted sets of languages.

In Chapter 3, *Regular Languages*, we study regular expressions and languages, five kinds of finite automata, algorithms for processing and converting between regular expressions and finite automata, properties of regular languages, and applications of regular expressions and finite automata to searching in text files, lexical analysis, and the design of finite state systems.

In Chapter **??**, *Context-free Languages*, we study context-free grammars and languages, algorithms for processing grammars and for converting regular expressions and finite automata to grammars, top-down (recursive descent) parsing, and properties of context-free languages. We prove that the set of context-free languages is a proper superset of the set of regular languages.

Finally, in Chapter **??**, *Recursive and Recursively Enumerable Languages*, we study the functional programming language that we use instead of Turing machines to define the recursive and recursively enumerable languages. We study algorithms for processing programs and for converting grammars to programs, and properties of recursive and recursively enumerable languages. We prove that the set of context-free languages is a proper subset of the set of recursive languages, that the set of recursive languages is a proper subset of the set of recursively enumerable languages, and that there are languages that are not recursively enumerable. Furthermore, we show that there are problems, like the halting problem (the problem of determining whether a program halts when run on a given input), that can't be solved by programs.

Further Reading and Related Work

This book covers most of the material that is typically presented in an undergraduate course on formal language theory. On the other hand, typical textbooks on formal language theory cover much more of the subject than we do. Readers who are interested in learning more, or who would like to be exposed to alternative presentations of some of the material in this book, should consult one of the many fine books on formal language theory, such as [HMU01, Koz97, LP98, Mar91, Lin01].

Neil Jones [Jon97] pioneered the use of a programming language with structured data as an alternative to Turing machines for studying the limits of what is computable. In Chapter **??**, we have followed Jones's approach in some ways. On the other hand, our programming language is functional, not imperative (assignment-oriented), and it has explicit support for the symbols and strings of formal language theory.

The existing formal languages toolsets fit into two categories. In the first category are tools, like JFLAP [BLP$^+$97, HR00, Rod06], Pâté [BLP$^+$97, HR00], the Java Computability Toolkit [RHND99], and Turing's World [BE93], that are graphically oriented and help students work out relatively small examples. The books [Rod06] (on JFLAP) and [Lin01] (an introduction to formal language theory) are intended to be used in conjunction with each other. The second category consists of toolsets that, like Forlan, are embedded in programming languages, and so support sophisticated experimentation with formal languages. Toolsets in this category include Automata [Sut92], Grail+ [RW94, RWYC17], HaLeX [Sar02], Leiß's Automata Library [Lei10] and Vaucanson [LRGS04]. I am not aware of other textbook/toolset packages whose toolsets are members of this second category.

Notes, Exercises and Website

In the "notes" subsections that conclude most sections of the book, I have restricted myself to describing how the book's approach differs from standard practice. Readers interested in the history of the subject can consult [HMU01, Koz97, LP98].

The book contains numerous fully worked-out examples, many of which consist of designing and proving the correctness of regular expressions, finite automata, grammars and programs. Similar exercises, as well as other kinds of exercises, are scattered throughout the book.

The Forlan website

`http://alleystoughton.us/forlan`

contains:

- instructions for downloading and installing the Forlan toolset, and JForlan;
- the Forlan manual;
- instructions for reporting errors or making suggestions; and
- the Forlan distribution, including the source for Forlan and JForlan, as well as the LaTeX source for this book.

Acknowledgments

Leonard Lee and Jessica Sherrill designed and implemented graphical editors for Forlan finite automata (JFA), and regular expression and parse trees (JTR), respectively. Their work was unified and enhanced (of particular note was the addition of support for program trees) by Srinivasa Aditya Uppu, resulting in an initial version of JForlan. Subsequently, Kenton Born carried out a major redevelopment of JForlan, resulting in JForlan Version 1.0. A further revision, by the author, led to JForlan Version 2.0.

It is a pleasure to acknowledge helpful discussions relating to the Forlan project with Eli Fox-Epstein, Brian Howard, Rod Howell, John Hughes, Nathan James, Patrik Jansson, Jace Kohlmeier, Dexter Kozen, Matthew Miller, Aarne Ranta, Ryan Stejskal, Colin Stirling, Lucio Torrico and Lyn Turbak. Much of this work was done while I was employed by Kansas State University, and some of the work was done while I was on sabbatical at the Department of Computing Science of Chalmers University of Technology.

Chapter 1

Mathematical Background

This chapter consists of the material on set theory, induction, trees, inductive definitions and recursion that is required in the remaining chapters.

1.1 Basic Set Theory

In this section, we will cover the material on sets, relations, functions and data structures that will be needed in what follows. Much of this material should be at least partly familiar.

1.1.1 Describing Sets by Listing Their Elements

We write $\emptyset$ for the *empty set*—the set with no elements. Finite sets can be described by listing their elements inside set braces: $\{x_1, \ldots, x_n\}$. E.g., $\{3\}$ is the *singleton set* whose only element is 3, and $\{1, 3, 5, 7\}$ is the set consisting of the first four odd numbers.

1.1.2 Sets of Numbers

We write:

- $\mathbb{N}$ for the set $\{0, 1, \ldots\}$ of all natural numbers;
- $\mathbb{Z}$ for the set $\{\ldots, -1, 0, 1, \ldots\}$ of all integers;
- $\mathbb{R}$ for the set of all real numbers.

Note that, for us, 0 *is* a natural number. This has many pleasant consequences, e.g., that the size of a finite set or the length of a list will always be a natural number.

1.1.3 Relationships between Sets

As usual, we write $x \in Y$ to mean that x is one of the elements (members) of the set Y. Sets A and B are equal ($A = B$) iff (if and only if) they have the same elements, i.e., for all x, $x \in A$ iff $x \in B$.

Suppose A and B are sets. We say that:

- A is a *subset* of B ($A \subseteq B$) iff, for all $x \in A$, $x \in B$;
- A is a *proper subset* of B ($A \subsetneq B$) iff $A \subseteq B$ but $A \neq B$.

In other words: A is a subset of B iff every everything in A is also in B, and A is a proper subset of B iff everything in A is in B, but there is at least one element of B that is not in A.

For example, $\emptyset \subsetneq \mathbb{N}$, $\mathbb{N} \subseteq \mathbb{N}$ and $\mathbb{N} \subsetneq \mathbb{Z}$. The definition of $\subseteq$ gives us the most common way of showing that $A \subseteq B$: we suppose that $x \in A$, and show (with no additional assumptions about x) that $x \in B$. Similarly, if we want to show that $A = B$, it will suffice to show that $A \subseteq B$ and $B \subseteq A$, i.e., that everything in A is in B, and everything in B is in A. Of course, we can also use the usual properties of equality to show set equalities. E.g., if $A = B$ and $B = C$, we have that $A = C$.

Note that, for all sets A, B and C:

- if $A \subseteq B \subseteq C$, then $A \subseteq C$;
- if $A \subseteq B \subsetneq C$, then $A \subsetneq C$;
- if $A \subsetneq B \subseteq C$, then $A \subsetneq C$;
- if $A \subsetneq B \subsetneq C$, then $A \subsetneq C$.

Given sets A and B, we say that:

- A is a *superset* of B ($A \supseteq B$) iff, for all $x \in B$, $x \in A$;
- A is a *proper superset* of B ($A \supsetneq B$) iff $A \supseteq B$ but $A \neq B$.

Of course, for all sets A and B, we have that: $A = B$ iff $A \supseteq B \supseteq A$; and $A \subseteq B$ iff $B \supseteq A$. Furthermore, for all sets A, B and C:

- if $A \supseteq B \supseteq C$, then $A \supseteq C$;
- if $A \supseteq B \supsetneq C$, then $A \supsetneq C$;
- if $A \supsetneq B \supseteq C$, then $A \supsetneq C$;
- if $A \supsetneq B \supsetneq C$, then $A \supsetneq C$.

1.1.4 Set Formation

We will make extensive use of the $\{\cdots \mid \cdots\}$ notation for forming sets. Let's consider two representative examples of its use.

For the first example, let

$$A = \{\, n \mid n \in \mathbb{N} \text{ and } n^2 \geq 20 \,\} = \{\, n \in \mathbb{N} \mid n^2 \geq 20 \,\}.$$

(where the third of these expressions abbreviates the second one). Here, n is a bound variable and is universally quantified—changing it uniformly to m, for instance, wouldn't change the meaning of A. By the definition of A, we have that, for all n,

$$n \in A \quad \text{iff} \quad n \in \mathbb{N} \text{ and } n^2 \geq 20.$$

Thus, e.g.,

$$5 \in A \quad \text{iff} \quad 5 \in \mathbb{N} \text{ and } 5^2 \geq 20.$$

Since $5 \in \mathbb{N}$ and $5^2 = 25 \geq 20$, it follows that $5 \in A$. On the other hand, $5.5 \not\in A$, since $5.5 \not\in \mathbb{N}$, and $4 \not\in A$, since $4^2 \not\geq 20$.

For the second example, let

$$B = \{\, n^3 + m^2 \mid n, m \in \mathbb{N} \text{ and } n, m \geq 1 \,\}.$$

Note that $n^3 + m^2$ is a term (expression), rather than a variable. The variables n and m are existentially quantified, rather than universally quantified, so that, for all l,

$$\begin{aligned} l \in B \quad &\text{iff} \quad l = n^3 + m^2, \text{ for some } n, m \text{ such that } n, m \in \mathbb{N} \text{ and } n, m \geq 1 \\ &\text{iff} \quad l = n^3 + m^2, \text{ for some } n, m \in \mathbb{N} \text{ such that } n, m \geq 1. \end{aligned}$$

Thus, to show that $9 \in B$, we would have to show that

$$9 = n^3 + m^2 \text{ and } n, m \in \mathbb{N} \text{ and } n, m \geq 1,$$

for some values of n, m. And, this holds, since $9 = 2^3 + 1^2$ and $2, 1 \in \mathbb{N}$ and $2, 1 \geq 1$.

We use set formation in the following definition. Given $n, m \in \mathbb{Z}$, we write $[n : m]$ for $\{\, l \in \mathbb{Z} \mid l \geq n \text{ and } l \leq m \,\}$. Thus $[n : m]$ is all of the integers that are at least n and no more than m. For example, $[-2 : 1]$ is $\{-2, -1, 0, 1\}$ and $[3 : 2]$ is $\emptyset$.

Some uses of the $\{\cdots \mid \cdots\}$ notation are too "big" to be sets, and instead are *proper classes*. A *class* is a collection of universe elements, and a class is *proper* iff it is not a set. E.g., $\{\, A \mid A \text{ is a set and } A \not\in A \,\}$ is a proper class: assuming that it is a set is inconsistent—makes it possible to prove anything. This is the so-called Russell's Paradox. To know that a set formation is valid, it suffices to find a set that includes all the elements in the class being defined. E.g., the above set formations are valid, because the classes being defined are subsets of $\mathbb{N}$.

1.1.5 Operations on Sets

Next, we consider some standard operations on sets. Recall the following operations on sets A and B:

$$\begin{aligned}
A \cup B &= \{\, x \mid x \in A \text{ or } x \in B \,\} && \text{(union)}\\
A \cap B &= \{\, x \mid x \in A \text{ and } x \in B \,\} && \text{(intersection)}\\
A - B &= \{\, x \in A \mid x \notin B \,\} && \text{(difference)}\\
A \times B &= \{\, (x, y) \mid x \in A \text{ and } y \in B \,\} && \text{(product)}\\
\mathcal{P}\, A &= \{\, X \mid X \subseteq A \,\} && \text{(power set).}
\end{aligned}$$

The axioms of set theory assert that all of these set formations are valid (intersection and difference are obviously valid).

Of course, union and intersection are both commutative and associative ($A \cup B = B \cup A$, $(A \cup B) \cup C = A \cup (B \cup C)$, $A \cap B = B \cap A$ and $(A \cap B) \cap C = A \cap (B \cap C)$, for all sets A, B, C). Furthermore, we have that union is idempotent ($A \cup A = A$, for all sets A), and that $\emptyset$ is the identity for union ($\emptyset \cup A = A = A \cup \emptyset$, for all sets A). Also, intersection is idempotent ($A \cap A = A$, for all sets A), and $\emptyset$ is the zero for intersection ($\emptyset \cap A = \emptyset = A \cap \emptyset$, for all sets A).

It is easy to see that, for all sets X and Y, $X \subseteq X \cup Y$ and $Y \subseteq X \cup Y$. We say that sets X and Y are *disjoint* iff $X \cap Y = \emptyset$, i.e., iff X and Y have nothing in common.

$A - B$ is formed by removing the elements of B from A, if necessary. For example, $\{0, 1, 2\} - \{1, 4\} = \{0, 2\}$. $A \times B$ consists of all ordered pairs (x, y), where x comes from A and y comes from B. For example, $\{0, 1\} \times \{1, 2\} = \{(0, 1), (0, 2), (1, 1), (1, 2)\}$. Remember that an ordered pair (x, y) is different from $\{x, y\}$, the set containing just x and y. In particular, we have that, for all x, x', y, y', $(x, y) = (x', y')$ iff $x = x'$ and $y = y'$, whereas $\{1, 2\} = \{2, 1\}$. If A and B have n and m elements, respectively, for $n, m \in \mathbb{N}$, then $A \times B$ will have nm elements. Finally, $\mathcal{P}\, A$ consists of all of the subsets of A. For example, $\mathcal{P}\, \{0, 1\} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. If A has n elements, for $n \in \mathbb{N}$, then $\mathcal{P}\, A$ will have 2^n elements.

We let $\times$ associate to the right, so that, e.g., $A \times B \times C = A \times (B \times C)$. And, we abbreviate $(x_1, (x_2, \cdots (x_{n-1}, x_n) \cdots))$ to $(x_1, x_2, \ldots, x_{n-1}, x_n)$, thinking of it as an *ordered n-tuple*. For example $(x, (y, z))$ is abbreviated to (x, y, z), and we think of it as an *ordered triple*.

As an example of a proof involving sets, let's prove the following simple proposition, which says that intersections may be distributed over unions:

Proposition 1.1.1
Suppose A, B and C are sets.

(1) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

(2) $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$.

Proof. We show (1), the proof of (2) being similar. It will suffice to show that $A\cap(B\cup C)\subseteq(A\cap B)\cup(A\cap C)\subseteq A\cap(B\cup C)$.

($A\cap(B\cup C)\subseteq(A\cap B)\cup(A\cap C)$) Suppose $x\in A\cap(B\cup C)$. We must show that $x\in(A\cap B)\cup(A\cap C)$. By our assumption, we have that $x\in A$ and $x\in B\cup C$. Since $x\in B\cup C$, there are two cases to consider.

- Suppose $x\in B$. Then $x\in A\cap B\subseteq(A\cap B)\cup(A\cap C)$, so that $x\in(A\cap B)\cup(A\cap C)$.
- Suppose $x\in C$. Then $x\in A\cap C\subseteq(A\cap B)\cup(A\cap C)$, so that $x\in(A\cap B)\cup(A\cap C)$.

($(A\cap B)\cup(A\cap C)\subseteq A\cap(B\cup C)$) Suppose $x\in(A\cap B)\cup(A\cap C)$. We must show that $x\in A\cap(B\cup C)$. There are two cases to consider.

- Suppose $x\in A\cap B$. Then $x\in A$ and $x\in B\subseteq B\cup C$, so that $x\in A\cap(B\cup C)$.
- Suppose $x\in A\cap C$. Then $x\in A$ and $x\in C\subseteq B\cup C$, so that $x\in A\cap(B\cup C)$.

□

Exercise 1.1.2
Suppose A, B and C are sets. Prove that union distributes over intersection, i.e., for all sets A, B and C:

(1) $A\cup(B\cap C)=(A\cup B)\cap(A\cup C)$.

(2) $(A\cap B)\cup C=(A\cup C)\cap(B\cup C)$.

Next, we consider generalized versions of union and intersection that work on sets of sets. If X is a set of sets, then the *generalized union* of X ($\bigcup X$) is

$$\{\,a\mid a\in A,\text{for some }A\in X\,\}.$$

Thus, to show that $a\in\bigcup X$, we must show that a is in at least one element A of X. For example

$$\bigcup\{\{0,1\},\{1,2\},\{2,3\}\}=\{0,1,2,3\}=\{0,1\}\cup\{1,2\}\cup\{2,3\},$$
$$\bigcup\emptyset=\emptyset.$$

(Again, we rely on set theory's axioms to know this set formation is valid.)

If X is a *nonempty* set of sets, then the *generalized intersection* of X ($\bigcap X$) is

$$\{\,a\mid a\in A,\text{for all }A\in X\,\}.$$

Thus, to show that $a \in \bigcap X$, we must show that a is in every element A of X. For example

$$\bigcap\{\{0,1\},\{1,2\},\{2,3\}\} = \emptyset = \{0,1\} \cap \{1,2\} \cap \{2,3\}.$$

If we allowed $\bigcap\emptyset$, then it would contain all elements a of our universe that are in all of the nonexistent elements of $\emptyset$, i.e., it would contain all elements of our universe. But this collection is a proper class, not a set. Let's consider the above reasoning in more detail. Suppose a is a universe element. To prove that $a \in A$, for all $A \in \emptyset$, suppose $A \in \emptyset$. But this is a logical contradiction, from which we may prove anything, in particular our desired conclusion, $a \in A$.

1.1.6 Relations and Functions

Next, we consider relations and functions. A *relation* R is a set of ordered pairs. The *domain* of a relation R ($\mathbf{domain}\, R$) is $\{\, x \mid (x,y) \in R, \text{for some } y\,\}$, and the *range* of R ($\mathbf{range}\, R$) is $\{\, y \mid (x,y) \in R, \text{for some } x\,\}$. We say that R is a *relation from* a set X *to* a set Y iff $\mathbf{domain}\, R \subseteq X$ and $\mathbf{range}\, R \subseteq Y$, and that R is a *relation on* a set A iff $\mathbf{domain}\, R \cup \mathbf{range}\, R \subseteq A$. We often write $x \mathrel{R} y$ for $(x,y) \in R$.

Consider the relation

$$R = \{(0,1),(1,2),(0,2)\}.$$

Then, $\mathbf{domain}\, R = \{0,1\}$, $\mathbf{range}\, R = \{1,2\}$, R is a relation from $\{0,1\}$ to $\{1,2\}$, and R is a relation on $\{0,1,2\}$. Of course, R is also, e.g., a relation between $\mathbb{N}$ and $\mathbb{R}$, as well as relation on $\mathbb{R}$.

We often form relations using set formation. For example, suppose $\phi(x,y)$ is a formula involving variables x and y. Then we can let $R = \{\,(x,y) \mid \phi(x,y)\,\}$, and it is easy to show that, for all x,y, $(x,y) \in R$ iff $\phi(x,y)$.

Given a set A, the *identity relation* on A ($\mathbf{id}_A$) is $\{\,(x,x) \mid x \in A\,\}$. For example, $\mathbf{id}_{\{1,3,5\}}$ is $\{(1,1),(3,3),(5,5)\}$. Given relations R and S, the *composition of* S and R ($S \circ R$) is $\{\,(x,z) \mid (x,y) \in R \text{ and } (y,z) \in S, \text{ for some } y\,\}$. Intuitively, it's the relation formed by starting with a pair $(x,y) \in R$, following it with a pair $(y,z) \in S$ (one that begins where the pair from R left off), and suppressing the intermediate value y, leaving us with (x,z). For example, if $R = \{(1,1),(1,2),(2,3)\}$ and $S = \{(2,3),(2,4),(3,4)\}$, then from $(1,2) \in R$ and $(2,3) \in S$, we can conclude $(1,3) \in S \circ R$. There are two more pairs in $S \circ R$, giving us $S \circ R = \{(1,3),(1,4),(2,4)\}$. For all sets A, B and C, and relations R and S, if R is a relation from A to B, and S is a relation from B to C, then $S \circ R$ is a relation from A to C.

It is easy to show that $\circ$ is associative and has the identity relations as its identities:

(1) For all sets A and B, and relations R from A to B, $\mathbf{id}_B \circ R = R = R \circ \mathbf{id}_A$.

(2) For all sets A, B, C and D, and relations R from A to B, S from B to C, and T from C to D, $(T \circ S) \circ R = T \circ (S \circ R)$.

Because of (2), we can write $T \circ S \circ R$, without worrying about how it is parenthesized.

The *inverse* of a relation R (R^{-1}) is the relation $\{\, (y,x) \mid (x,y) \in R \,\}$, i.e., it is the relation obtained by reversing each of the pairs in R. For example, if $R = \{(0,1),(1,2),(1,3)\}$, then the inverse of R is $\{(1,0),(2,1),(3,1)\}$. So for all sets A and B, and relations R, R is a relation from A to B iff R^{-1} is a relation from B to A. We also have that, for all sets A and B and relations R from A to B, $(R^{-1})^{-1} = R$. Furthermore, for all sets A, B and C, relations R from A to B, and relations S from B to C, $(S \circ R)^{-1} = R^{-1} \circ S^{-1}$.

A relation R is:

- *reflexive on* a set A iff, for all $x \in A$, $(x,x) \in R$;
- *transitive* iff, for all x, y, z, if $(x,y) \in R$ and $(y,z) \in R$, then $(x,z) \in R$;
- *symmetric* iff, for all x, y, if $(x,y) \in R$, then $(y,x) \in R$;
- *antisymmetric* iff, for all x, y, if $(x,y) \in R$ and $(y,x) \in R$, then $x = y$;
- *total on* a set A iff, for all $x, y \in A$, either $(x,y) \in R$ or $(y,x) \in R$;
- a *function* iff, for all x, y, z, if $(x,y) \in R$ and $(x,z) \in R$, then $y = z$.

Note that being antisymmetric is *not* the same as not being symmetric.

Suppose, e.g., that $R = \{(0,1),(1,2),(0,2)\}$. Then:

- R is not reflexive on $\{0,1,2\}$, since $(0,0) \not\in R$.
- R is transitive, since whenever (x,y) and (y,z) are in R, it follows that $(x,z) \in R$. Since $(0,1)$ and $(1,2)$ are in R, we must have that $(0,2)$ is in R, which is indeed true.
- R is not symmetric, since $(0,1) \in R$, but $(1,0) \not\in R$.
- R is antisymmetric, since there are no x, y such that (x,y) and (y,x) are both in R. (If we added $(1,0)$ to R, then R would not be antisymmetric, since then R would contain $(0,1)$ and $(1,0)$, but $0 \neq 1$.)
- R is not total on $\{0,1,2\}$, since $(0,0) \not\in R$.
- R is not a function, since $(0,1) \in R$ and $(0,2) \in R$. Intuitively, given an input of 0, it's not clear whether R's output is 1 or 2.

We say that R is a *total ordering on* a set A iff R is a transitive, antisymmetric and total relation on A. It is easy to see that such an R will also be reflexive on A. Furthermore, if R is a total ordering on A, then R^{-1} is also a total ordering on A.

We often use a symbol like $\leq$ to stand for a total ordering on a set A, which lets us use its mirror image, $\geq$, for its inverse, as well as to write $<$ for the relation on A defined by: for all $x, y \in A$, $x < y$ iff $x \leq y$ but $x \neq y$. $<$ is a *strict total ordering on* A, i.e., a transitive relation on A such that, for all $x, y \in A$, exactly one of $x < y$, $x = y$ and $y < x$ holds. We write $>$ for the inverse of $<$. We can also start with a strict total ordering $<$ on A, and then define a total ordering $\leq$ on A by: $x \leq y$ iff $x < y$ or $x = y$. The relations $\leq$ and $<$ on the natural numbers are examples of such relations.

The relation

$$f = \{(0,1), (1,2), (2,0)\}$$

is a function. We think of it as sending the input 0 to the output 1, the input 1 to the output 2, and the input 2 to the output 0.

If f is a function and $x \in \mathbf{domain}\, f$, we write $f\,x$ for the *application of f to* x, i.e., the unique y such that $(x, y) \in f$. We say that f is a *function from* a set X *to* a set Y iff f is a function, $\mathbf{domain}\, f = X$ and $\mathbf{range}\, f \subseteq Y$. Such an f is also a function from X to $\mathbf{range}\, f$. We write $X \rightarrow Y$ for the set of all functions from X to Y. If A has n elements and B has m elements, for $n, m \in \mathbb{N}$, then $A \rightarrow B$ will have m^n elements.

For the f defined above, we have that $f\,0 = 1$, $f\,1 = 2$, $f\,2 = 0$, f is a function from $\{0,1,2\}$ to $\{0,1,2\}$, and $f \in \{0,1,2\} \rightarrow \{0,1,2\}$. Of course, f is also in $\{0,1,2\} \rightarrow \mathbb{N}$, but it is not in $\mathbb{N} \rightarrow \{0,1,2\}$.

Exercise 1.1.3

Suppose X is a set, and $x \in X$. What are the elements of $\emptyset \rightarrow X$, $X \rightarrow \emptyset$, $\{x\} \rightarrow X$ and $X \rightarrow \{x\}$? Prove that your answers are correct.

We let $\rightarrow$ associate to the right and have lower precedence than ×, so that, e.g., $A \times B \rightarrow C \times D \rightarrow E \times F$ means $(A \times B) \rightarrow ((C \times D) \rightarrow (E \times F))$. An element of this set takes in a pair (a, b) in $A \times B$, and returns a function that takes in a pair (c, d) in $C \times D$, and returns an element of $E \times F$.

Suppose $f, g \in A \rightarrow B$. It is easy to show that $f = g$ iff, for all $x \in A$, $f\,x = g\,x$. This is the most common way of showing the equality of functions.

Given a set A, it is easy to see that $\mathbf{id}_A$, the identity relation on A, is a function from A to A, and we call it the *identity function* on A. It is the function that returns its input. Given sets A, B and C, if f is a function from A to B, and g is a function from B to C, then the composition $g \circ f$ of (the relations) g and f is the function h from A to C such that $h\,x = g(f\,x)$, for all $x \in A$. In other words, $g \circ f$ is the function that runs f and then g, in sequence.

Because of how composition of relations works, we have that $\circ$ is associative and has the identity functions as its identities:

(1) For all sets A and B, and functions f from A to B, $\mathbf{id}_B \circ f = f = f \circ \mathbf{id}_A$.

(2) For all sets A, B, C and D, and functions f from A to B, g from B to C, and h from C to D, $(h \circ g) \circ f = h \circ (g \circ f)$.

Because of (2), we can write $h \circ g \circ f$, without worrying about how it is parenthesized. It is the function that runs f, then g, then h, in sequence.

Given $f \in X \to Y$ and a subset A of X, we write $f(A)$ for the *image of A under f*, $\{\, f\,x \mid x \in A\,\}$. And given $f \in X \to Y$ and a subset B of Y, we write $f^{-1}(B)$ for the *inverse image of B under f*, $\{\, x \in X \mid f\,x \in B\,\}$. For example, if $f \in \mathbb{N} \to \mathbb{N}$ is the function that doubles its argument, then $f(\{3,5,7\}) = \{6,10,14\}$ and $f^{-1}(\{1,2,3,4\}) = \{1,2\}$.

Given a function f and a set $X \subseteq \mathbf{domain}\, f$, we write $f|X$ for the *restriction of f to X*, $\{\,(x,y) \in f \mid x \in X\,\}$. Hence $\mathbf{domain}(f|X) = X$. For example, if f is the function over $\mathbb{Z}$ that increases its argument by 2, then $f|\mathbb{N}$ is the function over $\mathbb{N}$ that increases its argument by 2. Given a function f and elements x, y of our universe, we write $f[x \mapsto y]$ for the *updating of f to send x to y*, $(f|(\mathbf{domain}\, f - \{x\})) \cup \{(x,y)\}$. This function is the same as f, except that it sends x to y. For example, if $f = \{(0,1),(1,2)\}$, then $f[1 \mapsto 0] = \{(0,1),(1,0)\}$, and $f[2 \mapsto 3] = \{(0,1),(1,2),(2,3)\}$.

We often define a function by saying how an element of its domain is transformed into an element of its range. E.g., we might say that $f \in \mathbb{N} \to \mathbb{Z}$ is the unique function such that, for all $n \in \mathbb{N}$,

$$f\,n = \begin{cases} -(n/2), & \text{if } n \text{ is even,} \\ (n+1)/2, & \text{if } n \text{ is odd.} \end{cases}$$

This is shorthand for saying that f is the set of all (n,m) such that

- $n \in \mathbb{N}$,
- if n is even, then $m = -(n/2)$, and
- if n is odd, then $m = (n+1)/2$.

Then $f\,0 = 0$, $f\,1 = 1$, $f\,2 = -1$, $f\,3 = 2$, $f\,4 = -2$, etc.

Exercise 1.1.4

There are three things wrong with the following "definition"—what are they? Let $f \in \mathbb{N} \to \mathbb{N}$ be the unique function such that, for all $n \in \mathbb{N}$,

$$f\,n = \begin{cases} n-2, & \text{if } n \geq 1 \text{ and } n \leq 10, \\ n+2, & \text{if } n \geq 10. \end{cases}$$

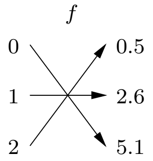
If the domain of f is a product $X_1 \times \cdots \times X_n$, for $n \geq 1$ and sets $X_1, \ldots, X_n$, we often abbreviate $f((x_1, \ldots, x_n))$ to $f(x_1, \ldots, x_n)$. We write $\sharp i_{X_1,\ldots,X_n}$ (or just $\sharp i$, if n and the X_i are clear from the context) for the i-th *projection* function, which selects the i-th component of a tuple, i.e., transforms an input $(x_1, \ldots, x_n)$ to x_i.

1.1.7 Set Cardinality

Next, we see how we can use functions to compare the sizes (or cardinalities) of sets. A *bijection* f *from* a set X *to* a set Y is a function from X to Y such that, for all $y \in Y$, there is a unique $x \in X$ such that $(x, y) \in f$. For example,

$$f = \{(0, 5.1), (1, 2.6), (2, 0.5)\}$$

is a bijection from $\{0, 1, 2\}$ to $\{0.5, 2.6, 5.1\}$. We can visualize f as a one-to-one correspondence between these sets:



A function f is an *injection* (or is *injective*) iff, for all x, y and z, if $(x, z) \in f$ and $(y, z) \in f$, then $x = y$, i.e., for all $x, y \in \mathbf{domain}\, f$, if $f\,x = f\,y$, then $x = y$. In other words, a function is injective iff it never sends two different elements of its domain to the same element of its range. For example, the function

$$\{(0, 1), (1, 2), (2, 3), (3, 0)\}$$

is injective, but the function

$$\{(0, 1), (1, 2), (2, 1)\}$$

is not injective (both 0 and 2 are sent to 1). We say that f is an *injection from* a set X *to* a set Y, iff f is a function from X to Y and f is injective.

It is easy to see that:

- For all sets A, $\mathbf{id}_A$ is injective.
- For all sets A, B and C, functions f from A to B, and functions g from B to C, if f and g are injective, then so is $g \circ f$.

Exercise 1.1.5

Suppose A and B are sets. Show that for all f, f is a bijection from A to B iff

- f is a function from A to B;

- $\mathbf{range}\, f = B$; and
- f is injective.

Consequently, if f is an injection from A to B, then f is a bijection from A to $\mathbf{range}\, f \subseteq B$.

Exercise 1.1.6
Show that:

(1) For all sets A, $\mathbf{id}_A$ is a bijection from A to A.

(2) For all sets A, B and C, bijections f from A to B, and bijections g from B to C, $g \circ f$ is a bijection from A to C.

(3) For all sets A and B, and bijections f from A to B, f^{-1} is a bijection from B to A.

We say that a set X has the *same size* as a set Y ($X \cong Y$) iff there is a bijection from X to Y. By Exercise 1.1.6, we have that, for all sets A, B, C:

(1) $A \cong A$;

(2) If $A \cong B \cong C$, then $A \cong C$; and

(3) If $A \cong B$, then $B \cong A$.

We say that a set X is:

- *finite* iff $X \cong [1 : n]$, for some $n \in \mathbb{N}$ (recall that $[1 : n]$ is all of the natural numbers that are at least 1 and no more than n, so that $[1 : 0] = \emptyset$);
- *infinite* iff it is not finite;
- *countably infinite* iff $X \cong \mathbb{N}$;
- *countable* iff X is either finite or countably infinite; and
- *uncountable* iff X is not countable.

Every set X has a *size* or *cardinality* ($|X|$) and we have that, for all sets X and Y, $|X| = |Y|$ iff $X \cong Y$. The sizes of finite sets are natural numbers.

We have that:

- The sets $\emptyset$ and $\{0.5, 2.6, 5.1\}$ are finite, and are thus also countable;
- The sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ and $\mathcal{P}\,\mathbb{N}$ are infinite;
- The set $\mathbb{N}$ is countably infinite, and is thus countable; and

- The set $\mathbb{Z}$ is countably infinite, and is thus countable, because of the existence of the following bijection:

$$\begin{tikzcd}
\cdots & -2 \arrow[dd] & -1 \arrow[dd] & 0 \arrow[dd] & 1 \arrow[dd] & 2 \arrow[dd] & \cdots \\
\cdots & & & & & & \cdots \\
\cdots & 4 & 2 & 0 & 1 & 3 & \cdots
\end{tikzcd}$$

- The sets $\mathbb{R}$ and $\mathcal{P}\,\mathbb{N}$ are uncountable.

To prove that $\mathbb{R}$ and $\mathcal{P}\,\mathbb{N}$ are uncountable, we can use an important technique called "diagonalization", which we will see again in Chapter **??**. Let's consider the proof that $\mathcal{P}\,\mathbb{N}$ is uncountable.

We proceed using proof by contradiction. Suppose $\mathcal{P}\,\mathbb{N}$ is countable. If we can obtain a contradiction, it will follow that $\mathcal{P}\,\mathbb{N}$ is uncountable. Since $\mathcal{P}\,\mathbb{N}$ is not finite, it follows that there is a bijection f from $\mathbb{N}$ to $\mathcal{P}\,\mathbb{N}$. Our plan is to define a subset X of $\mathbb{N}$ such that $X \not\in \mathbf{range}\, f$, thus obtaining a contradiction, since this will show that f is not a bijection from $\mathbb{N}$ to $\mathcal{P}\,\mathbb{N}$.

Consider the infinite table in which both the rows and the columns are indexed by the elements of $\mathbb{N}$, listed in ascending order, and where a cell (m, n) (m is the row, n is the column) contains 1 iff $m \in f\,n$, and contains 0 iff $m \not\in f\,n$. Thus the nth column of this table represents the set $f\,n$ of natural numbers.

Figure 1.1 shows how part of this table might look, where i, j and k are sample elements of $\mathbb{N}$. Because of the table's data, we have, e.g., that $i \in f\,i$ and $j \not\in f\,i$.

To define our $X \subseteq \mathbb{N}$, we work our way down the diagonal of the table, putting n into our set just when cell (n, n) of the table is 0, i.e., when $n \not\in f\,n$. This will ensure that, for all $n \in \mathbb{N}$, $f\,n \neq X$. With our example table:

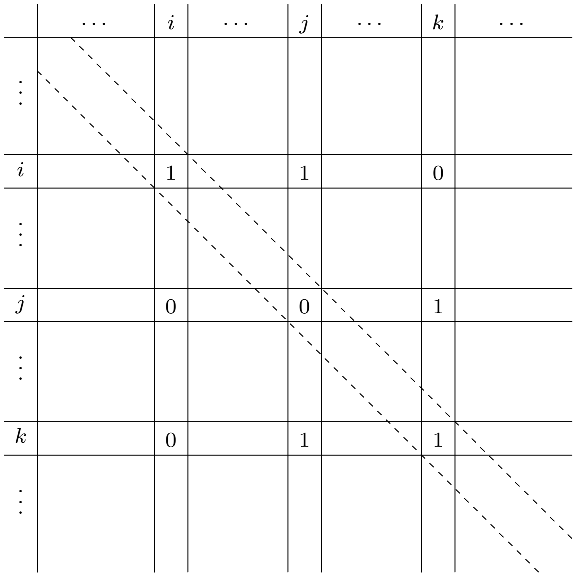
- since $i \in f\,i$, but $i \not\in X$, we have that $f\,i \neq X$;
- since $j \not\in f\,j$, but $j \in X$, we have that $f\,j \neq X$; and
- since $k \in f\,k$, but $k \not\in X$, we have that $f\,k \neq X$.

Next, we turn the above ideas into a shorter, but more opaque, proof that:

Proposition 1.1.7
$\mathcal{P}\,\mathbb{N}$ is uncountable.

Proof. Suppose, toward a contradiction, that $\mathcal{P}\,\mathbb{N}$ is countable. Because $\mathcal{P}\,\mathbb{N}$ is not finite, there is a bijection f from $\mathbb{N}$ to $\mathcal{P}\,\mathbb{N}$. Define $X \in \{\, n \in \mathbb{N} \mid n \not\in f\,n \,\}$, so that $X \in \mathcal{P}\,\mathbb{N}$. By the definition of f, it follows that $X = f\,n$, for some $n \in \mathbb{N}$. There are two cases to consider.

- Suppose $n \in X$. By the definition of X, it follows that $n \not\in f\,n = X$—contradiction.



	$\cdots$	i	$\cdots$	j	$\cdots$	k	$\cdots$
$\vdots$							
i		1		1		0	
$\vdots$							
j		0		0		1	
$\vdots$							
k		0		1		1	
$\vdots$							

Figure 1.1: Example Diagonalization Table for Cardinality Proof

- Suppose $n \not\in X$. Because $X = f\,n$, we have that $n \not\in f\,n$. Thus, since $n \in \mathbb{N}$ and $n \not\in f\,n$, it follows that $n \in X$—contradiction.

Since we obtained a contradiction in both cases, we have an overall contradiction. Thus $\mathcal{P}\,\mathbb{N}$ is uncountable. □

We have seen how bijections may be used to determine whether sets have the same size. But how can we compare the relative sizes of sets, i.e., say whether one set is smaller or larger than another? The answer is to make use of injective functions. We say that a set X is *no bigger* than a set Y ($X \preceq Y$) iff there is an injection from X to Y, i.e., an injective function whose domain is X and whose range is a subset of Y. Because identity functions are injective, we have that $X \subseteq Y$ implies $X \preceq Y$; e.g., $\mathbb{N} \preceq \mathbb{R}$. This definition makes sense, because X is no bigger than Y iff X has the same size as a subset of Y. We say that X is *strictly smaller* than Y iff $X \preceq Y$ and $X \not\approx Y$.

Using our observations about injections, we have that, for all sets A, B and C:

(1) $A \preceq A$;

(2) If $A \preceq B \preceq C$, then $A \preceq C$.

We can also characterize $\preceq$ using "surjectivity". We say that f is a *surjection* from X to Y iff f is a function from X to Y and $\mathbf{range}\,f = Y$. A consequence of set theory's Axiom of Choice is that, for all sets X and Y, $X \preceq Y$ iff $X = \emptyset$ or there is a surjection from Y to X.

Clearly, for all sets A and B, if $A \cong B$, then $A \preceq B \preceq A$. And, a famous result of set theory, the Schröder-Bernstein Theorem, says that the converse holds, i.e., for all sets A and B, if $A \preceq B \preceq A$, then $A \cong B$. This give us a powerful method for proving that two sets have the same size.

Exercise 1.1.8

Use the Schröder-Bernstein Theorem to show that $\mathbb{N} \cong \mathbb{N} \times \mathbb{N}$. Hint: use the following consequence of the Fundamental Theorem of Arithmetic: if two finite, ascending (each element is $\leq$ the next) sequences of prime numbers (natural numbers that are at least 2 and have no divisors other than 1 and themselves) have the same product (the product of the empty sequence is 1), then they are equal.

One of the forms of the Axiom of Choice says that, for all sets A and B, either $A \preceq B$ or $B \preceq A$, i.e., either A is no bigger than B, or B is no bigger than A. Furthermore, the sizes of sets are ordered in such a way that, for all sets A and B, $|A| \leq |B|$ iff $A \preceq B$, which tells us that, given sets A and B, either $|A| \leq |B|$ or $|B| \leq |A|$. Given the above machinery, one can strengthen Proposition 1.1.7 into: for all sets X, X is strictly smaller than $\mathcal{P}\,X$, i.e., $|X| < |\mathcal{P}\,X|$.

1.1.8 Data Structures

We conclude this section by introducing some data structures that are built out of sets. We write **Bool** for the set of booleans, $\{\mathbf{true}, \mathbf{false}\}$. (We can actually let $\mathbf{true} = 1$ and $\mathbf{false} = 0$, although we'll never make use of these equalities.) We define the negation function $\mathbf{not} \in \mathbf{Bool} \to \mathbf{Bool}$ by:

$$\mathbf{not}\,\mathbf{true} = \mathbf{false}, \qquad \mathbf{not}\,\mathbf{false} = \mathbf{true}.$$

And the conjunction and disjunction operations on the booleans are defined by:

$$\begin{aligned}&\mathbf{true}\,\mathbf{and}\,\mathbf{true} = \mathbf{true},\\&\mathbf{true}\,\mathbf{and}\,\mathbf{false} = \mathbf{false}\,\mathbf{and}\,\mathbf{true} = \mathbf{false}\,\mathbf{and}\,\mathbf{false} = \mathbf{false},\end{aligned}$$

and

$$\begin{aligned}&\mathbf{true}\,\mathbf{or}\,\mathbf{true} = \mathbf{true}\,\mathbf{or}\,\mathbf{false} = \mathbf{false}\,\mathbf{or}\,\mathbf{true} = \mathbf{true},\\&\mathbf{false}\,\mathbf{or}\,\mathbf{false} = \mathbf{false}.\end{aligned}$$

Given a set X, we write $\mathbf{Option}\,X$ for $\{\mathbf{none}\} \cup \{\, \mathbf{some}\,x \mid x \in X \,\}$, where we define $\mathbf{none} = (0,0)$ and $\mathbf{some}\,x = (1,x)$, which guarantees that $\mathbf{none} = \mathbf{some}\,x$ can't hold, and that $\mathbf{some}\,x = \mathbf{some}\,y$ only holds when $x = y$. (We won't make use of the particular way we've defined **none** and $\mathbf{some}\,x$.) For example, $\mathbf{Option}\,\mathbf{Bool} = \{\mathbf{none}, \mathbf{some}\,\mathbf{true}, \mathbf{some}\,\mathbf{false}\}$.

The idea is that an element of $\mathbf{Option}\,X$ is an optional element of X. E.g., when a function needs to either return an element of X or indicate that an error has occurred, it could return an element of $\mathbf{Option}\,X$, using **none** to indicate an error, and returning $\mathbf{some}\,x$ to indicate success with value x. E.g., we could define a function $f \in \mathbb{N} \times \mathbb{N} \to \mathbf{Option}\,\mathbf{Bool}$ by:

$$f(n,m) = \begin{cases} \mathbf{none}, & \text{if } m = 0, \\ \mathbf{some}\,\mathbf{true} & \text{if } m \neq 0 \text{ and } n = ml \text{ for some } l \in \mathbb{N}, \\ \mathbf{some}\,\mathbf{false} & \text{if } m \neq 0 \text{ and } n \neq ml \text{ for all } l \in \mathbb{N}. \end{cases}$$

Finally, we consider lists. A *list* is a function with domain $[1:n]$, for some $n \in \mathbb{N}$. (Recall that $[1:n]$ is all of the natural numbers that are at least 1 and no more than n.) For example, $\emptyset$ is a list, as it is a function with domain $\emptyset = [1:0]$. And $\{(1,3),(2,5),(3,7)\}$ is a list, as it is a function with domain $[1:3]$. Note that, if x is a list, then $|x|$, the size of the set x, doubles as the *length* of x.

We abbreviate a list $\{(1,x_1),(2,x_2),\ldots,(n,x_n)\}$ to $[x_1,x_2,\ldots,x_n]$. Thus $\emptyset$ and $\{(1,3),(2,5),(3,7)\}$ are abbreviated to $[\,]$ and $[3,5,7]$, respectively. If $[x_1,x_2,\ldots,x_n] = [y_1,y_2,\ldots,y_m]$, it is easy to see that $n = m$ and $x_i = y_i$, for all $i \in [1:n]$.

Given lists f and g, the *concatenation* of f and g ($f \mathbin{@} g$) is the list

$$f \cup \{\, (m + |f|, y) \mid (m,y) \in g \,\}.$$

For example,

$$\begin{aligned} [2,3] \mathbin{@} [4,5,6] &= \{(1,2),(2,3)\} \mathbin{@} \{(1,4),(2,5),(3,6)\} \\ &= \{(1,2),(2,3)\} \cup \{\, (m+2,y) \mid (m,y) \in \{(1,4),(2,5),(3,6)\} \,\} \\ &= \{(1,2),(2,3)\} \cup \{(1+2,4),(2+2,5),(3+2,6)\} \\ &= \{(1,2),(2,3),(3,4),(4,5),(5,6)\} \\ &= [2,3,4,5,6]. \end{aligned}$$

Given lists f and g, it is easy to see that $|f \mathbin{@} g| = |f| + |g|$. And, given $n \in [1 : |f \mathbin{@} g|]$, we have that

$$(f \mathbin{@} g)\,n = \begin{cases} f\,n, & \text{if } n \in [1:|f|], \\ g(n - |f|), & \text{if } n > |f|. \end{cases}$$

Using this fact, it is easy to prove that:

- $[\,]$ is the identity for concatenation: for all lists f,

$$[\,] \mathbin{@} f = f = f \mathbin{@} [\,].$$

- Concatenation is associative: for all lists f, g, h,

$$(f \mathbin{@} g) \mathbin{@} h = f \mathbin{@} (g \mathbin{@} h).$$

Because concatenation is associative, we can write $f \mathbin{@} g \mathbin{@} h$ without worrying where the parentheses go.

Given a set X, we write $\mathbf{List}\,X$ for the set of all X-*lists*, i.e., lists whose ranges are subsets of X, i.e., all of whose elements come from X. E.g., $[\,]$ and $[3, 5, 7]$ are elements of $\mathbf{List}\,\mathbb{N}$, the set of all lists of natural numbers. It is easy to see that $[\,] \in \mathbf{List}\,X$, for all sets X, and that, for all sets X and $f, g \in \mathbf{List}\,X$, $f \mathbin{@} g \in \mathbf{List}\,X$.

1.1.9 Notes

In a traditional treatment of set theory, e.g., [End77], the natural numbers, integers, real numbers and ordered pairs (x, y) are encoded as sets. But for our purposes, it is clearer to suppress this detail.

Furthermore, in the traditional approach, $\mathbb{N}$ is not a subset of $\mathbb{Z}$, and $\mathbb{Z}$ is not a subset of $\mathbb{R}$. On the other hand, there are proper subsets of $\mathbb{R}$ corresponding to $\mathbb{N}$ and $\mathbb{Z}$, and these are what we take $\mathbb{N}$ and $\mathbb{Z}$ to be, so that $\mathbb{N} \subsetneq \mathbb{Z} \subsetneq \mathbb{R}$.

1.2 Induction

In the section, we consider several induction principles, i.e., methods for proving that every element x of some set A has a property $P(x)$.

1.2.1 Mathematical Induction

We begin with the familiar principle of mathematical induction, which is a basic result of set theory.

Theorem 1.2.1 (Principle of Mathematical Induction)
Suppose $P(n)$ is a property of a natural number n. If

(basis step)

$$P(0) \textit{ and}$$

(inductive step)

$$\textit{for all } n \in \mathbb{N}, \textit{ if } \;(\dagger)\; P(n), \textit{ then } P(n+1),$$

then,

$$\textit{for all } n \in \mathbb{N},\ P(n).$$

We refer to the formula $(\dagger)$ as the *inductive hypothesis*. To use the principle to show that that every natural number has property P, we carry out two steps. In the basis step, we show that 0 has property P. In the inductive step, we assume that n is a natural number with property P. We then show that $n + 1$ has property P, without making any more assumptions about n.

Next we give an example of a mathematical induction.

Proposition 1.2.2
For all $n \in \mathbb{N}$, $3n^2 + 3n + 6$ *is divisible by* 6.

Proof. We proceed by mathematical induction.

(Basis Step) We have that $3 \cdot 0^2 + 3 \cdot 0 + 6 = 0 + 0 + 6 = 6 = 6 \cdot 1$. Thus $3 \cdot 0^2 + 3 \cdot 0 + 6$ is divisible by 6.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $3n^2 + 3n + 6$ is divisible by 6. Hence $3n^2 + 3n + 6 = 6m$, for some $m \in \mathbb{N}$. Thus, we have that

$$\begin{aligned} 3(n+1)^2 + 3(n+1) + 6 &= 3(n^2 + 2n + 1) + 3n + 3 + 6 \\ &= 3n^2 + 6n + 3 + 3n + 3 + 6 \\ &= (3n^2 + 3n + 6) + (6n + 6) \\ &= 6m + 6(n+1) \\ &= 6(m + (n+1)), \end{aligned}$$

showing that $3(n+1)^2 + 3(n+1) + 6$ is divisible by 6.

□

Exercise 1.2.3
Use Proposition 1.2.2 to prove, by mathematical induction, that, for all $n \in \mathbb{N}$, $n(n^2 + 5)$ is divisible by 6.

1.2.2 Strong Induction

Next, we consider the principle of strong induction.

Theorem 1.2.4 (Principle of Strong Induction)
Suppose $P(n)$ *is a property of a natural number* n*. If*

for all $n \in \mathbb{N}$,
if (†) *for all* $m \in \mathbb{N}$, *if* $m < n$, *then* $P(m)$,
then $P(n)$,

then

for all $n \in \mathbb{N}$, $P(n)$.

We refer to the formula (†) as the *inductive hypothesis*. To use the principle to show that every natural number has property P, we assume that n is a natural number, and that every natural number that is strictly smaller than n has property P. We then show that n has property P, without making any more assumptions about n.

It turns out that we can use mathematical induction to prove the validity of the principle of strong induction, by using a property $Q(n)$ derived from $P(n)$.

Proof. Suppose $P(n)$ is a property, and assume

(‡) for all $n \in \mathbb{N}$,
if for all $m \in \mathbb{N}$, if $m < n$, then $P(m)$,
then $P(n)$.

Let the property $Q(n)$ be

for all $m \in \mathbb{N}$, if $m < n$, then $P(m)$.

First, we use mathematical induction to show that, for all $n \in \mathbb{N}$, $Q(n)$.

(Basis Step) We must show $Q(0)$. Suppose $m \in \mathbb{N}$ and $m < 0$. We must show that $P(m)$. Since $m < 0$ is a contradiction, we are allowed to conclude anything. So, we conclude $P(m)$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $Q(n)$. We must show that $Q(n+1)$. Suppose $m \in \mathbb{N}$ and $m < n+1$. We must show that $P(m)$. Since $m \leq n$, there are two cases to consider.

- Suppose $m < n$. Because $Q(n)$, we have that $P(m)$.
- Suppose $m = n$. We must show that $P(n)$. By Property (‡), it will suffice to show that

  for all $m \in \mathbb{N}$, if $m < n$, then $P(m)$.

  But this formula is exactly $Q(n)$, and so were are done.

Now, we use the result of our mathematical induction to show that, for all $n \in \mathbb{N}$, $P(n)$. Suppose $n \in \mathbb{N}$. By our mathematical induction, we have $Q(n)$. By Property (‡), it will suffice to show that

for all $m \in \mathbb{N}$, if $m < n$, then $P(m)$.

But this formula is exactly $Q(n)$, and so we are done. □

As an example use of the principle of strong induction, we will prove a proposition that we would normally take for granted:

Proposition 1.2.5
Every nonempty set of natural numbers has a least element.

Proof. Let X be a nonempty set of natural numbers.

We begin by using strong induction to show that, for all $n \in \mathbb{N}$,

$$\text{if } n \in X, \text{ then } X \text{ has a least element.}$$

Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: for all $m \in \mathbb{N}$, if $m < n$, then

$$\text{if } m \in X, \text{ then } X \text{ has a least element.}$$

We must show that

$$\text{if } n \in X, \text{ then } X \text{ has a least element.}$$

Suppose $n \in X$. It remains to show that X has a least element. If n is less-than-or-equal-to every element of X, then we are done. Otherwise, there is an $m \in X$ such that $m < n$. By the inductive hypothesis, we have that

$$\text{if } m \in X, \text{ then } X \text{ has a least element.}$$

But $m \in X$, and thus X has a least element. This completes our strong induction.

Now we use the result of our strong induction to prove that X has a least element. Since X is a nonempty subset of $\mathbb{N}$, there is an $n \in \mathbb{N}$ such that $n \in X$. By the result of our induction, we can conclude that

$$\text{if } n \in X, \text{ then } X \text{ has a least element.}$$

But $n \in X$, and thus X has a least element. □

We conclude this subsection with one more proof using strong induction. Recall that a natural number is prime iff it is at least 2 and has no divisors other than 1 and itself.

Proposition 1.2.6
For all $n \in \mathbb{N}$,

$$\textit{if } n \geq 2, \textit{then there are } m, l \in \mathbb{N} \textit{ such that } n = ml \textit{ and } m \textit{ is prime.}$$

Proof. We proceed by strong induction. Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: for all $i \in \mathbb{N}$, if $i < n$, then

$$\text{if } i \geq 2, \text{then there are } m, l \in \mathbb{N} \text{ such that } i = ml \text{ and } m \text{ is prime.}$$

We must show that

$$\text{if } n \geq 2, \text{then there are } m, l \in \mathbb{N} \text{ such that } n = ml \text{ and } m \text{ is prime.}$$

Suppose $n \geq 2$. We must show that

$$\text{there are } m, l \in \mathbb{N} \text{ such that } n = ml \text{ and } m \text{ is prime.}$$

There are two cases to consider.

- Suppose n is prime. Then $n, 1 \in \mathbb{N}$, $n = n1$ and n is prime.
- Suppose n is not prime. Since $n \geq 2$, it follows that $n = ij$ for some $i, j \in \mathbb{N}$ such that $1 < i < n$. Thus, by the inductive hypothesis, we have that

  $$\text{if } i \geq 2, \text{then there are } m, l \in \mathbb{N} \text{ such that } i = ml \text{ and } m \text{ is prime.}$$

  But $i \geq 2$, and thus there are $m, l \in \mathbb{N}$ such that $i = ml$ and m is prime. Thus $m, lj \in \mathbb{N}$, $n = ij = (ml)j = m(lj)$ and m is prime.

□

Exercise 1.2.7
Use strong induction to prove that, for all $n \in \mathbb{N}$, if $n \geq 1$, then there are $i, j \in \mathbb{N}$ such that $n = 2^i(2j + 1)$.

1.2.3 Well-founded Induction

We can also do induction over a well-founded relation. A relation R on a set A is *well-founded* iff every nonempty subset X of A has an R-minimal element, where an element $x \in X$ is *R-minimal in X* iff there is no $y \in X$ such that $y \mathrel{R} x$.

Given a relation R on a set A, and $x, y \in A$, we say that *y is a predecessor of x in R* iff $y \mathrel{R} x$. Thus $x \in X$ is R-minimal in X iff none of x's predecessors in R (there may be none) are in X.

For example, in Proposition 1.2.5, we proved that the strict total ordering $<$ on $\mathbb{N}$ is well-founded. On the other hand, the strict total ordering $<$ on $\mathbb{Z}$ is *not* well-founded, as $\mathbb{Z}$ itself lacks a $<$-minimal element.

Here's another negative example, showing that even if the underlying set is finite, the relation need not be well-founded. Let $A = \{0, 1\}$, and $R = \{(0, 1), (1, 0)\}$. Then 0 is the only predecessor of 1 in R, and 1 is the only predecessor of 0 in R. Of the nonempty subsets of A, we have that $\{0\}$ and $\{1\}$ have R-minimal elements. But consider A itself. Then 0 is not R-minimal in A, because $1 \in A$ and $1 \mathrel{R} 0$. And 1 is not R-minimal in A, because $0 \in A$ and $0 \mathrel{R} 1$. Hence R is not well-founded.

Theorem 1.2.8 (Principle of Well-founded Induction)
Suppose A is a set, R is a well-founded relation on A, and $P(x)$ is a property of an element $x \in A$. If

for all $x \in A$,
if (†) for all $y \in A$, if $y \mathrel{R} x$, then $P(y)$,
then $P(x)$,

then

for all $x \in A$, $P(x)$.

We refer to the formula (†) as the *inductive hypothesis*. When $A = \mathbb{N}$ and $R = <$, this is the same as the principle of strong induction. But it's much more generally applicable than strong induction. Furthermore, the proof of this theorem isn't by induction.

Proof. Suppose A is a set, R is a well-founded relation on A, $P(x)$ is a property of an element $x \in A$, and

(‡) for all $x \in A$,
if for all $y \in A$, if $y \mathrel{R} x$, then $P(y)$,
then $P(x)$.

We must show that, for all $x \in A$, $P(x)$.

Suppose, toward a contradiction, that it is not the case that, for all $x \in A$, $P(x)$. Hence there is an $x \in A$ such that $P(x)$ is false. Let $X = \{\, x \in A \mid P(x)$ is false$\,\}$. Thus $x \in X$, showing that X is non-empty. Because R is well-founded on A, it follows that there is a $z \in X$ that is R-minimal in X, i.e., such that there is no $y \in X$ such that $y \mathrel{R} z$.

By (‡) and since $z \in X \subseteq A$, we have that

if for all $y \in A$, if $y \mathrel{R} z$, then $P(y)$,
then $P(z)$.

Because $z \in X$, we have that $P(z)$ is false. Thus, to obtain a contradiction, it will suffice to show that

for all $y \in A$, if $y \mathrel{R} z$, then $P(y)$.

Suppose $y \in A$, and $y \mathrel{R} z$. We must show that $P(y)$. Because z is R-minimal in X, it follows that $y \notin X$. Thus $P(y)$.

Because we obtained our contradiction, we have that, for all $x \in A$, $P(x)$, as required. □

We conclude this subsection by considering three ways of building well-founded relations. The first is by taking a subset of a well-founded relation:

Proposition 1.2.9
Suppose R is a well-founded relation on a set A. If $S \subseteq R$, then S is also a well-founded relation on A.

Proof. Suppose R is a well-founded relation on A, and $S \subseteq R$. Let X be a nonempty subset of A. Let $x \in X$ be R-minimal in X. Suppose, toward a contradiction, that x is not S-minimal in X. Thus there is a $y \in X$ such that $y \mathrel{S} x$. But $S \subseteq R$, and thus $y \mathrel{R} x$—contradiction. Thus x is S-minimal in X, as required. □

Let the *predecessor* relation $\mathbf{pred}_{\mathbb{N}}$ on $\mathbb{N}$ be $\{\,(n, n{+}1) \mid n \in \mathbb{N}\,\}$. Thus, for all $n, m \in \mathbb{N}$, $m \ \mathbf{pred}_{\mathbb{N}} \ n$ iff m is exactly one less than n. Because $\mathbf{pred}_{\mathbb{N}} \subseteq <$, and $<$ is well-founded on $\mathbb{N}$, Proposition 1.2.9 tells us that $\mathbf{pred}_{\mathbb{N}}$ is well-founded on $\mathbb{N}$. 0 has no predecessors in $\mathbf{pred}_{\mathbb{N}}$, and, for all $n \in \mathbb{N}$, n is the only predecessor of $n + 1$ in $\mathbf{pred}_{\mathbb{N}}$. Consequently, if a zero/non-zero case analysis is used, a proof by well-founded induction on $\mathbf{pred}_{\mathbb{N}}$ will look like a proof by mathematical induction.

Suppose A and B are sets, S is a relation on B, and $f \in A \to B$. Then the *inverse image of the relation S under f*, $f^{-1}(S)$, is the relation R on A defined by: for all $x, y \in A$, $x \ R \ y$ iff $f\,x \ S \ f\,y$.

Proposition 1.2.10
Suppose A and B are sets, S is a well-founded relation on B, and $f \in A \to B$. Then $f^{-1}(S)$ is a well-founded relation on A.

Proof. Let $R = f^{-1}(S)$. To see that R is well-founded on A, suppose X is a nonempty subset of A. We must show that there is an R-minimal element of X. Let $Y = f(X) = \{\, f\,x \mid x \in X\,\}$. Thus Y is a nonempty subset of B. Because S is well-founded on B, it follows that there is an S-minimal element y of Y. Hence $y = f\,x$ for some $x \in X$. Suppose, toward a contradiction, that x is not R-minimal in X. Thus there is an $x' \in X$ such that $x' \ R \ x$. Hence $f\,x' \in Y$ and $f\,x' \ S \ f\,x = y$, contradicting the S-minimality of y in Y. Thus x is R-minimal in X. $\Box$

For example, let R be the relation on $\mathbb{Z}$ such that, for all $n, m \in \mathbb{Z}$, $n \ R \ m$ iff $|n| < |m|$ (where we're writing $|\cdot|$ for the absolute value of an integer). Since $<$ is well-founded on $\mathbb{N}$, Proposition 1.2.10 tells us that R is well-founded on $\mathbb{Z}$. If we do a well-founded induction on R, when proving $P(n)$, for $n \in \mathbb{Z}$, we can make use of $P(m)$ for any $m \in \mathbb{Z}$ whose absolute value is strictly less than the absolute value of n. E.g., when proving $P(-10)$, we could make use of $P(5)$ or $P(-9)$.

If R and S are relations on sets A and B, respectively, then the *lexicographic relation of R and then S*, $R \triangleright S$, is the relation on $A \times B$ defined by: $(x, y) \ R \triangleright S \ (x', y')$ iff

- $x \ R \ x'$, or
- $x = x'$ and $y \ S \ y'$.

Proposition 1.2.11
Suppose R and S are well-founded relations on sets A and B, respectively. Then $R \triangleright S$ is a well-founded relation on $A \times B$.

Proof. Suppose T is a nonempty subset of $A \times B$. We must show that there is an $R \triangleright S$-minimal element of T. By our assumption, T is a relation from A to

B. Because T is nonempty, it follows that $\mathbf{domain}\,T$ is a nonempty subset of A. Since R is a well-founded relation on A, it follows that there is an R-minimal $x \in \mathbf{domain}\,T$. Let $Y = \{ y \in B \mid (x,y) \in T \}$. Because Y is a nonempty subset of B, and S is well-founded on B, there exists an S-minimal $y \in Y$. Thus $(x,y) \in T$.

Suppose, toward a contradiction, that (x,y) is not $R \triangleright S$-minimal in T. Thus there are $x' \in A$ and $y' \in B$ such that $(x',y') \in T$ and $(x',y')\ R \triangleright S\ (x,y)$. Hence, there are two cases to consider.

- Suppose $x'\ R\ x$. Because $x' \in \mathbf{domain}\,T$, this contradicts the R-minimality of x in $\mathbf{domain}\,T$.
- Suppose $x' = x$ and $y'\ S\ y$. Thus $(x,y') = (x',y') \in T$, so that $y' \in Y$. But this contradicts the S-minimality of y in Y.

Because we obtained a contradiction in both cases, we have an overall contradiction. Thus (x,y) is $R \triangleright S$-minimal in T. □

For example, if we consider the strict total ordering $<$ on $\mathbb{N}$, then $< \triangleright <$ is a well-founded relation on $\mathbb{N} \times \mathbb{N}$. If we do a well-founded induction on $< \triangleright <$, when proving that $P((x,y))$ holds, we can use $P((x',y'))$, whenever $x' < x$ or $x = x'$ but $y' < y$.

Just as we abbreviate $A \times (B \times C)$ to $A \times B \times C$, and abbreviate $(x,(y,z))$ to (x,y,z), we abbreviate $R \triangleright (S \triangleright T)$ to $R \triangleright S \triangleright T$. If R, S and T are well-founded relations on sets A, B and C, respectively, then $R \triangleright S \triangleright T$ is the well-founded relation on $A \times B \times C$ such that, for all $x \in A$, $y \in B$ and $z \in C$: $(x,y,z)\ R \triangleright S \triangleright T\ (x',y',z')$ iff

- $x\ R\ x'$, or
- $x = x'$ and $y\ S\ y'$, or
- $x = x'$, $y = y'$ and $z\ T\ z'$.

And we can do the analogous thing with four or more well-founded relations.

1.2.4 Notes

A typical book on formal language theory doesn't introduce well-founded relations and induction. But this material, and our treatment of well-founded recursion in the next section, will prove to be useful in subsequent chapters.

1.3 Inductive Definitions and Recursion

In this section, we will introduce and study ordered trees of arbitrary (finite) arity, whose nodes are labeled by elements of some set. In later chapters, we will define regular expressions (in Chapter 3), parse trees (in Chapter **??**) and programs (in Chapter **??**) as restrictions of the trees we consider here.

The definition of the set of all trees over a set of labels is our first example of an inductive definition—a definition in which we collect together all of the values that can be constructed using a set of rules. We will see many examples of inductive definitions in the book. In this section, we will also see how to define functions by recursion.

1.3.1 Inductive Definition of Trees

Suppose X is a set. The set $\mathbf{Tree}\,X$ of X-*trees* is the least set such that, for all $x \in X$ and $\mathit{trs} \in \mathbf{List}(\mathbf{Tree}\,X)$, $(x, \mathit{trs}) \in \mathbf{Tree}\,X$. Recall that saying $\mathit{trs} \in \mathbf{List}(\mathbf{Tree}\,X)$ simply means that trs is a list all of whose elements come from $\mathbf{Tree}\,X$.

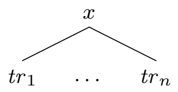
Ignoring the adjective "least" for the moment, some example elements of $\mathbf{Tree}\,\mathbb{N}$ (the case when the set X of tree labels is $\mathbb{N}$) are:

- Since $3 \in \mathbb{N}$ and $[\,] \in \mathbf{List}(\mathbf{Tree}\,\mathbb{N})$, we have that $(3, [\,]) \in \mathbf{Tree}\,\mathbb{N}$. For similar reasons, $(1, [\,])$, $(6, [\,])$ and all pairs of the form $(n, [\,])$, for $n \in \mathbb{N}$, are in $\mathbf{Tree}\,\mathbb{N}$.

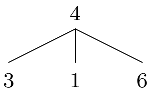
- Because $4 \in \mathbb{N}$, and $[(3, [\,]), (1, [\,]), (6, [\,])] \in \mathbf{List}(\mathbf{Tree}\,\mathbb{N})$, we have that $(4, [(3, [\,]), (1, [\,]), (6, [\,])]) \in \mathbf{Tree}\,\mathbb{N}$.

- And we can continue like this forever.

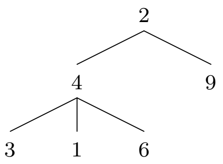
Trees are often easier to comprehend if they are drawn. We draw the X-tree $(x, [\mathit{tr}_1, \ldots, \mathit{tr}_n])$ as



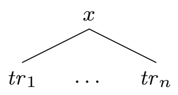
For example,



is the drawing of the $\mathbb{N}$-tree $(4, [(3, [\,]), (1, [\,]), (6, [\,])])$. And



is the $\mathbb{N}$-tree $(2, [(4, [(3, [\,]), (1, [\,]), (6, [\,])]), (9, [\,])])$. Consider the tree



again. We say that the *root label* of this tree is x, and tr_1 is the tree's *first child*, etc. We write $\textbf{rootLabel}\, tr$ for the root label of tr. We often write a tree $(x, [tr_1, \ldots, tr_n])$ in a more compact, linear syntax:

- $x(tr_1, \ldots, tr_n)$, when $n \geq 1$, and
- x, when $n = 0$.

Thus $(2, [(4, [(3, [\,]), (1, [\,]), (6, [\,])]), (9, [\,])])$ can be written as $2(4(3, 1, 6), 9)$.

Consider the definition of $\textbf{Tree}\, X$ again: the set $\textbf{Tree}\, X$ of X-*trees* is the least set such that, (†) for all $x \in X$ and $trs \in \textbf{List}(\textbf{Tree}\, X)$, $(x, trs) \in \textbf{Tree}\, X$. Let's call a set U X-*closed* iff it satisfies property (†), where we have replaced $\textbf{Tree}\, X$ by U: for all $x \in X$ and $trs \in \textbf{List}\, U$, $(x, trs) \in U$. Thus the definition says that $\textbf{Tree}\, X$ is the least X-closed set, where we've yet to say just what "least" means.

First we address the concern that there might not be any X-closed sets. A easy result of set theory says that:

Lemma 1.3.1
For all sets X, there is a set U such that $X \subseteq U$, $U \times U \subseteq U$ and $\textbf{List}\, U \subseteq U$.

In other words, the lemma says that, given any set X, there exists a superset U of X such that every pair of elements of U is already an element of U, and every list of elements of U is already an element of U. Now, suppose X is a set, and let U be as in the lemma. To see that U is X-closed, suppose $x \in X$ and $trs \in \textbf{List}\, U$. Thus $x \in U$ and $trs \in U$, so that $(x, trs) \in U$, as required.

E.g., we know that there is an $\mathbb{Z}$-closed set U. But $\mathbb{N} \subseteq \mathbb{Z}$, and thus U is also $\mathbb{N}$-closed. But if $\textbf{Tree}\, \mathbb{N}$ turned out to be U, it would have elements like $(-5, [\,])$, which are not wanted.

To keep $\textbf{Tree}\, X$ from having junk, we say that $\textbf{Tree}\, X$ is the set U such that:

- U is X-closed, and
- for all X-closed sets V, $U \subseteq V$.

This is what we mean by saying that $\textbf{Tree}\, X$ is the *least* X-closed set. It is our first example of an *inductive definition*, the least (relative to $\subseteq$) set satisfying a given set of rules saying that if some elements are already in the set, then some other elements are also in the set.

To see that there is a unique least X-closed set, we first prove the following lemma.

Lemma 1.3.2
Suppose X is a set and $\mathcal{W}$ is a nonempty set of X-closed sets. Then $\bigcap\mathcal{W}$ is an X-closed set.

Proof. Suppose X is a set and $\mathcal{W}$ is a nonempty set of X-closed sets. Because $\mathcal{W}$ is nonempty, $\bigcap\mathcal{W}$ is well-defined. To see that $\bigcap\mathcal{W}$ is X-closed, suppose $x \in X$ and $\mathit{trs} \in \mathbf{List}\bigcap\mathcal{W}$. We must show that $(x, \mathit{trs}) \in \bigcap\mathcal{W}$, i.e., that $(x, \mathit{trs}) \in W$, for all $W \in \mathcal{W}$. Suppose $W \in \mathcal{W}$. We must show that $(x, \mathit{trs}) \in W$. Because $W \in \mathcal{W}$, we have that $\bigcap\mathcal{W} \subseteq W$, so that $\mathbf{List}\bigcap\mathcal{W} \subseteq \mathbf{List}\, W$. Thus, since $\mathit{trs} \in \mathbf{List}\bigcap\mathcal{W}$, it follows that $\mathit{trs} \in \mathbf{List}\, W$. But W is X-closed, and thus $(x, \mathit{trs}) \in W$, as required. □

As explained above, we have that there is an X-closed set, V. Let $\mathcal{W}$ be the set of all subsets of V that are X-closed. Thus $\mathcal{W}$ is a nonempty set of X-closed sets, since $V \in \mathcal{W}$. Let $U = \bigcap\mathcal{W}$. By Lemma 1.3.2, we have that U is X-closed. To see that $U \subseteq T$ for all X-closed sets T, suppose T is X-closed. By Lemma 1.3.2, we have that $V \cap T = \bigcap\{V, T\}$ is X-closed. And $V \cap T \subseteq V$, so that $V \cap T \in \mathcal{W}$. Hence $U = \bigcap\mathcal{W} \subseteq V \cap T \subseteq T$, as required. Finally, suppose that U' is also an X-closed set such that, for all X-closed sets T, $U' \subseteq T$. Then $U \subseteq U' \subseteq U$, showing that $U' = U$. Thus U is *the* least X-closed set.

Suppose X is a set, $x, x' \in X$ and $\mathit{trs}, \mathit{trs}' \in \mathbf{List}\,(\mathbf{Tree}\, X)$. It is easy to see that $(x, \mathit{trs}) = (x', \mathit{trs}')$ iff $x = x'$, $|\mathit{trs}| = |\mathit{trs}'|$ and, for all $i \in [1 : |\mathit{trs}|]$, $\mathit{trs}\, i = \mathit{trs}'\, i$.

Because trees are defined via an inductive definition, we get an induction principle for trees almost for free:

Theorem 1.3.3 (Principle of Induction on Trees)
Suppose X is a set and $P(\mathit{tr})$ is a property of an element $\mathit{tr} \in \mathbf{Tree}\, X$. If

for all $x \in X$ and $\mathit{trs} \in \mathbf{List}(\mathbf{Tree}\, X)$,
if (†) for all $i \in [1 : |\mathit{trs}|]$, $P(\mathit{trs}\, i)$,
then $P((x, \mathit{trs}))$,

then

for all $\mathit{tr} \in \mathbf{Tree}\, X$, $P(\mathit{tr})$.

We refer to (†) as the inductive hypothesis.

Proof. Suppose X is a set, $P(\mathit{tr})$ is a property of an element $\mathit{tr} \in \mathbf{Tree}\, X$, and

(‡) for all $x \in X$ and $\mathit{trs} \in \mathbf{List}(\mathbf{Tree}\, X)$,
if for all $i \in [1 : |\mathit{trs}|]$, $P(\mathit{trs}\, i)$,
then $P((x, \mathit{trs}))$.

We must show that

$$\text{for all } tr \in \mathbf{Tree}\,X,\ P(tr).$$

Let $U = \{\, tr \in \mathbf{Tree}\,X \mid P(tr)\,\}$. We will show that U is X-closed. Suppose $x \in X$ and $trs \in \mathbf{List}\,U$. We must show that $(x, trs) \in U$. It will suffice to show that $P((x, trs))$. By (‡), it will suffice to show that, for all $i \in [1 : |trs|]$, $P(trs\, i)$. Suppose $i \in [1 : |trs|]$. We must show that $P(trs\, i)$. Because $trs \in \mathbf{List}\,U$, we have that $trs\, i \in U$. Hence $P(trs\, i)$, as required.

Because U is X-closed, we have that $\mathbf{Tree}\,X \subseteq U$, as $\mathbf{Tree}\,X$ is the least X-closed set. Hence, for all $tr \in \mathbf{Tree}\,X$, $tr \in U$, so that, for all $tr \in \mathbf{Tree}\,X$, $P(tr)$. $\square$

Using our induction principle, we can now prove that every X-tree can be "destructed" into an element of X paired with a list of X-trees:

Proposition 1.3.4
Suppose X is a set. For all $tr \in \mathbf{Tree}\,X$, there are $x \in X$ and $trs \in \mathbf{List}(\mathbf{Tree}\,X)$ such that $tr = (x, trs)$.

Proof. Suppose X is a set. We use induction on trees to prove that, for all $tr \in \mathbf{Tree}\,X$, there are $x \in X$ and $trs \in \mathbf{List}(\mathbf{Tree}\,X)$ such that $tr = (x, trs)$. Suppose $x \in X$, $trs \in \mathbf{List}(\mathbf{Tree}\,X)$, and assume the inductive hypothesis: for all $i \in [1 : |trs|]$, there are $x' \in X$ and $trs' \in \mathbf{List}(\mathbf{Tree}\,X)$ such that $trs\, i = (x', trs')$. We must show that there are $x' \in X$ and $trs' \in \mathbf{List}(\mathbf{Tree}\,X)$ such that $(x, trs) = (x', trs')$. And this holds, since $x \in X$, $trs \in \mathbf{List}(\mathbf{Tree}\,X)$ and $(x, trs) = (x, trs)$. $\square$

Note that the preceding induction makes no use of its inductive hypothesis, and yet the induction is still necessary.

Suppose X is a set. Let the predecessor relation $\mathbf{pred}_{\mathbf{Tree}\,X}$ on $\mathbf{Tree}\,X$ be the set of all pairs of X-trees (tr, tr') such that there are $x \in X$ and $trs' \in \mathbf{List}(\mathbf{Tree}\,X)$ such that $tr' = (x, trs')$ and $trs'\, i = tr$ for some $i \in [1 : |trs'|]$, i.e., such that tr is one of the children of tr'. Thus the predecessors of a tree $(x, [tr_1, \ldots, tr_n])$ are its children $tr_1, \ldots, tr_n$.

Proposition 1.3.5
If X is a set, then $\mathbf{pred}_{\mathbf{Tree}\,X}$ is a well-founded relation on $\mathbf{Tree}\,X$.

Proof. Suppose X is a set and Y is a nonempty subset of $\mathbf{Tree}\,X$. Mimicking Proposition 1.2.5, we can use the principle of induction on trees to prove that, for all $tr \in \mathbf{Tree}\,X$, if $tr \in Y$, then Y has a $\mathbf{pred}_{\mathbf{Tree}\,X}$-minimal element. Because Y is nonempty, we can conclude that Y has a $\mathbf{pred}_{\mathbf{Tree}\,X}$-minimal element. $\square$

Exercise 1.3.6
Do the induction on trees used by the preceding proof.

1.3.2 Recursion

Suppose R is a well-founded relation on a set A. We can define a function f from A to a set B by *well-founded recursion on* R. The idea is simple: when f is called with an element $x \in A$, it may call itself recursively on any of the predecessors of x in R. Typically, such a definition will be concrete enough that we can regard it as defining an algorithm as well as a function.

If R is a well-founded relation on a set A, and B is a set, then we write $\mathbf{Rec}_{A,R,B}$ for

$$\{\,(x, f) \mid x \in A \text{ and } f \in \{\,y \in A \mid y\,R\,x\,\} \to B\,\} \to B.$$

An element F of $\mathbf{Rec}_{A,R,B}$ may only be called with a pair (x, f) such that $x \in A$ and f is a function from the predecessors of x in R to B. Intutively, F's job is to transform x into an element of B, using f to carry out recursive calls.

Theorem 1.3.7 (Well-Founded Recursion)
Suppose R is a well-founded relation on a set A, B is a set, and $F \in \mathbf{Rec}_{A,R,B}$. There is a unique $f \in A \to B$ such that, for all $x \in A$,

$$f\,x = F(x, f|\{\,y \in A \mid y\,R\,x\,\}).$$

The second argument to F in the definition of f is the restriction of f to the predecessors of x in R, i.e., it's the subset of f whose domain is $\{\,y \in A \mid y\,R\,x\,\}$.

If we can understand F as an algorithm, then we can understand the definition of f as an algorithm. If we call f with an $x \in A$, then F may return an element of B without consulting its second argument. Alternatively, it may call this second argument with a predecessor of x in R. This is a recursive call of f, which must complete before F continues. Once it does complete, F may make more recursive calls, but must eventually return an element of B.

Proof. We start with an inductive definition: let f be the least subset of $A \times B$ such that, for all $x \in A$ and $g \in \{\,y \in A \mid y\,R\,x\,\} \to B$,

$$\text{if } g \subseteq f, \text{ then } (x, F(x, g)) \in f.$$

We say that a subset U of $A \times B$ is *closed* iff, for all $x \in A$ and $g \in \{\,y \in A \mid y\,R\,x\,\} \to B$,

$$\text{if } g \subseteq U, \text{ then } (x, F(x, g)) \in U.$$

Thus, we are saying that f is the least closed subset of $A \times B$. This definition is well-defined because $A \times B$ is closed, and if $\mathcal{W}$ is a nonempty set of closed subsets of $A \times B$, then $\bigcap \mathcal{W}$ is also closed. Thus we can let f be the intersection of all closed subsets of $A \times B$.

Thus f is a relation from A to B. An easy well-founded induction on R suffices to show that, for all $x \in A$, $x \in \mathbf{domain}\, f$. Suppose $x \in A$, and assume the inductive hypothesis: for all $y \in A$, if $y \mathrel{R} x$, then $y \in \mathbf{domain}\, f$. We must show that $x \in \mathbf{domain}\, f$. By the inductive hypothesis, we have that for all $y \in \{\, y \in A \mid y \mathrel{R} x \,\}$, there is a $z \in B$ such that $(y, z) \in f$. Thus there is a subset g of f such that $g \in \{\, y \in A \mid y \mathrel{R} x \,\} \to B$. (Since we don't know at this point that f is a function, we are using the Axiom of Choice in this last step.) Hence $(x, F(x, g)) \in f$, showing that $x \in \mathbf{domain}\, f$. Thus $\mathbf{domain}\, f = A$.

Next we show that f is a function. Let h be

$$\{\, (x, y) \in f \mid \text{for all } y' \in B, \text{ if } (x, y') \in f, \text{ then } y = y' \,\}.$$

If we can show that h is closed, then we will have that $f \subseteq h$, because f is the least closed set, and thus we'll be able to conclude that f is a function. To show that h is closed, suppose $x \in A$, $g \in \{\, y \in A \mid y \mathrel{R} x \,\} \to B$ and $g \subseteq h$. We must show that $(x, F(x, g)) \in h$. It will suffice to show that, for all $y' \in B$, if $(x, y') \in f$, then $F(x, g) = y'$. Suppose $y' \in B$ and $(x, y') \in f$. We must show that $F(x, g) = y'$. Because $(x, y') \in f$, and f is the least closed subset of $A \times B$, there must be a $g' \in \{\, y \in A \mid y \mathrel{R} x \,\} \to B$ such that $g' \subseteq f$ and $y' = F(x, g')$. Thus it will suffice to show that $F(x, g) = F(x, g')$, which will follow from showing that $g = g'$, i.e., for all $z \in \{\, y \in A \mid y \mathrel{R} x \,\}$, $g\, z = g'\, z$. Suppose $z \in \{\, y \in A \mid y \mathrel{R} x \,\}$. We must show that $g\, z = g'\, z$. Since $z \in \{\, y \in A \mid y \mathrel{R} x \,\}$, we have that $z \in A$ and $z \mathrel{R} x$. Because $(z, g\, z) \in g \subseteq h$, we have that $(z, g\, z) \in h$. Since $(z, g'\, z) \in g' \subseteq f$, we have that $(z, g'\, z) \in f$. Hence, by the definition of h, we have that $g\, z = g'\, z$, as required.

Summarizing, we know that $f \in A \to B$. Next, we must show that, for all $x \in A$,

$$f\, x = F(x, f|\{\, y \in A \mid y \mathrel{R} x \,\}).$$

Suppose $x \in A$. Because $f|\{\, y \in A \mid y \mathrel{R} x \,\} \in \{\, y \in A \mid y \mathrel{R} x \,\} \to B$, we have that $(x, F(x, f|\{\, y \in A \mid y \mathrel{R} x \,\})) \in f$, so that $f\, x = F(x, f|\{\, y \in A \mid y \mathrel{R} x \,\})$.

Finally, suppose that $f' \in A \to B$ and for all $x \in A$,

$$f'\, x = F(x, f'|\{\, y \in A \mid y \mathrel{R} x \,\}).$$

To see that $f = f'$, it will suffice to show that, for all $x \in A$, $f\, x = f'\, x$. We proceed by well-founded induction on R. Suppose $x \in A$ and assume the inductive hypothesis: for all $y \in A$, if $y \mathrel{R} x$, then $f\, y = f'\, y$. We must show that $f\, x = f'\, x$. By the inductive hypothesis, we have that $f|\{\, y \in A \mid y \mathrel{R} x \,\} = f'|\{\, y \in A \mid y \mathrel{R} x \,\}$. Thus

$$\begin{aligned} f\, x &= F(x, f|\{\, y \in A \mid y \mathrel{R} x \,\}) \\ &= F(x, f'|\{\, y \in A \mid y \mathrel{R} x \,\}) \\ &= f'\, x, \end{aligned}$$

as required. □

Here are some examples of well-founded recursion:

- If we define $f \in \mathbb{N} \rightarrow B$ by well-founded recursion on $<$, then, when f is called with $n \in \mathbb{N}$, it may call itself recursively on any strictly smaller natural numbers. In the case $n = 0$, it can't make any recursive calls.

- If we define $f \in \mathbb{N} \rightarrow B$ by well-founded recursion on the predecessor relation $\mathbf{pred}_{\mathbb{N}}$, then when f is called with $n \in \mathbb{N}$, it may call itself recursively on $n - 1$, in the case when $n \geq 1$, and may make no recursive calls, when $n = 0$.

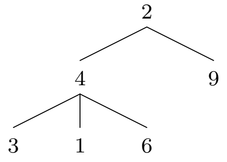
  Thus, if such a definition case-splits according to whether its input is 0 or not, it can be split into two parts:

  - $f\,0 = \cdots;$
  - for all $n \in \mathbb{N}$, $f(n+1) = \cdots f\,n \cdots.$

  We say that such a definition is by *recursion on* $\mathbb{N}$.

- If we define $f \in \mathbf{Tree}\,X \rightarrow B$ by well-founded recursion on the predecessor relation $\mathbf{pred}_{\mathbf{Tree}\,X}$, then when f is called on an X-tree $(x, [tr_1, \ldots, tr_n])$, it may call itself recursively on any of tr_1, ..., tr_n. When $n = 0$, it may make no recursive calls. We say that such a definition is by *structural recursion.*

- We may define the *size* of an X-tree $(x, [tr_1, \ldots, tr_n])$ by summing the recursively computed sizes of tr_1, ..., tr_n, and then adding 1. (When $n = 0$, the sum of no sizes is 0, and so we get 1.) Then, e.g., the size of

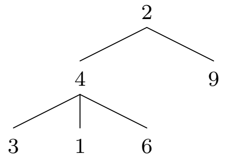


  is 6. This defines a function $\mathbf{size} \in \mathbf{Tree}\,X \rightarrow \mathbb{N}$.

- We may define the *number of leaves* of an X-tree $(x, [tr_1, \ldots, tr_n])$ as

  - 1, when $n = 0$, and
  - the sum of the recursively computed numbers of leaves of tr_1, ..., tr_n, when $n \geq 1$.

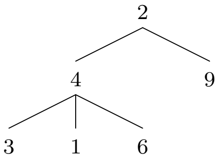
  Then, e.g., the number of leaves of



is 4. This defines a function $\textbf{numLeaves} \in \textbf{Tree}\,X \to \mathbb{N}$.

- We may define the *height* of an X-tree $(x, [tr_1, \ldots, tr_n])$ as
  - 0, when $n = 0$, and
  - 1 plus the maximum of the recursively computed heights of $tr_1, \ldots, tr_n$, when $n \geq 1$.

  E.g., the height of

  is 2. This defines a function $\textbf{height} \in \textbf{Tree}\,X \to \mathbb{N}$.

- Given a set X, we can define a well-founded relation $\textbf{size}_{\textbf{Tree}\,X}$ on $\textbf{Tree}\,X$ by: for all $tr, tr' \in \textbf{Tree}\,X$, $tr\ \textbf{size}_{\textbf{Tree}\,X}\ tr'$ iff $\textbf{size}\ tr < \textbf{size}\,tr'$. (This is an application of Proposition 1.2.10.)

  If we define a function $f \in \textbf{Tree}\,X \to B$ by well-founded recursion on $\textbf{size}_{\textbf{Tree}\,X}$, when f is called with an X-tree tr, it may call itself recursively on any X-trees with strictly smaller sizes.

- Given a set X, we can define a well-founded relation $\textbf{height}_{\textbf{Tree}\,X}$ on $\textbf{Tree}\,X$ by: for all $tr, tr' \in \textbf{Tree}\,X$, $tr\ \textbf{height}_{\textbf{Tree}\,X}\ tr'$ iff $\textbf{height}\ tr < \textbf{height}\,tr'$.

  If we define a function $f \in \textbf{Tree}\,X \to B$ by well-founded recursion on $\textbf{height}_{\textbf{Tree}\,X}$, when f is called with an X-tree tr, it may call itself recursively on any X-trees with strictly smaller heights.

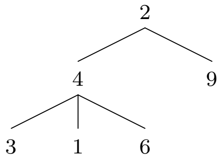
- Given a set X, we can define a well-founded relation $\textbf{length}_{\textbf{List}\,X}$ on $\textbf{List}\,X$ by: for all $xs, ys \in \textbf{List}\,X$, $xs\ \textbf{length}_{\textbf{List}\,X}\ ys$ iff $|xs| < |ys|$.

  If we define a function $f \in \textbf{List}\,X \to B$ by well-founded recursion on $\textbf{length}_{\textbf{List}\,X}$, when f is called with an X-list xs, it may call itself recursively on any X-lists with strictly smaller lengths.

1.3.3 Paths in Trees

We can think of an $\mathbb{N} - \{0\}$-list $[n_1, n_2, \ldots, n_m]$ as a *path* through an X-tree tr: one starts with tr itself, goes to the n_1-th child of tr, selects the n_2-th child of that tree, etc., stopping when the list is exhausted.

Consider the $\mathbb{N}$-tree



Then:

- $[\,]$ takes us to the whole tree.
- $[1]$ takes us to the tree $4(3,1,6)$.
- $[1,3]$ takes us to the tree 6.
- $[1,4]$ takes us to no tree.

We define the valid paths of an X-tree via structural recursion. For a set X, we define $\mathbf{validPaths}_X \in \mathbf{Tree}\,X \to \mathbf{List}(\mathbb{N}-\{0\})$ (we often drop the subscript X, when it's clear from the context) by: for all $x \in X$ and $\mathit{trs} \in \mathbf{List}(\mathbf{Tree}\,X)$, $\mathbf{validPaths}(x, \mathit{trs})$ is

$$\{[\,]\} \cup \{\, [i] \mathbin{@} \mathit{xs} \mid i \in [1:|\mathit{trs}|] \text{ and } \mathit{xs} \in \mathbf{validPaths}(\mathit{trs}\, i)\,\}.$$

We say that $\mathit{xs} \in \mathbf{List}(\mathbb{N}-\{0\})$ *is a valid path for* an X-tree tr iff $\mathit{xs} \in \mathbf{validPaths}\,\mathit{tr}$. For example, $\mathbf{validPaths}(3(4,1(7),6)) = \{[\,],[1],[2],[2,1],[3]\}$. Thus, e.g., $[2,1]$ is a valid path for $3(4,1(7),6)$, whereas $[2,2]$ and $[4]$ are not valid paths for this tree.

Now, we define a function **select** that takes in an X-tree tr and a valid path xs for tr, and returns the subtree of tr pointed to by xs. Let Y_X be

$$\{\, (\mathit{tr}, \mathit{xs}) \in \mathbf{Tree}\,X \times \mathbf{List}(\mathbb{N}-\{0\}) \mid \mathit{xs} \text{ is a valid path for } \mathit{tr}\,\}.$$

Let the relation R on Y_X be

$$\{\, ((\mathit{tr}, \mathit{xs}), (\mathit{tr}', \mathit{xs}')) \in Y_X \times Y_X \mid |\mathit{xs}| < |\mathit{xs}'|\,\}.$$

By Proposition 1.2.10, we have that R is well-founded, and so we can use well-founded recursion on R to define a function $\mathbf{select}_X$ (we often drop the subscript X) from Y_X to $\mathbf{Tree}\,X$. Suppose, we are given an input $((x, \mathit{trs}), \mathit{xs}) \in Y$. If xs is $[\,]$, then we return (x, trs). Otherwise, $\mathit{xs} = [i] \mathbin{@} \mathit{xs}'$, for some $i \in \mathbb{N}-\{0\}$ and $\mathit{xs}' \in \mathbf{List}(\mathbb{N}-\{0\})$. Because $((x, \mathit{trs}), \mathit{xs}) \in Y$, it follows that $i \in [1:|\mathit{trs}|]$ and xs' is a valid path for $\mathit{trs}\, i$. Thus $(\mathit{trs}\, i, \mathit{xs}')$ is in Y, and is a predecessor of $((x, \mathit{trs}), \mathit{xs})$ in R, so that we can call ourselves recursively on $(\mathit{trs}\, i, \mathit{xs}')$ and return the resulting tree. For example $\mathbf{select}(4(3(2,1(7)),6),[\,]) = 4(3(2,1(7)),6)$ and $\mathbf{select}(4(3(2,1(7)),6),[1,2]) = 1(7)$.

We say that an X-tree tr' *is a subtree of* an X-tree tr iff there is a valid path xs for tr such that $\mathit{tr}' = \mathbf{select}(\mathit{tr}, \mathit{xs})$. And tr' *is a leaf of* tr iff tr' is a subtree of tr and tr' has no children.

Finally, we can define a function that takes in a X-tree tr, a valid path xs for tr, and an X-tree tr', and returns the result of replacing the subtree of tr pointed to by xs with tr'. Let Y_X be

$$\{\,(tr, xs, tr') \in \mathbf{Tree}\,X \times \mathbf{List}(\mathbb{N} - \{0\}) \times \mathbf{Tree}\,X \mid xs \text{ is a valid path for } tr\,\}.$$

We use well-founded recursion on the size of the second components (the paths) of the elements of Y_X to define a function $\mathbf{update}_X$ (we often drop the subscript) from Y_X to $\mathbf{Tree}\,X$. Suppose, we are given an input $((x, trs), xs, tr') \in Y$. If xs is $[\,]$, then we return tr'. Otherwise, $xs = [i] \mathbin{@} xs'$, for some $i \in \mathbb{N} - \{0\}$ and $xs' \in \mathbf{List}(\mathbb{N} - \{0\})$. Because $((x, trs), xs, tr') \in Y$, it follows that $i \in [1 : |trs|]$ and xs' is a valid path for $trs\,i$. Thus $(trs\,i, xs', tr')$ is in Y, and $|xs'| < |xs|$. Hence, we can let tr'' be the result of calling ourselves recursively on $(trs\,i, xs', tr')$. Finally, we can return (x, trs'), where $trs' = trs[i \mapsto tr'']$ (which is the same as trs, excepts that its ith element is tr''. For example $\mathbf{update}(4(3(2, 1(7)), 6), [\,], 3(7, 8)) = 3(7, 8)$ and $\mathbf{update}(4(3(2, 1(7)), 6), [1, 2], 3(7, 8)) = 4(3(2, 3(7, 8)), 6)$.

1.3.4 Notes

Our treatment of trees, inductive definition, and well-founded recursion is more formal than what one finds in typical books on formal language theory. But those with a background in set theory will find nothing surprising in this section, and our foundational approach will serve the reader well in later chapters.

Chapter 2

Formal Languages

In this chapter we say what symbols, strings, alphabets and (formal) languages are, show how to use various induction principles to prove language equalities, and give an introduction to the Forlan toolset. In subsequent chapters, we will study four more restricted kinds of languages: the regular (Chapter 3), context-free (Chapter **??**), recursive and recursively enumerable (Chapter **??**) languages.

2.1 Symbols, Strings, Alphabets and (Formal) Languages

In this section, we define the basic notions of the subject: symbols, strings, alphabets and (formal) languages. In most presentations of formal language theory, the "symbols" that make up strings are allowed to be arbitrary elements of the mathematical universe. This is convenient in some ways, but it means that, e.g., the collection of all strings is too "big" to be a set. Furthermore, if we were to adopt this convention, we wouldn't be able to have notation in Forlan for all strings and symbols. These considerations lead us to the following definition.

2.1.1 Symbols

The set **Char** of *symbol characters* consists of the following 65 elements:

- the comma (",");
- the *digits* $\mathsf{0}$–$\mathsf{9}$;
- the *letters* a–z and A–Z; and
- the angle brackets ("$\langle$" and "$\rangle$").

We order **Char** as follows:

$$, < \mathsf{0} < \cdots < \mathsf{9} < \mathsf{a} < \cdots \mathsf{z} < \mathsf{A} < \cdots \mathsf{Z} < \langle < \rangle.$$

The set **Sym** of *symbols* is the least subset of **List Char** such that:

- for all digits and letters c, $[c] \in \mathbf{Sym}$; and
- for all $n \in \mathbb{N}$ and $x_1, \ldots, x_n \in \{[\,,]\} \cup \mathbf{Sym}$,

$$[\,\langle\,] \mathbin{@} x_1 \mathbin{@} \cdots \mathbin{@} x_n \mathbin{@} [\,\rangle\,] \in \mathbf{Sym}.$$

This is an inductive definition (see Section 1.3). **Sym** consists of just those lists of symbol characters that can be built using the above, two rules. For example, $[\mathsf{9}]$, $[\,\langle, \rangle]$, $[\,\langle, \mathsf{i}, \mathsf{d}, \rangle]$ and $[\,\langle, \langle, \mathsf{a}, \mathsf{,}, \rangle, \mathsf{b}, \rangle]$ are symbols. On the other hand, $[\,\langle, \rangle, \rangle]$ is not a symbol.

We can prove by induction that, for all $z \in \mathbf{Sym}$, for all $x, y \in \mathbf{List\,Char}$, if $z = x \mathbin{@} y \in \mathbf{Sym}$, then:

- if $x \in \mathbf{Sym}$, then $y = [\,]$;
- if $y \in \mathbf{Sym}$, then $x = [\,]$.

Thus a symbol never starts or ends with another symbol.

We normally abbreviate a symbol $[c_1, \ldots, c_n]$ to $c_1 \cdots c_n$, so that $\mathsf{9}$, $\langle\,\rangle$, $\langle\mathsf{id}\rangle$ and $\langle\langle\mathsf{a},\rangle\mathsf{b}\rangle$ are symbols. And if x and y are elements of **List Char**, we typically abbreviate $x \mathbin{@} y$ to xy.

Whenever possible, we will use the mathematical variables a, b and c to name symbols. We order **Sym** first by length, and then lexicographically (in dictionary order). So, we have that

$$\mathsf{0} < \cdots < \mathsf{9} < \mathsf{A} < \cdots < \mathsf{Z} < \mathsf{a} < \cdots < \mathsf{z},$$

and, e.g.,

$$\mathsf{z} < \langle\mathsf{be}\rangle < \langle\mathsf{by}\rangle < \langle\mathsf{on}\rangle < \langle\mathsf{can}\rangle < \langle\mathsf{con}\rangle.$$

Obviously, **Sym** is infinite, but is it countably infinite? The answer is "yes", because we can enumerate the symbols in order.

2.1.2 Strings

A *string* is a list of symbols. Whenever possible, we will use the mathematical variables u, v, w, x, y and z to name strings.

We typically abbreviate the empty string $[\,]$ to %, and abbreviate $[a_1, \ldots, a_n]$ to $a_1 \cdots a_n$, when $n \geq 1$. For example $[\mathsf{0}, \langle\mathsf{0}\rangle, \mathsf{1}, \langle\langle,\rangle\rangle]$ is abbreviated to $\mathsf{0}\langle\mathsf{0}\rangle\mathsf{1}\langle\langle,\rangle\rangle$. We name the empty string by %, instead of following convention and using ϵ, since this symbol can also be used in Forlan.

We write **Str** for **List Sym**, the set of all strings. We order **Str** first by length and then lexicographically, using our order on **Sym**. Thus, e.g.,

$$\% < \mathsf{ab} < \mathsf{a}\langle\mathsf{be}\rangle < \mathsf{a}\langle\mathsf{by}\rangle < \langle\mathsf{can}\rangle\langle\mathsf{be}\rangle < \mathsf{abc}.$$

Since every string can be written as a finite sequence of ASCII characters, it follows that **Str** is countably infinite.

Because strings are lists, we have that $|x|$ is the *length* of a string x, and that $x \mathbin{@} y$ is the *concatenation* of strings x and y. We typically abbreviate $x \mathbin{@} y$ to xy. For example:

- $|\%| = |[\,]| = 0$;
- $|\mathsf{0}\langle\mathsf{0}\rangle\mathsf{1}\langle\langle,\rangle\rangle| = |[\mathsf{0}, \langle\mathsf{0}\rangle, \mathsf{1}, \langle\langle,\rangle\rangle]| = 4$; and
- $(\mathsf{01})(\mathsf{00}) = [\mathsf{0}, \mathsf{1}] \mathbin{@} [\mathsf{0}, \mathsf{0}] = [\mathsf{0}, \mathsf{1}, \mathsf{0}, \mathsf{0}] = \mathsf{0100}$.

From our study of lists, we know that:

- Concatenation is associative: for all $x, y, z \in \mathbf{Str}$,
$$(xy)z = x(yz).$$
- % is the identify for concatenation: for all $x \in \mathbf{Str}$,
$$\%x = x = x\%.$$

It is easy to see that, for all $x, y, x', y' \in \mathbf{Str}$:

- $xy = xy'$ iff $y = y'$; and
- $xy = x'y$ iff $x = x'$.

We define the string x^n *formed by raising* a string x *to the power* $n \in \mathbb{N}$ by recursion on n:

$$x^0 = \%, \text{ for all } x \in \mathbf{Str}; \text{ and}$$
$$x^{n+1} = xx^n, \text{ for all } x \in \mathbf{Str} \text{ and } n \in \mathbb{N}.$$

We assign this operation higher precedence than concatenation, so that xx^n means $x(x^n)$ in the above definition. For example, we have that

$$(\mathsf{ab})^2 = (\mathsf{ab})(\mathsf{ab})^1 = (\mathsf{ab})(\mathsf{ab})(\mathsf{ab})^0 = (\mathsf{ab})(\mathsf{ab})\% = \mathsf{abab}.$$

Proposition 2.1.1
For all $x \in \mathbf{Str}$ and $n, m \in \mathbb{N}$, $x^{n+m} = x^n x^m$.

Proof. Suppose $x \in \mathbf{Str}$ and $m \in \mathbb{N}$. We use mathematical induction to show that, for all $n \in \mathbb{N}$, $x^{n+m} = x^n x^m$.

(Basis Step) We have that $x^{0+m} = x^m = \%x^m = x^0 x^m$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $x^{n+m} = x^n x^m$. We must show that $x^{(n+1)+m} = x^{n+1}x^m$. We have that

$$\begin{aligned} x^{(n+1)+m} &= x^{(n+m)+1} \\ &= xx^{n+m} && \text{(definition of } x^{(n+m)+1}\text{)} \\ &= x(x^n x^m) && \text{(inductive hypothesis)} \\ &= (xx^n)x^m \\ &= x^{n+1}x^m && \text{(definition of } x^{n+1}\text{).} \end{aligned}$$

□

Thus, if $x \in \mathbf{Str}$ and $n \in \mathbb{N}$, then

$$x^{n+1} = xx^n \qquad \text{(definition),}$$

and

$$x^{n+1} = x^n x^1 = x^n x \qquad \text{(Proposition 2.1.1).}$$

Next, we consider the prefix, suffix and substring relations on strings. Suppose x and y are strings. We say that:

- x is a *prefix* of y iff $y = xv$ for some $v \in \mathbf{Str}$;
- x is a *suffix* of y iff $y = ux$ for some $u \in \mathbf{Str}$; and
- x is a *substring* of y iff $y = uxv$ for some $u, v \in \mathbf{Str}$.

In other words, x is a prefix of y iff x is an initial part of y, x is a suffix of y iff x is a trailing part of y, and x is a substring of y iff x appears in the middle of y. But note that the strings u and v can be empty in these definitions. Thus, e.g., a string x is always a prefix of itself, since $x = x\%$. A prefix, suffix or substring of a string other than the string itself is called *proper*.

For example:

- % is a proper prefix, suffix and substring of ab;
- a is a proper prefix and substring of ab;
- b is a proper suffix and substring of ab; and
- ab is a (non-proper) prefix, suffix and substring of ab.

Proposition 2.1.2
For all $x, y, x', y' \in \mathbf{Str}$, $xy = x'y'$ iff either

- *$xu = x'$ and $y = uy'$, for some $u \in \mathbf{Str}$, or*

- *$x'u = x$ and $y' = uy$, for some $u \in$ **Str**.*

Proof. Straightforward. □

As a consequence of this proposition, we have that:

- For all $x, x', y' \in$ **Str**, x is a prefix of $x'y'$ iff either
  - x is a prefix of x', or
  - $x = x'u$, for some prefix u of y'.
- For all $x, x', y' \in$ **Str**, x is a suffix of $x'y'$ iff either
  - x is a suffix of y', or
  - $x = uy'$, for some suffix u of x'.
- For all $x, x', y' \in$ **Str**, x is a substring of $x'y'$ iff either
  - x is a substring of x', or
  - $x = uv$, for some $u, v \in$ **Str** such that u is a suffix of x' and v is a prefix of y', or
  - x is a substring of y'.

2.1.3 Alphabets

Having said what symbols and strings are, we now come to alphabets. An *alphabet* is a finite subset of **Sym**. We use Σ (upper case Greek letter sigma) to name alphabets. For example, $\emptyset$, $\{\mathsf{0}\}$ and $\{\mathsf{0}, \mathsf{1}\}$ are alphabets. We write **Alp** for the set of all alphabets. **Alp** is countably infinite.

We define **alphabet** $\in$ **Str** $\rightarrow$ **Alp** by recursion on (the length of) strings:

$$\begin{aligned} \textbf{alphabet}\,\% &= \emptyset, \\ \textbf{alphabet}(ax) &= \{a\} \cup \textbf{alphabet}\,x, \text{ for all } a \in \textbf{Sym} \text{ and } x \in \textbf{Str}. \end{aligned}$$

I.e., **alphabet** w consists of all of the symbols occurring in the string w. E.g., $\textbf{alphabet}(\mathsf{01101}) = \{\mathsf{0}, \mathsf{1}\}$. Because the string x appears on the right side of ax in the rule $\textbf{alphabet}(ax) = \{a\} \cup \textbf{alphabet}\,x$, we call this *right* recursion. (Since $\cup$ is associative and commutative, it would have been equivalent to use *left* recursion, $\textbf{alphabet}(xa) = \{a\} \cup \textbf{alphabet}\,x$.) We say that **alphabet** x is the *alphabet of* x.

If Σ is an alphabet, then we write Σ^* for **List** Σ. I.e., Σ^* consists of all of the strings that can be built using the symbols of Σ. For example, the elements of $\{\mathsf{0}, \mathsf{1}\}^* = \textbf{List}\,\{\mathsf{0}, \mathsf{1}\}$ are:

$$\%, \mathsf{0}, \mathsf{1}, \mathsf{00}, \mathsf{01}, \mathsf{10}, \mathsf{11}, \mathsf{000}, \ldots$$

2.1.4 Languages

We say that L is a *formal language* (or just *language*) iff $L \subseteq \Sigma^*$, for some $\Sigma \in \mathbf{Alp}$. In other words, a language is a set of strings over some alphabet. If $\Sigma \in \mathbf{Alp}$, then we say that L is a Σ-*language* iff $L \subseteq \Sigma^*$.

Here are some example languages (all are $\{\mathsf{0}, \mathsf{1}\}$-languages):

- $\emptyset$;
- $\{\mathsf{0}, \mathsf{1}\}^*$;
- $\{\mathsf{010}, \mathsf{1001}, \mathsf{1101}\}$;
- $\{\, \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \,\} = \{\mathsf{0}^0\mathsf{1}^0, \mathsf{0}^1\mathsf{1}^1, \mathsf{0}^2\mathsf{1}^2, \ldots\} = \{\%, \mathsf{01}, \mathsf{0011}, \ldots\}$; and
- $\{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid w \text{ is a palindrome} \,\}$.

(A *palindrome* is a string that reads the same backwards and forwards, i.e., that is equal to its own reversal.) On the other hand, the set of strings $X = \{\langle\rangle, \langle\mathsf{0}\rangle, \langle\mathsf{00}\rangle, \ldots\}$, is not a language, since it involves infinitely many symbols, i.e., since there is no alphabet Σ such that $X \subseteq \Sigma^*$.

Since **Str** is countably infinite and every language is a subset of **Str**, it follows that every language is countable. Furthermore, Σ^* is countably infinite, as long as the alphabet Σ is nonempty ($\emptyset^* = \{\%\}$).

We write **Lan** for the set of all languages. It turns out that **Lan** is uncountable. In fact even $\mathcal{P}(\{\mathsf{0}\}^*)$, the set of all $\{\mathsf{0}\}$-languages, has the same size as $\mathcal{P}(\mathbb{N})$, and is thus uncountable.

Exercise 2.1.3
Show that $\mathcal{P}(\mathbb{N})$ has the same size as $\mathcal{P}(\{\mathsf{0}\}^*)$.

Given a language L, we write **alphabet** L for the *alphabet*

$$\bigcup\{\, \mathbf{alphabet}\, w \mid w \in L \,\}.$$

of L. I.e., **alphabet** L consists of all of the symbols occurring in the strings of L. For example,

$$\begin{aligned}\mathbf{alphabet}\,\{\mathsf{011}, \mathsf{112}\} &= \bigcup\{\mathbf{alphabet}(\mathsf{011}), \mathbf{alphabet}(\mathsf{112})\}\\ &= \bigcup\{\{\mathsf{0}, \mathsf{1}\}, \{\mathsf{1}, \mathsf{2}\}\} = \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}.\end{aligned}$$

Note that, for all languages L, $L \subseteq (\mathbf{alphabet}\, L)^*$.

If A is an infinite subset of **Sym** (and so is not an alphabet), we allow ourselves to write A^* for **List** A. I.e., A^* consists of all of the strings that can be built using the symbols of A. For example, $\mathbf{Sym}^* = \mathbf{Str}$.

2.1.5 Notes

In a traditional approach to the subject, symbols may be anything, real numbers, sets, etc. But such a choice would mean that not all symbols could be expressed in Forlan's syntax, and would needlessly complicate the set theoretic foundations of the subject. By working with a fixed, countably infinite set of symbols, all symbols can be expressed in Forlan, and we have that that strings, regular expressions, etc., are sets, not set-indexed families of sets.

Representing strings as lists of symbols, which in turn are represented as functions, is nontraditional, but should seem a natural approach to those with a background in set theory or functional programming.

2.2 Using Induction to Prove Language Equalities

In this section, we introduce three string induction principles, ways of showing that every $w \in A^*$ has property $P(w)$, where A is some set of symbols. Typically, A will be an alphabet, i.e., a finite set of symbols. But when we want to prove that all strings have some property, we can let $A = \mathbf{Sym}$, so that $A^* = \mathbf{Str}$. Each of these principles corresponds to an instance of well-founded induction. We also look at how different kinds of induction can be used to show that two languages are equal.

2.2.1 String Induction Principles

Suppose A is a set of symbols. We define well-founded relations $\mathbf{right}_A$, $\mathbf{left}_A$ and $\mathbf{strong}_A$ on A^* by:

- $x \ \mathbf{right}_A \ y$ iff $y = ax$ for some $a \in A$ (it's called *right* because the string x is on the right side of ax);
- $x \ \mathbf{left}_A \ y$ iff $y = xa$ for some $a \in A$ (it's called *left* because the string x is on the left side of xa);
- $x \ \mathbf{strong}_A \ y$ iff x is a proper substring of y.

Thus, for all $a \in A$ and $x \in A^*$, the only predecessor of ax in $\mathbf{right}_A$ is x, and the only predecessor of xa in $\mathbf{left}_A$ is x. And, for all $y \in A^*$, the predecessors of y in $\mathbf{strong}_A$ are the proper substrings of y. The empty string, %, has no predecessors in any of these relations.

The well-foundedness of $\mathbf{right}_A$, $\mathbf{left}_A$ and $\mathbf{strong}_A$ follows by Proposition 1.2.9, since each of these relations is a subset of $\mathbf{length}_{\mathbf{List}\,A}$, which is a well-founded relation on $\mathbf{List}\,A$.

We can do well-founded induction and recursion on these relations. In fact, what we called right and left recursion on strings in Section 2.1 correspond to recursion on $\mathbf{right}_{\mathbf{Sym}}$ and $\mathbf{left}_{\mathbf{Sym}}$.

We now introduce string induction principles corresponding to well-founded induction on each of the above relations.

Theorem 2.2.1 (Principle of Right String Induction)
Suppose $A \subseteq \mathbf{Sym}$ and $P(w)$ is a property of a string w. If

(basis step)

$$P(\%) \textit{ and}$$

(inductive step)

$$\textit{for all } a \in A \textit{ and } w \in A^*,\ \textit{if}\ \ (\dagger)\ P(w),\ \textit{then}\ P(aw),$$

then,

$$\textit{for all } w \in A^*,\ P(w).$$

We refer to the formula (†) as the *inductive hypothesis*. According to the induction principle, to show that every $w \in A^*$ has property P, we show that the empty string has property P, and then assume that $a \in A$, $w \in A^*$ and that (the inductive hypothesis) w has property P, and show that aw has property P.

Proof. Equivalent to well-founded induction on $\mathbf{right}_A$. □

By switching aw to wa in the inductive step, we get the principle of left string induction.

Theorem 2.2.2 (Principle of Left String Induction)
Suppose $A \subseteq \mathbf{Sym}$ and $P(w)$ is a property of a string w. If

(basis step)

$$P(\%) \textit{ and}$$

(inductive step)

$$\textit{for all } a \in A \textit{ and } w \in A^*,\ \textit{if}\ \ (\dagger)\ P(w),\ \textit{then}\ P(wa),$$

then,

$$\textit{for all } w \in A^*,\ P(w).$$

We refer to the formula (†) as the *inductive hypothesis*.

Proof. Equivalent to well-founded induction on $\mathbf{left}_A$. □

Theorem 2.2.3 (Principle of Strong String Induction)
Suppose $A \subseteq \mathbf{Sym}$ and $P(w)$ is a property of a string w. If

> *for all $w \in A^*$,*
> *if (†) for all $x \in A^*$, if x is a proper substring of w, then $P(x)$,*
> *then $P(w)$,*

then,

$$\textit{for all } w \in A^*,\ P(w).$$

We refer to (†) as the inductive hypothesis. It says that all the proper substrings of w have property P. According to the induction principle, to show that every $w \in A^*$ has property P, we let $w \in A^*$, and assume (the inductive hypothesis) that every proper substring of w has property P. Then we must show that w has property P.

Proof. Equivalent to well-founded induction on $\mathbf{strong}_A$. $\square$

The next subsection, on proving language equalities, contains two examples of proofs by strong string induction. Before moving on to that subsection, we give an example proof by right string induction.

We define the *reversal* $x^R \in \mathbf{Str}$ *of* a string x by right recursion on strings:

$$\begin{aligned} \%^R &= \%; \\ (ax)^R &= x^R a, \text{ for all } a \in \mathbf{Sym} \text{ and } x \in \mathbf{Str}. \end{aligned}$$

E.g., we have that $(\mathsf{021})^R = \mathsf{120}$. And, an easy calculation shows that, for all $a \in \mathbf{Sym}$, $a^R = a$. We let the reversal operation have higher precedence than string concatenation, so that, e.g., $xx^R = x(x^R)$.

Proposition 2.2.4
For all $x, y \in \mathbf{Str}$, $(xy)^R = y^R x^R$.

As usual, we must start by figuring out which of x and y to do induction on, as well as what sort of induction to use. Because we defined string reversal using right recursion, it turns out that we should do right string induction on x.

Proof. Suppose $y \in \mathbf{Str}$. Since $\mathbf{Sym}^* = \mathbf{Str}$, it will suffice to show that, for all $x \in \mathbf{Sym}^*$, $(xy)^R = y^R x^R$. We proceed by right string induction.

(Basis Step) We have that $(\%y)^R = y^R = y^R\% = y^R\%^R$.

(Inductive Step) Suppose $a \in \mathbf{Sym}$ and $x \in \mathbf{Sym}^*$. Assume the inductive hypothesis: $(xy)^R = y^R x^R$. Then,

$$\begin{aligned}
((ax)y)^R &= (a(xy))^R \\
&= (xy)^R a && \text{(definition of } (a(xy))^R\text{)} \\
&= (y^R x^R)a && \text{(inductive hypothesis)} \\
&= y^R(x^R a) \\
&= y^R(ax)^R && \text{(definition of } (ax)^R\text{).}
\end{aligned}$$

□

Exercise 2.2.5
Use right string induction and Proposition 2.2.4 to prove that, for all $x \in \mathbf{Str}$, $(x^R)^R = x$.

Exercise 2.2.6
In Section 2.1, we used right recursion to define the function $\mathbf{alphabet} \in \mathbf{Str} \to \mathbf{Alp}$. Use right string induction to show that, for all $x, y \in \mathbf{Str}$, $\mathbf{alphabet}(xy) = \mathbf{alphabet}\, x \cup \mathbf{alphabet}\, y$.

2.2.2 Proving Language Equalities

In this subsection, we show two examples of how strong string induction and induction over inductively defined languages can be used to show that two languages are equal.

For the first example, let X be the least subset of $\{0, 1\}^*$ such that:

(1) $\% \in X$; and

(2) for all $a \in \{0, 1\}$ and $x \in X$, $axa \in X$.

This is another example of an inductive definition: X consists of just those strings of 0's and 1's that can be constructed using (1) and (2). For example, by (1) and (2), we have that $00 = 0\%0 \in X$. Thus, by (2), we have that $1001 = 1(00)1 \in X$. In general, we have that X contains the elements:

$$\%, 00, 11, 0000, 0110, 1001, 1111, \ldots$$

We will show that $X = Y$, where $Y = \{\, w \in \{0, 1\}^* \mid w \text{ is a palindrome and } |w| \text{ is even} \,\}$.

Lemma 2.2.7
$Y \subseteq X$.

Proof. Since $Y \subseteq \{0,1\}^*$, it will suffice to show that, for all $w \in \{0,1\}^*$,

$$\text{if } w \in Y, \text{ then } w \in X.$$

We proceed by strong string induction.

Suppose $w \in \{0,1\}^*$, and assume the inductive hypothesis: for all $x \in \{0,1\}^*$, if x is a proper substring of w, then

$$\text{if } x \in Y, \text{ then } x \in X.$$

We must show that

$$\text{if } w \in Y, \text{ then } w \in X.$$

Suppose $w \in Y$, so that $w \in \{0,1\}^*$, w is a palindrome and $|w|$ is even. It remains to show that $w \in X$. If $w = \%$, then $w = \% \in X$, by Part (1) of the definition of X. So, suppose $w \neq \%$. Since $|w|$ is even, we have that $|w| \geq 2$, and thus that $w = axb$ for some $a, b \in \{0,1\}$ and $x \in \{0,1\}^*$. Because $|w|$ is even, it follows that $|x|$ is even. Furthermore, because w is a palindrome, it follows that $a = b$ and x is a palindrome. Thus $w = axa$ and $x \in Y$. Since x is a proper substring of w, the inductive hypothesis tells us that

$$\text{if } x \in Y, \text{ then } x \in X.$$

But $x \in Y$, and thus $x \in X$. Thus, by Part (2) of the definition of X, we have that $w = axa \in X$. $\square$

We could also prove $X \subseteq Y$ by strong string induction. But an alternative approach is more elegant and generally applicable: we use the induction principle that comes from the inductive definition of X.

Proposition 2.2.8 (Principle of Induction on X)
Suppose $P(w)$ is a property of a string w. If

(1)

$$P(\%), \textit{and}$$

(2)

$$\textit{for all } a \in \{0,1\} \textit{ and } x \in X, \textit{ if } (\dagger)\ P(x), \textit{ then } P(axa),$$

then,

$$\textit{for all } w \in X,\ P(w).$$

We refer to (†) as the *inductive hypothesis* of Part (2). By Part (1) of the definition of X, $\% \in X$. Thus Part (1) of the induction principle requires us to show $P(\%)$. By Part (2) of the definition of X, if $a \in \{\mathtt{0}, \mathtt{1}\}$ and $x \in X$, then $axa \in X$. Thus in Part (2) of the induction principle, when proving that the "new" element axa has property P, we're allowed to assume that the "old" element has property P.

Lemma 2.2.9
$X \subseteq Y$.

Proof. We use induction on X to show that, for all $w \in X$, $w \in Y$.

There are two steps to show.

(1) Since $\% \in \{\mathtt{0}, \mathtt{1}\}^*$, $\%$ is a palindrome and $|\%| = 0$ is even, we have that $\% \in Y$.

(2) Let $a \in \{\mathtt{0}, \mathtt{1}\}$ and $x \in X$. Assume the inductive hypothesis: $x \in Y$. We must show that $axa \in Y$. Since $x \in Y$, we have that $x \in \{\mathtt{0}, \mathtt{1}\}^*$, x is a palindrome and $|x|$ is even. Because $a \in \{\mathtt{0}, \mathtt{1}\}$ and $x \in \{\mathtt{0}, \mathtt{1}\}^*$, it follows that $axa \in \{\mathtt{0}, \mathtt{1}\}^*$. Since x is a palindrome, we have that axa is also a palindrome. And, because $|axa| = |x| + 2$ and $|x|$ is even, it follows that $|axa|$ is even. Thus $axa \in Y$, as required.

□

Proposition 2.2.10
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.7 and 2.2.9. □

We end this subsection by proving a more complex language equality. One of the languages is defined using a "difference" function on strings, which we will use a number of times in later chapters. Define $\mathbf{diff} \in \{\mathtt{0}, \mathtt{1}\}^* \to \mathbb{Z}$ by: for all $w \in \{\mathtt{0}, \mathtt{1}\}^*$,

$$\mathbf{diff}\, w = \text{the number of } \mathtt{1}\text{'s in } w - \text{the number of } \mathtt{0}\text{'s in } w.$$

Then:

- $\mathbf{diff}\, \% = 0$;
- $\mathbf{diff}\, \mathtt{1} = 1$;
- $\mathbf{diff}\, \mathtt{0} = -1$; and
- for all $x, y \in \{\mathtt{0}, \mathtt{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\, x + \mathbf{diff}\, y$.

Note that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}\, w = 0$ iff w has an equal number of 0's and 1's. If we think of a 1 as representing the production of one unit of some resource, and of a 0 as representing the consumption of one unit of that resource, then a string will have a diff of 0 iff it is balanced in terms of production and consumption. Note that such such a string may have prefixes with negative diff's, i.e., it may temporarily go "into the red".

Let X (forget the previous definition of X) be the least subset of $\{\mathsf{0},\mathsf{1}\}^*$ such that:

(1) $\% \in X$;

(2) for all $x, y \in X$, $xy \in X$;

(3) for all $x \in X$, $\mathsf{0}x\mathsf{1} \in X$; and

(4) for all $x \in X$, $\mathsf{1}x\mathsf{0} \in X$.

Let $Y = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{diff}\, w = 0 \,\}$.

For example, since $\% \in X$, it follows, by (3) and (4) that $\mathsf{01} = \mathsf{0}\%\mathsf{1} \in X$ and $\mathsf{10} = \mathsf{1}\%\mathsf{0} \in X$. Thus, by (2), we have that $\mathsf{0110} = (\mathsf{01})(\mathsf{10}) \in X$. And, Y consists of all strings of 0's and 1's with an equal number of 0's and 1's.

Our goal is to prove that $X = Y$, i.e., that: (the easy direction) every string that can be constructed using X's rules has an equal number of 0's and 1's; and (the hard direction) that every string of 0's and 1's with an equal number of 0's and 1's can be constructed using X's rules.

Because X was defined inductively, it gives rise to an induction principle, which we will use to prove the following lemma. (Because of Part (2) of the definition of X, we wouldn't be able to prove this lemma using strong string induction.)

Lemma 2.2.11
$X \subseteq Y$.

Proof. We use induction on X to show that, for all $w \in X$, $w \in Y$. There are four steps to show, corresponding to the four rules of X's definition.

(1) We must show $\% \in Y$. Since $\% \in \{\mathsf{0},\mathsf{1}\}^*$ and $\mathbf{diff}\,\% = 0$, we have that $\% \in Y$.

(2) Suppose $x, y \in X$, and assume the inductive hypothesis: $x, y \in Y$. We must show that $xy \in Y$. Since $x, y \in Y$, we have that $xy \in \{\mathsf{0},\mathsf{1}\}^*$ and $\mathbf{diff}(xy) = \mathbf{diff}\, x + \mathbf{diff}\, y = 0 + 0 = 0$. Thus $xy \in Y$.

(3) Suppose $x \in X$, and assume the inductive hypothesis: $x \in Y$. We must show that $\mathsf{0}x\mathsf{1} \in Y$. Since $x \in Y$, we have that $\mathsf{0}x\mathsf{1} \in \{\mathsf{0},\mathsf{1}\}^*$ and $\mathbf{diff}(\mathsf{0}x\mathsf{1}) = \mathbf{diff}\,\mathsf{0} + \mathbf{diff}\, x + \mathbf{diff}\,\mathsf{1} = -1 + 0 + 1 = 0$. Thus $\mathsf{0}x\mathsf{1} \in Y$.

(4) Suppose $x \in X$, and assume the inductive hypothesis: $x \in Y$. We must show that $1x0 \in Y$. Since $x \in Y$, we have that $1x0 \in \{0,1\}^*$ and $\mathbf{diff}(1x0) = \mathbf{diff}\,1 + \mathbf{diff}\,x + \mathbf{diff}\,0 = 1 + 0 + -1 = 0$. Thus $1x0 \in Y$.

□

Lemma 2.2.12
$Y \subseteq X$.

Proof. Since $Y \subseteq \{0,1\}^*$, it will suffice to show that, for all $w \in \{0,1\}^*$,

$$\text{if } w \in Y, \text{ then } w \in X.$$

We proceed by strong string induction. Suppose $w \in \{0,1\}^*$, and assume the inductive hypothesis: for all $x \in \{0,1\}^*$, if x is a proper substring of w, then

$$\text{if } x \in Y, \text{ then } x \in X.$$

We must show that

$$\text{if } w \in Y, \text{ then } w \in X.$$

Suppose $w \in Y$. We must show that $w \in X$. There are three cases to consider.

- Suppose $w = \%$. Then $w = \% \in X$, by Part (1) of the definition of X.

- Suppose $w = 0t$ for some $t \in \{0,1\}^*$. Since $w \in Y$, we have that $-1 + \mathbf{diff}\,t = \mathbf{diff}\,0 + \mathbf{diff}\,t = \mathbf{diff}(0t) = \mathbf{diff}\,w = 0$, and thus that $\mathbf{diff}\,t = 1$.

  Let u be the shortest prefix of t such that $\mathbf{diff}\,u \geq 1$. (Since t is a prefix of itself and $\mathbf{diff}\,t = 1 \geq 1$, it follows that u is well-defined.) Let $z \in \{0,1\}^*$ be such that $t = uz$. Clearly, $u \neq \%$, and thus $u = yb$ for some $y \in \{0,1\}^*$ and $b \in \{0,1\}$. Hence $t = uz = ybz$. Since y is a shorter prefix of t than u, we have that $\mathbf{diff}\,y \leq 0$.

  Suppose, toward a contradiction, that $b = 0$. Then $\mathbf{diff}\,y + -1 = \mathbf{diff}\,y + \mathbf{diff}\,0 = \mathbf{diff}\,y + \mathbf{diff}\,b = \mathbf{diff}(yb) = \mathbf{diff}\,u \geq 1$, so that $\mathbf{diff}\,y \geq 2$. But $\mathbf{diff}\,y \leq 0$—contradiction. Hence $b = 1$.

  Summarizing, we have that $u = yb = y1$, $t = uz = y1z$ and $w = 0t = 0y1z$. Since $\mathbf{diff}\,y + 1 = \mathbf{diff}\,y + \mathbf{diff}\,1 = \mathbf{diff}(y1) = \mathbf{diff}\,u \geq 1$, it follows that $\mathbf{diff}\,y \geq 0$. But $\mathbf{diff}\,y \leq 0$, and thus $\mathbf{diff}\,y = 0$. Thus $y \in Y$. Since $1 + \mathbf{diff}\,z = 0 + 1 + \mathbf{diff}\,z = \mathbf{diff}\,y + \mathbf{diff}\,1 + \mathbf{diff}\,z = \mathbf{diff}(y1z) = \mathbf{diff}\,t = 1$, it follows that $\mathbf{diff}\,z = 0$. Thus $z \in Y$.

  Because y and z are proper substrings of w, and $y, z \in Y$, the inductive hypothesis tells us that $y, z \in X$. Thus, by Part (3) of the definition of X, we have that $0y1 \in X$. Hence, Part (2) of the definition of X tells us that $w = 0y1z = (0y1)z \in X$.

- Suppose $w = 1t$ for some $t \in \{0,1\}^*$. Since $w \in Y$, we have that $1 + \textbf{diff}\, t = \textbf{diff}\, 1 + \textbf{diff}\, t = \textbf{diff}(1t) = \textbf{diff}\, w = 0$, and thus that $\textbf{diff}\, t = -1$.

  Let u be the shortest prefix of t such that $\textbf{diff}\, u \leq -1$. (Since t is a prefix of itself and $\textbf{diff}\, t = -1 \leq -1$, it follows that u is well-defined.) Let $z \in \{0,1\}^*$ be such that $t = uz$. Clearly, $u \neq \%$, and thus $u = yb$ for some $y \in \{0,1\}^*$ and $b \in \{0,1\}$. Hence $t = uz = ybz$. Since y is a shorter prefix of t than u, we have that $\textbf{diff}\, y \geq 0$.

  Suppose, toward a contradiction, that $b = 1$. Then $\textbf{diff}\, y + 1 = \textbf{diff}\, y + \textbf{diff}\, 1 = \textbf{diff}\, y + \textbf{diff}\, b = \textbf{diff}(yb) = \textbf{diff}\, u \leq -1$, so that $\textbf{diff}\, y \leq -2$. But $\textbf{diff}\, y \geq 0$—contradiction. Hence $b = 0$.

  Summarizing, we have that $u = yb = y0$, $t = uz = y0z$ and $w = 1t = 1y0z$. Since $\textbf{diff}\, y + -1 = \textbf{diff}\, y + \textbf{diff}\, 0 = \textbf{diff}(y0) = \textbf{diff}\, u \leq -1$, it follows that $\textbf{diff}\, y \leq 0$. But $\textbf{diff}\, y \geq 0$, and thus $\textbf{diff}\, y = 0$. Thus $y \in Y$. Since $-1 + \textbf{diff}\, z = 0 + -1 + \textbf{diff}\, z = \textbf{diff}\, y + \textbf{diff}\, 0 + \textbf{diff}\, z = \textbf{diff}(y0z) = \textbf{diff}\, t = -1$, it follows that $\textbf{diff}\, z = 0$. Thus $z \in Y$.

  Because y and z are proper substrings of w, and $y, z \in Y$, the inductive hypothesis tells us that $y, z \in X$. Thus, by Part (4) of the definition of X, we have that $1y0 \in X$. Hence, Part (2) of the definition of X tells us that $w = 1y0z = (1y0)z \in X$.

□

In the proof of the preceding lemma we made use of all four rules of X's definition. If this had not been the case, we would have known that the unused rules were redundant (or that we had made a mistake in our proof!).

Proposition 2.2.13
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.11 and 2.2.12. □

Exercise 2.2.14
Define the function $\textbf{diff} \in \{0,1\}^* \to \mathbb{Z}$ as in the above example. Let X be the least subset of $\{0,1\}^*$ such that:

(1) $\% \in X$;

(2) for all $x \in X$, $0x1 \in X$; and

(3) for all $x, y \in X$, $xy \in X$.

Let $Y = \{\, w \in \{0,1\}^* \mid \textbf{diff}\, w = 0 \text{ and, for all prefixes } v \text{ of } w, \textbf{diff}\, v \leq 0 \,\}$. Prove that $X = Y$.

Exercise 2.2.15
Define a function $\mathbf{diff} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{Z}$ by: for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

$$\mathbf{diff}\, w = \text{the number of } \mathsf{1}\text{'s in } w - 2(\text{the number of } \mathsf{0}\text{'s in } w).$$

Thus $\mathbf{diff}\,\% = 0$, $\mathbf{diff}\,\mathsf{0} = -2$, $\mathbf{diff}\,\mathsf{1} = 1$, and for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\,x + \mathbf{diff}\,y$. Furthermore, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}\,w = 0$ iff w has twice as many $\mathsf{1}$'s as $\mathsf{0}$'s. Let X be the least subset of $\{\mathsf{0},\mathsf{1}\}^*$ such that:

(1) $\% \in X$;

(2) $\mathsf{1} \in X$;

(3) for all $x, y \in X$, $\mathsf{1}x\mathsf{1}y\mathsf{0} \in X$; and

(4) for all $x, y \in X$, $xy \in X$.

Let $Y = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \text{for all prefixes } v \text{ of } w, \mathbf{diff}\,v \geq 0 \,\}$. Prove that $X = Y$.

2.2.3 Notes

A novel feature of this book is the introduction and use of explicit string induction principles, as an alternative to doing proofs by induction (mathematical or strong) on the length of strings. Also novel is our focus on languages defined using "difference" functions.

2.3 Introduction to Forlan

The Forlan toolset is an extension of the Standard ML of New Jersey (SML/NJ) implementation of Standard ML (SML). It is implemented as a set of SML modules. It is used interactively, and users can extend Forlan by definining SML functions.

Instructions for installing and running Forlan on machines running Linux, macOS and Windows can be found on the Forlan website:

`http://alleystoughton.us/forlan`.

A manual for Forlan is available on the Forlan website, and describes Forlan's modules in considerably more detail than does this book. See the manual for instructions for setting system parameters controlling such things as the search path used for loading files, the line length used by Forlan's pretty printer, and the number of elements of a list that the Forlan top-level displays.

In the concrete syntax for describing Forlan objects—automata, grammars, etc.—*comments* begin with a "`#`", and run through the end of the line. Comments and whitespace may be arbitrarily inserted into the descriptions of Forlan objects without changing how the objects will be lexically analyzed and parsed. For instance,

```
ab cd # this is a comment
efg h
```

describes the Forlan string `abcdefgh`. Forlan's input functions prompt with "`@`" when reading from the standard input, in which case the user signifies end-of-file by typing a line consisting of a single dot ("`.`").

We begin this section by showing how to invoke Forlan, and giving a quick introduction to the SML core of Forlan. We then show how symbols, strings, finite sets of symbols and strings, and finite relations on symbols can be manipulated using Forlan.

2.3.1 Invoking Forlan

To invoke Forlan, type the command `forlan` to your shell (command processor):

```
% forlan
Forlan Version m (based on Standard ML of New Jersey Version n)
val it = () : unit
-
```

(m and n will be the Forlan and SML/NJ versions, respectively.) The identifier `it` is normally bound to the value of the most recently evaluated expression. Initially, though, its value is the empty tuple `()`, the single element of the type `unit`. The value `()` is used in circumstances when a value is required, but it makes no difference what that value is.

Forlan's primary prompt is "`-`". To exit Forlan, type *CTRL*-`d` under Linux and macOS, and *CTRL*-`z` under Windows. To interrupt back to the Forlan top-level, type *CTRL*-`c`.

On Windows, you may find it more convenient to invoke Forlan by double-clicking on the Forlan icon. On all platforms, a much more flexible and satisfying way of running Forlan is as a subprocess of the Emacs text editor. See the Forlan website for information about how to do this.

2.3.2 The SML Core of Forlan

This subsection gives a quick introduction to the SML core of Forlan. Let's begin by using Forlan as a calculator:

```
- 4 + 5;
val it = 9 : int
- it * it;
val it = 81 : int
- it - 1;
val it = 80 : int
- 5 div 2;
val it = 2 : int
- 5 mod 2;
```

```
val it = 1 : int
- ~4 + 2;
val it = ~2 : int
```

Forlan responds to each expression by printing its value and type (`int` is the type of integers), and noting that the expression's value has been bound to the identifier `it`. Expressions must be terminated with semicolons. The operators `div` and `mod` compute integer division and remainder, respectively, and negative numbers begin with `~`.

In addition to the type `int` of integers, SML has types `string` and `bool`, product types $t_1 * \cdots * t_n$, and list types t `list`.

```
- "hello" ^ " " ^ "there";
val it = "hello there" : string
- true andalso (false orelse true);
val it = true : bool
- if 5 < 7 then "hello" else "bye";
val it = "hello" : string
- (3 + 1, 4 = 4, "a" ^ "b");
val it = (4,true,"ab") : int * bool * string
- #2 it;
val it = true : bool
- [1, 3, 5] @ [7, 9, 11];
val it = [1,3,5,7,9,11] : int list
- rev it;
val it = [11,9,7,5,3,1] : int list
- length it;
val it = 6 : int
- null[];
val it = true : bool
- null[1, 2];
val it = false : bool
- hd[1, 2, 3];
val it = 1 : int
- tl[1, 2, 3];
val it = [2,3] : int list
```

The operator `^` is string concatenation. The conjunction `andalso` evaluates its left-hand side first, and yields `false` without evaluating its right-hand side, if the value of the left-hand side is `false`. Similarly, the disjunction `orelse` evaluates its left-hand side first, and yields `true` without evaluating its right-hand side, if the value of the left-hand side is `true`. A conditional (`if-then-else`) is evaluated by first evaluating its boolean expression, and then evaluating its `then`-part, if the boolean expression's value is `true`, and evaluating its `else`-part, if its value is `false`. Tuples are evaluated from left to right, and the function `#`n selects the nth (starting from 1) element of a tuple. The operator `@` appends lists. The function `rev` reverses a list, the function `length` computes the length of a list, and the function `null` tests whether a list is empty. Finally, the functions `hd` and tl return the head (first element) and tail (all but the first element) of a list.

nil and :: (pronounced "cons", for "constructor"), which have types 'a list and 'a * 'a list -> 'a list, respectively, are the constructors for type 'a list. These constructors are *polymorphic*, having all of the types that can be formed by instantiating the type variable 'a with a type. E.g., nil has type int list, bool list, (int * bool)list, etc. :: is an infix operator, i.e., one writes $x :: xs$ for the list whose first element is x and remaining elements are those in the list xs.

```
- nil;
val it = [] : 'a list
- 1 :: nil;
val it = [1] : int list
- 1 :: 2 :: nil;
val it = [1,2] : int list
- 3 :: [5, 7, 9];
val it = [3,5,7,9] : int list
```

Lists are implemented as linked-lists, so that doing a cons involves the creation of a single list node.

SML also has option types t option, whose values are built using the type's two constructors: NONE of type 'a option, and SOME of type 'a -> 'a option. This is a predefined datatype, declared by

```
datatype 'a option = NONE | SOME of 'a
```

E.g., NONE, SOME 1 and SOME ~6 are three of the values of type int option, and NONE, SOME true and SOME false are the only values of type bool option.

```
- NONE;
val it = NONE : 'a option
- SOME 3;
val it = SOME 3 : int option
- SOME true;
val it = SOME true : bool option
```

In addition to the usual operators <, <=, > and >= for comparing integers, SML offers a function Int.compare of type int * int -> order, where the order type contains three elements: LESS, EQUAL and GREATER.

```
- Int.compare(3, 4);
val it = LESS : order
- Int.compare(4, 4);
val it = EQUAL : order
- Int.compare(4, 3);
val it = GREATER : order
```

It is possible to bind the value of an expression to an identifier using a value declaration:

```
- val x = 3 + 4;
val x = 7 : int
- val y = x + 1;
val y = 8 : int
- val x = 5 * x;
val x = 35 : int
- y;
val it = 8 : int
```

In the first declaration of x, its right-hand side is first evaluated, resulting in 7, and then x is bound to this value. Note that the redeclaration of x doesn't change the value of the previous declaration of x, it just makes that declaration inaccessible.

One can use a value declaration to give names to the components of a tuple, or give a name to the data of a non-NONE optional value:

```
- val (x, y, z) = (3 + 1, 4 = 4, "a" ^ "b");
val x = 4 : int
val y = true : bool
val z = "ab" : string
- val SOME n = SOME(4 * 25);
stdIn:39.5-39.26 Warning: binding not exhaustive
          SOME n = ...
val n = 100 : int
```

This last declaration uses pattern matching: SOME(4 * 25) is evaluated to SOME 100, and is then matched against the pattern SOME n. Because the constructors match, the pattern matching succeeds, and n becomes bound to 100. The warning is because the SML typechecker doesn't know the expression won't evaluate to NONE.

One can use a let expression to carry out some declarations in a local environment, evaluate an expression in that environment, and yield the result of that evaluation:

```
- val x = 3;
val x = 3 : int
- val z = 10;
val z = 10 : int
- let val x = 4 * 5
=     val y = x * z
= in (x, y, x + y) end;
val it = (20,200,220) : int * int * int
- x;
val it = 3 : int
```

When a declaration or expression spans more than one line, Forlan uses its secondary prompt, =, on all of the lines except for the first one. Forlan doesn't process a declaration or expression until it is terminated with a semicolon.

One can declare functions, and apply those functions to arguments:

```
- fun f n = n * 2;
val f = fn : int -> int
- f 3;
val it = 6 : int
- f(4 + 5);
val it = 18 : int
- f 4 + 5;
val it = 13 : int
- fun g(x, y) = (x ^ y, y ^ x);
val g = fn : string * string -> string * string
- val (u, v) = g("a", "b");
val u = "ab" : string
val v = "ba" : string
```

The function `f` doubles its argument. All function values are printed as `fn`. A type t_1 `->` t_2 is the type of all functions taking arguments of type t_1 and producing results (if they terminate without raising exceptions) of type t_2. SML infers the types of functions. The function application `f(4 + 5)` is evaluated as follows. First, the argument `4 + 5` is evaluated, resulting in `9`. Then a local envionment is created in which `n` is bound to `9`, and `f`'s body is evaluated in that environment, producing `18`. Function application has higher precedence than operators like `+`.

Technically, the function `g` matches its single argument, which must be a pair, against the pair pattern `(x, y)`, binding `x` and `y` to the left and right sides of this argument, and then evaluates its body. But we can think such a function as having multiple arguments. The type operator `*` has higher precedence than the operator `->`.

Except for basic entities like integers and booleans, all values in SML are represented by pointers, so that passing such a value to a function, or putting it in a datastructure, only involves copying a pointer.

Given functions f and g of types t_1 `->` t_2 and t_2 `->` t_3, respectively, g `o` f is the composition of g and f, the function of type t_1 `->` t_3 that, when given an argument x of type t_1, evaluates the expression g`(`f x`)`. For example, we have that:

```
- fun f x = x >= 1 andalso x <= 10;
val f = fn : int -> bool
- fun g x = if x then "inside" else "outside";
val g = fn : bool -> string
- val h = g o f;
val h = fn : int -> string
- h ~5;
val it = "outside" : string
- h 6;
val it = "inside" : string
- h 14;
val it = "outside" : string
```

SML also has anonymous functions, which may also be given names using value declarations:

```
- (fn x => x + 1)(3 + 4);
val it = 8 : int
- val f = fn x => x + 1;
val f = fn : int -> int
- f(3 + 4);
val it = 8 : int
```

The anonymous function `fn x => x + 1` has type `int -> int` and adds one to its argument.

Functions are data: they may be passed to functions, returned from functions (a function that returns a function is called *curried*), be components of tuples or lists, etc. For example,

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

is a polymorphic, curried function. The type operator `->` associates to the right, so that `map`'s type is

```
val map : ('a -> 'b) -> ('a list -> 'b list)
```

`map` takes in a function f of type `'a -> 'b`, and returns a function that when called with a list of elements of type `'a`, transforms each element using f, forming a list of elements of type `'b`.

```
- val f = map(fn x => x + 1);
val f = fn : int list -> int list
- f[2, 4, 6];
val it = [3,5,7] : int list
- f[~2, ~1, 0];
val it = [~1,0,1] : int list
- map (fn x => x mod 2 = 1) [3, 4, 5, 6, 7];
val it = [true,false,true,false,true] : bool list
```

In the last use of `map`, we are using the fact that function application associates to the left, so that $f\,x\,y$ means $(f\,x)y$, i.e., apply f to x, and then apply the resulting function to y.

The following example shows that local envionments are kept alive as as long as there are accessible function values referring to them:

```
- val f =
=       let val x = 2 * 10
=       in fn y => y * x end;
val f = fn : int -> int
- f 4;
val it = 80 : int
- f 7;
```

```
val it = 140 : int
```

If the local environment containing the binding of x was discarded, then calling f would fail.

It's also possible to declare recursive functions, like the factorial function:

```
- fun fact n =
=       if n = 0
=       then 1
=       else n * fact(n - 1);
val fact = fn : int -> int
- fact 4;
val it = 24 : int
```

One can load the contents of a file into Forlan using the function

```
val use : string -> unit
```

For example, if the file fact.sml contains the declaration of the factorial function, then this declaration can be loaded into the system as follows:

```
- use "fact.sml";
[opening fact.sml]
val fact = fn : int -> int
val it = () : unit
- fact 4;
val it = 24 : int
```

The factorial function can also be defined using pattern matching, either by using a case expression in the body of the function, or by using multiple clauses in the function's definition:

```
- fun fact n =
=       case n of
=            0 => 1
=          | n => n * fact(n - 1);
val fact = fn : int -> int
- fact 3;
val it = 6 : int
- fun fact 0 = 1
=   | fact n = n * fact(n - 1);
val fact = fn : int -> int
- fact 4;
val it = 24 : int
```

The order of the clauses of a case expression or function definition is significant. If the clauses of either the case expression or the function definition were reversed, the function being defined would never return.

Pattern matching is especially useful when doing list processing. E.g., we could (inefficiently) define the list reversal function like this:

```
- fun rev nil       = nil
=   | rev (x :: xs) = rev xs @ [x];
val rev = fn : ’a list -> ’a list
```

Calling `rev` with the empty list will result in the empty list being returned. And calling it with a nonempty list will temporarily bind `x` to the list’s head, bind `xs` to its tail, and then evaluate the expression `rev xs @ [x]`, recursively calling `rev`, and then returning the result of appending the result of this recursive call and `[x]`. Unfortunately, this definition of `rev` is slow for long lists, as `@` rebuilds its left argument. Instead, the official definition of `rev` is much more efficient:

```
- fun rev xs =
=       let fun rv(nil,     vs) = vs
=             | rv(u :: us, vs) = rv(us, u :: vs)
=       in rv(xs, nil) end;
val rev = fn : ’a list -> ’a list
- rev [1, 2, 3, 4];
val it = [4,3,2,1] : int list
```

When `rev` is called with `[1, 2, 3, 4]`, it calls the auxiliary function `rv` with the pair `([1, 2, 3, 4], [])`. The recursive calls that `rv` makes to itself are a special kind of recursion called *tail recursion*. The SML/NJ compiler generates code for such calls that simply jumps back to the beginning of `rv`, only changing its arguments. We have the following sequence of calls: `rv([1, 2, 3, 4], [])` calls `rv([2, 3, 4], [1])` calls `rv([3, 4], [2, 1])` calls `rv([4], [3, 2, 1])` calls `rv([], [4, 3, 2, 1])`, which returns `[4, 2, 2, 1]`, which is returned as the result of `rev`.

Finally, we can define recursive datatypes, and define functions by structural recursion on recursive datatypes. E.g., here’s how we can define the datatype of labeled binary trees (both leaves and nodes (non-leaves) can have labels):

```
- datatype (’a, ’b) tree =
=     Leaf of ’b
=   | Node of ’a * (’a, ’b) tree * (’a, ’b) tree;
datatype (’a,’b) tree
  = Leaf of ’b | Node of ’a * (’a,’b) tree * (’a,’b) tree
- Leaf;
val it = fn : ’a -> (’b,’a) tree
- Node;
val it = fn : ’a * (’a,’b) tree * (’a,’b) tree -> (’a,’b) tree
- val tr = Node(true, Node(false, Leaf 7, Leaf ~1), Leaf 8);
val tr = Node (true,Node (false,Leaf 7,Leaf ~1),Leaf 8)
  : (bool,int) tree
```

Then we can define a function for reversing a tree, and apply it to `tr`:

```
- fun revTree (Leaf n)            = Leaf n
```

```
=   | revTree (Node(m, tr1, tr2)) =
=       Node(m, revTree tr2, revTree tr1);
val revTree = fn : ('a,'b) tree -> ('a,'b) tree
- revTree tr;
val it = Node (true,Leaf 8,Node (false,Leaf ~1,Leaf 7))
  : (bool,int) tree
- revTree it;
val it = Node (true,Node (false,Leaf 7,Leaf ~1),Leaf 8)
  : (bool,int) tree
```

2.3.3 Symbols

The Forlan module `Sym` defines the abstract type `sym` of Forlan symbols, as well as some functions for processing symbols, including:

```
val input   : string -> sym
val output  : string * sym -> unit
val compare : sym * sym -> order
val equal   : string * string -> bool
```

Symbols are expressed in Forlan's syntax as sequences of symbol characters, i.e., as a or ⟨id⟩, rather than [a] or [⟨, i, d, ⟩]. The above functions behave as follows:

- `input` *fil* reads a symbol from file *fil*; if *fil* = `""`, then the symbol is read from the standard input;
- `output(`*fil*`,` *a*`)` writes the symbol *a* to the file *fil*; if *fil* = `""`, then the string is written to the standard output;
- `compare` implements our total ordering on symbols; and
- `equal` tests whether two symbols are equal.

All of Forlan's input functions read from the standard input when called with `""` instead of a file, and all of Forlan's output functions write to the standard output when given `""` instead of a file.

The type `sym` is bound in the top-level environment. On the other hand, one must write `Sym.`*f* to select the function *f* of module `Sym`. As described above, interactive input is terminated by a line consisting of a single "`.`" (dot), and Forlan's input prompt is "`@`".

The module `Sym` also provides the functions

```
val fromString : string -> sym
val toString   : sym -> string
```

where `fromString` is like `input`, except that it takes its input from a string, and `toString` is like `output`, except that it writes its output to a string. These functions are especially useful when defining functions. In the sequel, whenever a module/type has `input` and `output` functions, you may assume that it also has `fromString` and `toString` functions.

Here are some example uses of the functions of `Sym`:

```
- val a = Sym.input "";
@ <i
@ d>
@ .
val a = - : sym
- val b = Sym.fromString "<num>";
val b = - : sym
- Sym.output("", a);
<id>
val it = () : unit
- Sym.compare(a, b);
val it = LESS : order
- Sym.equal(a, b);
val it = false : bool
- Sym.equal(a, Sym.fromString "<id>");
val it = true : bool
```

Values of abstract types (like `sym`) are printed as "`-`".

2.3.4 Sets

The module `Set` defines the abstract type

```
type 'a set
```

of finite sets of elements of type `'a`. It is bound in the top-level environment. E.g., `sym set` is the type of sets of symbols.

Each set has an associated total ordering, and some of the functions of `Set` take total orderings as arguments. See the Forlan manual for the details. In the book, we won't have to work with such functions explicitly.

`Set` provides various constants and functions for processing sets, but we will only make direct use of a few of them:

```
val toList : 'a set -> 'a list
val size   : 'a set -> int
val empty  : 'a set
val sing   : 'a -> 'a set
val filter : ('a -> bool) -> 'a set -> 'a set
```

These values are polymorphic: `'a` can be `int`, `sym`, etc. The function `toList` returns the elements of a set, listing them in ascending order, according to the set's total ordering. The function `size` returns the size of a set. The value `empty` is the empty set. The function `sing` makes a value x into the singleton set $\{x\}$. And `filter` goes through the elements of a set, keeping those elements on which the supplied predicate function returns `true`.

2.3.5 Sets of Symbols

The module `SymSet` defines various functions for processing finite sets of symbols (elements of type `sym set`; alphabets), including:

```
val input    : string -> sym set
val output   : string * sym set -> unit
val fromList : sym list -> sym set
val memb     : sym * sym set -> bool
val subset   : sym set * sym set -> bool
val equal    : sym set * sym set -> bool
val union    : sym set * sym set -> sym set
val inter    : sym set * sym set -> sym set
val minus    : sym set * sym set -> sym set
val genUnion : sym set list -> sym set
val genInter : sym set list -> sym set
```

The total ordering associated with sets of symbols is our total ordering on symbols. Sets of symbols are expressed in Forlan's syntax as sequences of symbols, separated by commas.

The function `fromList` returns a set with the same elements of the list of symbols it is called with. The function `memb` tests whether a symbol is a member (element) of a set of symbols, `subset` tests whether a first set of symbols is a subset of a second one, and `equal` tests whether two sets of symbols are equal. The functions `union`, `inter` and `minus` compute the union, intersection and difference of two sets of symbols. The function `genUnion` computes the generalized intersection of a list of sets of symbols *xss*, returning the set of all symbols appearing in at least one element of *xss*. And, the function `genInter` computes the generalized intersection of a nonempty list of sets of symbols *xss*, returning the set of all symbols appearing in all elements of *xss*.

Here are some example uses of the functions of `SymSet`:

```
- val bs = SymSet.input "";
@ a, <id>, 0, <num>
@ .
val bs = - : sym set
- SymSet.output("", bs);
0, a, <id>, <num>
val it = () : unit
- val cs = SymSet.input "";
@ a, <char>
@ .
val cs = - : sym set
- SymSet.subset(cs, bs);
val it = false : bool
- SymSet.output("", SymSet.union(bs, cs));
0, a, <id>, <num>, <char>
val it = () : unit
```

```
- SymSet.output("", SymSet.inter(bs, cs));
a
val it = () : unit
- SymSet.output("", SymSet.minus(bs, cs));
0, <id>, <num>
val it = () : unit
- val ds = SymSet.fromString "<char>, <>";
val ds = - : sym set
- SymSet.output("", SymSet.genUnion[bs, cs, ds]);
0, a, <>, <id>, <num>, <char>
val it = () : unit
- SymSet.output("", SymSet.genInter[bs, cs, ds]);

val it = () : unit
```

2.3.6 Strings

We will be working with two kinds of strings:

- SML strings, i.e., elements of type `string`;
- The strings of formal language theory, which we call "formal language strings", when necessary.

The module `Str` defines the type `str` of formal language strings, which is bound in the top-level environment, and is equal to `sym list`, the type of lists of symbols. Because strings are lists, we can use SML's list processing functions on them. Strings are expressed in Forlan's syntax as either a single `%` or a nonempty sequence of symbols.

The module `Str` also defines some functions for processing strings, including:

```
val input      : string -> str
val output     : string * str -> unit
val alphabet   : str -> sym set
val compare    : str * str -> order
val equal      : str * str -> bool
val prefix     : str * str -> bool
val suffix     : str * str -> bool
val substr     : str * str -> bool
val power      : str * int -> str
val last       : str -> sym
val allButLast : str -> str
```

The function `alphabet` returns the alphabet of a string, and `compare` implements our total ordering on strings. `prefix(`x`, `y`)` tests whether x is a prefix of y, and `suffix` and `substring` work similarly. `power(`x`, `n`)` raises x to the power n. And `last` and `allButLast` return the last symbol and all but the last symbol of a string, respectively.

Here are some example uses of the functions of `Str`:

```
- val x = Str.input "";
@ hello<there>
@ .
val x = [-,-,-,-,-,-] : str
- length x;
val it = 6 : int
- Str.output("", x);
hello<there>
val it = () : unit
- SymSet.output("", Str.alphabet x);
e, h, l, o, <there>
val it = () : unit
- Str.output("", Str.power(x, 3));
hello<there>hello<there>hello<there>
val it = () : unit
- val y = Str.fromString "ello";
val y = [-,-,-,-] : str
- Str.compare(y, x);
val it = LESS : order
- Str.equal(y, x);
val it = false : bool
- Str.prefix(y, x);
val it = false : bool
- Str.substr(y, x);
val it = true : bool
- val z = Str.fromString "h" @ y;
val z = [-,-,-,-,-] : sym list
- Str.prefix(z, x);
val it = true : bool
- val x = Str.fromString "hellothere";
val x = [-,-,-,-,-,-,-,-,-,-] : str
- null x;
val it = false : bool
- Sym.output("", hd x);
h
val it = () : unit
- Str.output("", tl x);
ellothere
val it = () : unit
- Sym.output("", Str.last x);
e
val it = () : unit
- Str.output("", Str.allButLast x);
hellother
val it = () : unit
```

2.3.7 Sets of Strings

The module StrSet defines various functions for processing finite sets of strings (elements of type str set; finite languages), including:

```
val input    : string -> str set
val output   : string * str set -> unit
val fromList : str list -> str set
val memb     : str * str set -> bool
val subset   : str set * str set -> bool
val equal    : str set * str set -> bool
val union    : str set * str set -> str set
val inter    : str set * str set -> str set
val minus    : str set * str set -> str set
val genUnion : str set list -> str set
val genInter : str set list -> str set
val alphabet : str set -> sym set
```

The total ordering associated with sets of strings is our total ordering on strings. Sets of strings are expressed in Forlan's syntax as sequences of strings, separated by commas.

Here are some example uses of the functions of StrSet:

```
- val xs = StrSet.input "";
@ hello, <id><num>, %
@ .
val xs = - : str set
- val ys = StrSet.input "";
@ <id><num>, another
@ .
val ys = - : str set
- val zs = StrSet.union(xs, ys);
val zs = - : str set
- Set.size zs;
val it = 4 : int
- StrSet.output("", zs);
%, <id><num>, hello, another
val it = () : unit
- val us = Set.filter (fn x => length x mod 2 = 0) zs;
val us = - : sym list set
- StrSet.output("", us);
%, <id><num>
val it = () : unit
- SymSet.output("", StrSet.alphabet zs);
a, e, h, l, n, o, r, t, <id>, <num>
val it = () : unit
```

In this transcript, us was declared to be all the even-length elements of zs.

2.3.8 Relations on Symbols

The module `SymRel` defines the type `sym_rel` of finite relations on symbols. It is bound in the top-level environment, and is equal to `(sym * sym)set`, i.e., its elements are finite sets of pairs of symbols. The total ordering associated with relations on symbols orders pairs of symbols first according to their left-hand sides (using the total ordering on symbols), and then according to their right-hand sides. Relations on symbols are expressed in Forlan's syntax as sequences of ordered pairs `(`*a*`,`*b*`)` of symbols, separated by commas.

`SymRel` also defines various functions for processing finite relations on symbols, including:

```
val input           : string -> sym_rel
val output          : string * sym_rel -> unit
val fromList        : (sym * sym)list -> sym_rel
val memb            : (sym * sym) * sym_rel -> bool
val subset          : sym_rel * sym_rel -> bool
val equal           : sym_rel * sym_rel -> bool
val union           : sym_rel * sym_rel -> sym_rel
val inter           : sym_rel * sym_rel -> sym_rel
val minus           : sym_rel * sym_rel -> sym_rel
val genUnion        : sym_rel list -> sym_rel
val genInter        : sym_rel list -> sym_rel
val domain          : sym_rel -> sym set
val range           : sym_rel -> sym set
val relationFromTo  : sym_rel * sym_set * sym_set -> bool
val reflexive       : sym_rel * sym set -> bool
val symmetric       : sym_rel -> bool
val antisymmetric   : sym_rel -> bool
val transitive      : sym_rel -> bool
val total           : sym_rel -> bool
val inverse         : sym_rel -> sym_rel
val compose         : sym_rel * sym_rel -> sym_rel
val function        : sym_rel -> bool
val functionFromTo  : sym_rel * sym_set * sym_set -> bool
val injection       : sym_rel -> bool
val bijectionFromTo : sym_rel * sym_set * sym_set -> bool
val applyFunction   : sym_rel -> sym -> sym
val restrictFunction : sym_rel * sym_set -> sym_rel
val updateFunction  : sym_rel * sym * sym -> sym_rel
```

The functions `domain` and `range` return the domain and range, respectively, of a `relation`. `relationFromTo`(*rel*, *bs*, *cs*) tests whether *rel* is a relation from *bs* to *cs*.

`reflexive`(*rel*, *bs*) tests whether *rel* is reflexive on *bs*. The functions `symmetric`, `antisymmetric` and `transitive` test whether a relation is symmetric, antisymmetric or transitive, respectively. `total`(*rel*, *bs*) tests whether *rel* is total on *bs*.

The function inverse computes the inverse of a relation, and compose composes two relations.

The function function tests whether a relation is a function. The function applyFunction is curried. Given a relation *rel*, applyFunction checks that *rel* is a function, issuing an error message, and raising an exception, otherwise. If it is a function, it returns a function of type sym -> sym that, when called with a symbol a, will apply the function *rel* to a, issuing an error message if a is not in the domain of *rel*. functionFromTo(*rel*, *bs*, *cs*) tests whether *rel* is a function from *bs* to *cs*. The function injection tests whether a relation is an injective function. bijectionFromTo(*rel*, *bs*, *cs*) tests whether *rel* is a bijection from *bs* to *cs*.

restrictFunction(*rel*, *bs*) restricts the function *rel* to *bs*; it issues an error message if *rel* is not a function, or *bs* is not a subset of the domain of *rel*. And, updateFunction(*rel*, a, b) returns the updating of the function *rel* to send a to b; it issues an error message if *rel* isn't a function.

Here is how we can work with total orderings using functions from SymRel:

```
- val rel = SymRel.input "";
@ (0, 1), (1, 2), (0, 2), (0, 0), (1, 1), (2, 2)
@ .
val rel = - : sym_rel
- SymRel.output("", rel);
(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)
val it = () : unit
- SymSet.output("", SymRel.domain rel);
0, 1, 2
val it = () : unit
- SymSet.output("", SymRel.range rel);
0, 1, 2
val it = () : unit
- SymRel.relationFromTo
= (rel, SymSet.fromString "0, 1, 2", SymSet.fromString "0, 1, 2");
val it = true : bool
- SymRel.relationOn(rel, SymSet.fromString "0, 1, 2");
val it = true : bool
- SymRel.reflexive(rel, SymSet.fromString "0, 1, 2");
val it = true : bool
- SymRel.symmetric rel;
val it = false : bool
- SymRel.antisymmetric rel;
val it = true : bool
- SymRel.transitive rel;
val it = true : bool
- SymRel.total(rel, SymSet.fromString "0, 1, 2");
val it = true : bool
- val rel' = SymRel.inverse rel;
```

```
val rel' = - : sym_rel
- SymRel.output("", rel');
(0, 0), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2)
val it = () : unit
- val rel'' = SymRel.compose(rel', rel);
val rel'' = - : sym_rel
- SymRel.output("", rel'');
(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1),
(2, 2)
val it = () : unit
```

And here is how we can work with relations that are functions:

```
- val rel = SymRel.input "";
@ (1, 2), (2, 3), (3, 4)
@ .
val rel = - : sym_rel
- SymRel.output("", rel);
(1, 2), (2, 3), (3, 4)
val it = () : unit
- SymSet.output("", SymRel.domain rel);
1, 2, 3
val it = () : unit
- SymSet.output("", SymRel.range rel);
2, 3, 4
val it = () : unit
- SymRel.function rel;
val it = true : bool
- SymRel.functionFromTo
= (rel, SymSet.fromString "1, 2, 3", SymSet.fromString "2, 3, 4");
val it = true : bool
- SymRel.injection rel;
val it = true : bool
- SymRel.bijectionFromTo
= (rel, SymSet.fromString "1, 2, 3", SymSet.fromString "2, 3, 4");
val it = true : bool
- val f = SymRel.applyFunction rel;
val f = fn : sym -> sym
- Sym.output("", f(Sym.fromString "1"));
2
val it = () : unit
- Sym.output("", f(Sym.fromString "2"));
3
val it = () : unit
- Sym.output("", f(Sym.fromString "3"));
4
val it = () : unit
- Sym.output("", f(Sym.fromString "4"));
argument not in domain
```

```
uncaught exception Error
- val rel' = SymRel.input "";
@ (4, 3), (3, 2), (2, 1)
@ .
val rel' = - : sym_rel
- val rel'' = SymRel.compose(rel', rel);
val rel'' = - : sym_rel
- SymRel.functionFromTo
= (rel'', SymSet.fromString "1, 2, 3",
=  SymSet.fromString "1, 2, 3");
val it = true : bool
- SymRel.output("", rel'');
(1, 1), (2, 2), (3, 3)
val it = () : unit
```

2.3.9 Notes

The book and toolset were designed and developed together, which made it possible to minimize the notational and conceptual distance between the two.

Chapter 3

Regular Languages

In this chapter, we study our most restrictive set of languages, the regular languages. We begin by introducing regular expressions, and saying that a language is regular iff it is generated by a regular expression. We study regular expression equivalance, look at how regular expressions can be synthesized and proved correct, and study several algorithms for regular expression simplification.

We go on to study five kinds of finite automata, culminating in finite automata whose transitions are labeled by regular expressions. We introduce methods for synthesizing and proving the correctness of finite automata, and study numerous algorithms for processing and converting between regular expressions and finite automata. Because of these conversions, the set of languages accepted by the finite automata is exactly the regular languages. The chapter concludes by considering the application of regular expressions and finite automata to searching in text files, lexical analysis, and the design of finite state systems.

3.1 Regular Expressions and Languages

In this section, we define several operations on languages, say what regular expressions are, what they mean, and what regular languages are, and begin to show how regular expressions can be processed by Forlan.

3.1.1 Operations on Languages

The union, intersection and set-difference operations on sets are also operations on languages, i.e., if $L_1, L_2 \in \mathbf{Lan}$, then $L_1 \cup L_2$, $L_1 \cap L_2$ and $L_1 - L_2$ are all languages. (Since $L_1, L_2 \in \mathbf{Lan}$, we have that $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, for alphabets Σ_1 and Σ_2. Let $\Sigma = \Sigma_1 \cup \Sigma_2$, so that Σ is an alphabet, $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$. Thus $L_1 \cup L_2$, $L_1 \cap L_2$ and $L_1 - L_2$ are all subsets of Σ^*, and so are all languages.)

The first new operation on languages is language concatenation. The *concatenation* of languages L_1 and L_2 ($L_1 @ L_2$) is the language

$$\{\, x_1 @ x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2 \,\}.$$

I.e., $L_1 @ L_2$ consists of all strings that can be formed by concatenating an element of L_1 with an element of L_2. For example,

$$\begin{aligned}\{\mathsf{ab}, \mathsf{abc}\} @ \{\mathsf{cd}, \mathsf{d}\} &= \{(\mathsf{ab})(\mathsf{cd}), (\mathsf{ab})(\mathsf{d}), (\mathsf{abc})(\mathsf{cd}), (\mathsf{abc})(\mathsf{d})\} \\ &= \{\mathsf{abcd}, \mathsf{abd}, \mathsf{abccd}\}.\end{aligned}$$

Note that, if $L_1, L_2 \subseteq \Sigma^*$, for an alphabet Σ, then $L_1 @ L_2 \subseteq \Sigma^*$.

Concatenation of languages is associative: for all $L_1, L_2, L_3 \in \mathbf{Lan}$,

$$(L_1 @ L_2) @ L_3 = L_1 @ (L_2 @ L_3).$$

And, $\{\%\}$ is the identity for concatenation: for all $L \in \mathbf{Lan}$,

$$\{\%\} @ L = L @ \{\%\} = L.$$

Furthermore, $\emptyset$ is the zero for concatenation: for all $L \in \mathbf{Lan}$,

$$\emptyset @ L = L @ \emptyset = \emptyset.$$

We often abbreviate $L_1 @ L_2$ to L_1L_2.

Now that we know what language concatenation is, we can say what it means to raise a language to a power. We define the *language L^n formed by raising* a language L *to the power* $n \in \mathbb{N}$ by recursion on n:

$$\begin{aligned}L^0 &= \{\%\}, \text{ for all } L \in \mathbf{Lan}; \text{ and} \\ L^{n+1} &= LL^n, \text{ for all } L \in \mathbf{Lan} \text{ and } n \in \mathbb{N}.\end{aligned}$$

We assign this exponentiation operation higher precedence than concatenation, so that LL^n means $L(L^n)$ in the above definition. Note that, if $L \subseteq \Sigma^*$, for an alphabet Σ, then $L^n \subseteq \Sigma^*$, for all $n \in \mathbb{N}$.

For example, we have that

$$\begin{aligned}\{\mathsf{a}, \mathsf{b}\}^2 &= \{\mathsf{a}, \mathsf{b}\}\{\mathsf{a}, \mathsf{b}\}^1 = \{\mathsf{a}, \mathsf{b}\}\{\mathsf{a}, \mathsf{b}\}\{\mathsf{a}, \mathsf{b}\}^0 \\ &= \{\mathsf{a}, \mathsf{b}\}\{\mathsf{a}, \mathsf{b}\}\{\%\} = \{\mathsf{a}, \mathsf{b}\}\{\mathsf{a}, \mathsf{b}\} \\ &= \{\mathsf{aa}, \mathsf{ab}, \mathsf{ba}, \mathsf{bb}\}.\end{aligned}$$

Proposition 3.1.1
For all $L \in \mathbf{Lan}$ and $n, m \in \mathbb{N}$, $L^{n+m} = L^nL^m$.

Proof. An easy mathematical induction on n. The language L and the natural number m can be fixed at the beginning of the proof. □

Thus, if $L \in \mathbf{Lan}$ and $n \in \mathbb{N}$, then

$$L^{n+1} = LL^n \qquad \text{(definition)},$$

and

$$L^{n+1} = L^n L^1 = L^n L \qquad \text{(Proposition 3.1.1)}.$$

Another useful fact about language exponentiation is:

Proposition 3.1.2
For all $w \in \mathbf{Str}$ and $n \in \mathbb{N}$, $\{w\}^n = \{w^n\}$.

Proof. By mathematical induction on n. □

For example, we have that $\{\mathsf{01}\}^4 = \{(\mathsf{01})^4\} = \{\mathsf{01010101}\}$.

Now we consider a language operation that is named after Stephen Cole Kleene, one of the founders of formal language theory. The *Kleene closure* (or just *closure*) of a language L (L^*) is the language

$$\bigcup \{\, L^n \mid n \in \mathbb{N}\,\}.$$

Thus, for all w,

$$\begin{aligned} w \in L^* \quad &\text{iff} \quad w \in A, \text{ for some } A \in \{\, L^n \mid n \in \mathbb{N}\,\} \\ &\text{iff} \quad w \in L^n \text{ for some } n \in \mathbb{N}. \end{aligned}$$

Or, in other words:

- $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$; and
- L^* consists of all strings that can be formed by concatenating together some number (maybe none) of elements of L (the same element of L can be used as many times as is desired).

For example,

$$\begin{aligned} \{\mathsf{a}, \mathsf{ba}\}^* &= \{\mathsf{a}, \mathsf{ba}\}^0 \cup \{\mathsf{a}, \mathsf{ba}\}^1 \cup \{\mathsf{a}, \mathsf{ba}\}^2 \cup \cdots \\ &= \{\%\} \cup \{\mathsf{a}, \mathsf{ba}\} \cup \{\mathsf{aa}, \mathsf{aba}, \mathsf{baa}, \mathsf{baba}\} \cup \cdots \end{aligned}$$

If L is a language, then $L \subseteq \Sigma^*$ for some alphabet Σ, and thus L^* is also a subset of Σ^*—showing that L^* is a language, not just a set of strings.

Suppose $w \in \mathbf{Str}$. By Proposition 3.1.2, we have that, for all x,

$$\begin{aligned} x \in \{w\}^* \quad &\text{iff} \quad x \in \{w\}^n, \text{ for some } n \in \mathbb{N}, \\ &\text{iff} \quad x \in \{w^n\}, \text{ for some } n \in \mathbb{N}, \\ &\text{iff} \quad x = w^n, \text{ for some } n \in \mathbb{N}. \end{aligned}$$

If we write $\{\mathsf{0}, \mathsf{1}\}^*$, then this could mean:

- all strings over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ (Section 2.1); or
- the closure of the language $\{\mathtt{0}, \mathtt{1}\}$.

Fortunately, these languages are equal (both are all strings of 0's and 1's), and this kind of ambiguity is harmless.

We assign our operations on languages relative precedences as follows:

Highest: closure ($(\cdot)^*$) and raising to a power ($(\cdot)^n$);

Intermediate: concatenation (@, or just juxtapositioning); and

Lowest: union ($\cup$), intersection ($\cap$) and difference ($-$).

For example, if $n \in \mathbb{N}$ and $A, B, C \in \mathbf{Lan}$, then $A^*BC^n \cup B$ abbreviates $((A^*)B(C^n)) \cup B$. The language $((A \cup B)C)^*$ can't be abbreviated, since removing either pair of parentheses will change its meaning. If we removed the outer pair, then we would have $(A \cup B)(C^*)$, and removing the inner pair would yield $(A \cup (BC))^*$.

Suppose L, L_1 and L_2 are languages, and $n \in \mathbb{N}$. It is easy to see that $\mathbf{alphabet}(L_1 \cup L_2) = \mathbf{alphabet}(L_1) \cup \mathbf{alphabet}(L_2)$. And, if L_1 and L_2 are both nonempty, then $\mathbf{alphabet}(L_1L_2) = \mathbf{alphabet}(L_1) \cup \mathbf{alphabet}(L_2)$, and otherwise, $\mathbf{alphabet}(L_1L_2) = \emptyset$. Furthermore, if $n \geq 1$, then $\mathbf{alphabet}(L^n) = \mathbf{alphabet}(L)$; otherwise, $\mathbf{alphabet}(L^n) = \emptyset$. Finally, we have that $\mathbf{alphabet}(L^*) = \mathbf{alphabet}(L)$.

In Section 2.3, we introduced the Forlan module `StrSet`, which defines various functions for processing finite sets of strings, i.e., finite languages. This module also defines the functions

```
val concat : str set * str set -> str set
val power  : str set * int -> str set
```

which implement our concatenation and exponentiation operations on finite languages. Here are some examples of how these functions can be used:

```
- val xs = StrSet.fromString "ab, cd";
val xs = - : str set
- val ys = StrSet.fromString "uv, wx";
val ys = - : str set
- StrSet.output("", StrSet.concat(xs, ys));
abuv, abwx, cduv, cdwx
val it = () : unit
- StrSet.output("", StrSet.power(xs, 0));
%
val it = () : unit
- StrSet.output("", StrSet.power(xs, 1));
ab, cd
val it = () : unit
```

```
- StrSet.output("", StrSet.power(xs, 2));
abab, abcd, cdab, cdcd
val it = () : unit
- StrSet.output("", StrSet.power(xs, 3));
ababab, ababcd, abcdab, abcdcd, cdabab, cdabcd, cdcdab, cdcdcd
val it = () : unit
```

3.1.2 Regular Expressions

Next, we define the set of all regular expressions. Let the set **RegLab** of *regular expression labels* be

$$\mathbf{Sym} \cup \{\%, \$, \ast, @, +\}.$$

Let the set **Reg** of *regular expressions* be the least subset of $\mathbf{Tree}_{\mathbf{RegLab}}$ such that:

(empty string) $\% \in \mathbf{Reg}$;

(empty set) $\$ \in \mathbf{Reg}$;

(symbol) for all $a \in \mathbf{Sym}$, $a \in \mathbf{Reg}$;

(closure) for all $\alpha \in \mathbf{Reg}$, $\ast(\alpha) \in \mathbf{Reg}$;

(concatenation) for all $\alpha, \beta \in \mathbf{Reg}$, $@(\alpha, \beta) \in \mathbf{Reg}$; and

(union) for all $\alpha, \beta \in \mathbf{Reg}$, $+(\alpha, \beta) \in \mathbf{Reg}$.

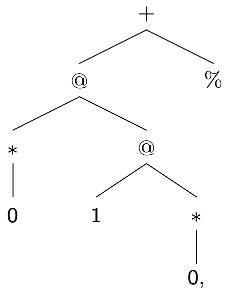
This is yet another example of an inductive definition. The elements of **Reg** are precisely those **RegLab**-trees (trees (See Section 1.3) whose labels come from **RegLab**) that can be built using these six rules.

Whenever possible, we will use the mathematical variables α, β and γ to name regular expressions. Since regular expressions are **RegLab**-trees, we may talk of their sizes and heights.

For example,

$$+(@(\ast(\mathsf{0}), @(\mathsf{1}, \ast(\mathsf{0}))), \%),$$

i.e.,



is a regular expression. On the other hand, the **RegLab**-tree $*(*, *)$ is *not* a regular expression, since it can't be built using our six rules.

We order the elements of **RegLab** as follows:

$$\% < \$ < \text{symbols in order} < * < @ < +.$$

It is important that $+$ be the greatest element of **RegLab**; if this were not so, then the definition of weakly simplified regular expressions (see Section **??**) would have to be altered.

We order regular expressions first by their root labels, and then, recursively, by their children, working from left to right. For example, we have that

$$\% < *(\%) < *(@(\$, *(\$))) < *(@(\mathsf{a}, \%)) < @(\%, \$).$$

Because **Reg** is defined inductively, it gives rise to an induction principle.

Theorem 3.1.3 (Principle of Induction on Regular Expressions) *Suppose $P(\alpha)$ is a property of a regular expression α. If*

- $P(\%)$,
- $P(\$)$,
- *for all $a \in \mathbf{Sym}$, $P(a)$,*
- *for all $\alpha \in \mathbf{Reg}$, if $P(\alpha)$, then $P(*(\alpha))$,*
- *for all $\alpha, \beta \in \mathbf{Reg}$, if $P(\alpha)$ and $P(\beta)$, then $P(@(\alpha, \beta))$, and*
- *for all $\alpha, \beta \in \mathbf{Reg}$, if $P(\alpha)$ and $P(\beta)$, then $P(+(\alpha, \beta))$,*

then

$$\textit{for all } \alpha \in \mathbf{Reg},\ P(\alpha).$$

To increase readability, we use infix and postfix notation, abbreviating:

- $*(\alpha)$ to α^* or $\alpha*$;
- $@(\alpha, \beta)$ to $\alpha \mathbin{@} \beta$; and
- $+(\alpha, \beta)$ to $\alpha + \beta$.

We assign the operators $(\cdot)^*$, $@$ and $+$ the following precedences and associativities:

Highest: $(\cdot)^*$;

Intermediate: $@$ (right associative); and

Lowest: $+$ (right associative).

We parenthesize regular expressions when we need to override the default precedences and associativities, and for reasons of clarity. Furthermore, we often abbreviate $\alpha \mathbin{@} \beta$ to $\alpha\beta$.

For example, we can abbreviate the regular expression

$$+(@(*(\mathsf{0}), @(\mathsf{1}, *(\mathsf{0}))), \%)$$

to $\mathsf{0}^* \mathbin{@} \mathsf{1} \mathbin{@} \mathsf{0}^* + \%$ or $\mathsf{0}^*\mathsf{1}\mathsf{0}^* + \%$. On the other hand, the regular expression $((\mathsf{0}+\mathsf{1})\mathsf{2})^*$ can't be further abbreviated, since removing either pair of parentheses would result in a different regular expression. Removing the outer pair would result in $(\mathsf{0}+\mathsf{1})(\mathsf{2}^*) = (\mathsf{0}+\mathsf{1})\mathsf{2}^*$, and removing the inner pair would yield $(\mathsf{0}+(\mathsf{1}\mathsf{2}))^* = (\mathsf{0}+\mathsf{1}\mathsf{2})^*$.

Now we can say what regular expressions mean, using some of our language operations. The *language generated by* a regular expression α ($L(\alpha)$) is defined by recursion:

$$\begin{aligned}
L(\%) &= \{\%\};\\
L(\$) &= \emptyset;\\
L(a) &= \{[a]\} = \{a\}, \text{ for all } a \in \textbf{Sym};\\
L(*(\alpha)) &= L(\alpha)^*, \text{ for all } \alpha \in \textbf{Reg};\\
L(@(\alpha,\beta)) &= L(\alpha) \mathbin{@} L(\beta), \text{ for all } \alpha,\beta \in \textbf{Reg}; \text{ and}\\
L(+(\alpha,\beta)) &= L(\alpha) \cup L(\beta), \text{ for all } \alpha,\beta \in \textbf{Reg}.
\end{aligned}$$

This is a good definition since, if L is a language, then so is L^*, and, if L_1 and L_2 are languages, then so are L_1L_2 and $L_1 \cup L_2$. We say that *w is generated by α* iff $w \in L(\alpha)$.

For example,

$$\begin{aligned}
L(\mathsf{0}^*\mathsf{1}\mathsf{0}^* + \%) &= L(+(@(*(\mathsf{0}), @(\mathsf{1}, *(\mathsf{0}))), \%))\\
&= L(@(*(\mathsf{0}), @(\mathsf{1}, *(\mathsf{0})))) \cup L(\%)\\
&= L(*(\mathsf{0}))L(@(\mathsf{1}, *(\mathsf{0}))) \cup \{\%\}\\
&= L(\mathsf{0})^*L(\mathsf{1})L(*(\mathsf{0})) \cup \{\%\}\\
&= \{\mathsf{0}\}^*\{\mathsf{1}\}L(\mathsf{0})^* \cup \{\%\}\\
&= \{\mathsf{0}\}^*\{\mathsf{1}\}\{\mathsf{0}\}^* \cup \{\%\}\\
&= \{\, \mathsf{0}^n\mathsf{1}\mathsf{0}^m \mid n, m \in \mathbb{N} \,\} \cup \{\%\}.
\end{aligned}$$

E.g., $\mathsf{0001000}$, $\mathsf{10}$, $\mathsf{001}$ and $\%$ are generated by $\mathsf{0}^*\mathsf{1}\mathsf{0}^* + \%$.

We define functions **symToReg** $\in$ **Sym**$\to$**Reg** and **strToReg** $\in$ **Str**$\to$**Reg**, as follows. Given a symbol $a \in \textbf{Sym}$, $\textbf{symToReg}\, a = a$. And, given a string x, $\textbf{strToReg}\, x$ is the *canonical regular expression for* x: $\%$, if $x = \%$, and $@(a_1, @(a_2, \ldots a_n \ldots)) = a_1a_2\ldots a_n$, if $x = a_1a_2\ldots a_n$, for symbols $a_1, a_2, \ldots, a_n$

and $n \geq 1$. It is easy to see that, for all $a \in \mathbf{Sym}$, $L(\mathbf{symToReg}\, a) = \{a\}$, and, for all $x \in \mathbf{Str}$, $L(\mathbf{strToReg}\, x) = \{x\}$.

We define the *regular expression* α^n *formed by raising* a regular expression α *to the power* $n \in \mathbb{N}$ by recursion on n:

$$\begin{aligned} \alpha^0 &= \%, \text{ for all } \alpha \in \mathbf{Reg}; \\ \alpha^1 &= \alpha, \text{ for all } \alpha \in \mathbf{Reg}; \text{ and} \\ \alpha^{n+1} &= \alpha\alpha^n, \text{ for all } \alpha \in \mathbf{Reg} \text{ and } n \in \mathbb{N} - \{0\}. \end{aligned}$$

We assign this operation the same precedence as closure, so that $\alpha\alpha^n$ means $\alpha(\alpha^n)$ in the above definition. Note that, in contrast to the definitions of x^n and L^n, we have split the case $n+1$ into two subcases, depending upon whether $n = 0$ or $n \geq 1$. Thus α^1 is α, not $\alpha\%$. For example, $(0+1)^3 = (0+1)(0+1)(0+1)$.

Proposition 3.1.4
For all $\alpha \in \mathbf{Reg}$ *and* $n \in \mathbb{N}$, $L(\alpha^n) = L(\alpha)^n$.

Proof. An easy mathematical induction on n. α may be fixed at the beginning of the proof. □

An example consequence of the proposition is that $L((0+1)^3) = L(0+1)^3 = \{0,1\}^3$, the set of all strings of 0's and 1's of length 3.

We define $\mathbf{alphabet} \in \mathbf{Reg} \to \mathbf{Alp}$ by recursion:

$$\begin{aligned} \mathbf{alphabet}\,\% &= \emptyset; \\ \mathbf{alphabet}\,\$ &= \emptyset; \\ \mathbf{alphabet}\,a &= \{a\}, \text{ for all } a \in \mathbf{Sym}; \\ \mathbf{alphabet}(*(\alpha)) &= \mathbf{alphabet}\,\alpha, \text{ for all } \alpha \in \mathbf{Reg}; \\ \mathbf{alphabet}(@(\alpha,\beta)) &= \mathbf{alphabet}\,\alpha \cup \mathbf{alphabet}\,\beta, \text{ for all } \alpha,\beta \in \mathbf{Reg}; \text{ and} \\ \mathbf{alphabet}(+(\alpha,\beta)) &= \mathbf{alphabet}\,\alpha \cup \mathbf{alphabet}\,\beta, \text{ for all } \alpha,\beta \in \mathbf{Reg}. \end{aligned}$$

This is a good definition, since the union of two alphabets is an alphabet. For example, $\mathbf{alphabet}(0^*10^* + \%) = \{0,1\}$. We say that $\mathbf{alphabet}\,\alpha$ is *the alphabet of* a regular expression α.

Proposition 3.1.5
For all $\alpha \in \mathbf{Reg}$, $\mathbf{alphabet}(L(\alpha)) \subseteq \mathbf{alphabet}\,\alpha$.

In other words, the proposition says that every symbol of every string in $L(\alpha)$ comes from $\mathbf{alphabet}\,\alpha$.

Proof. An easy induction on regular expressions. □

For example, since $L(\mathsf{1\$}) = \{\mathsf{1}\}\emptyset = \emptyset$, we have that

$$\begin{aligned}\textbf{alphabet}(L(\mathsf{0}^* + \mathsf{1\$})) &= \textbf{alphabet}(\{\mathsf{0}\}^*) \\ &= \{\mathsf{0}\} \\ &\subseteq \{\mathsf{0}, \mathsf{1}\} \\ &= \textbf{alphabet}(\mathsf{0}^* + \mathsf{1\$}).\end{aligned}$$

Next, we define some useful auxiliary functions on regular expressions. The *generalized concatenation* function $\textbf{genConcat} \in \textbf{List Reg} \to \textbf{Reg}$ is defined by right recursion:

$$\begin{aligned}\textbf{genConcat}\,[\,] &= \%, \\ \textbf{genConcat}\,[\alpha] &= \alpha, \text{ and} \\ \textbf{genConcat}([\alpha] \mathbin{@} \bar{\alpha}) &= @(\alpha, \textbf{genConcat}\,\bar{\alpha}), \text{ if } \bar{\alpha} \neq [\,].\end{aligned}$$

And the *generalized union* function $\textbf{genUnion} \in \textbf{List Reg} \to \textbf{Reg}$ is defined by right recursion:

$$\begin{aligned}\textbf{genUnion}\,[\,] &= \%, \\ \textbf{genUnion}\,[\alpha] &= \alpha, \text{ and} \\ \textbf{genUnion}([\alpha] \mathbin{@} \bar{\alpha}) &= +(\alpha, \textbf{genUnion}\,\bar{\alpha}), \text{ if } \bar{\alpha} \neq [\,].\end{aligned}$$

E.g., $\textbf{genConcat}[\mathsf{1}, \mathsf{0}, \mathsf{12}, \mathsf{3} + \mathsf{4}] = \mathsf{10(12)(3+4)}$ and $\textbf{genUnion}[\mathsf{1}, \mathsf{0}, \mathsf{12}, \mathsf{3} + \mathsf{4}] = \mathsf{1} + \mathsf{0} + (\mathsf{12}) + \mathsf{3} + \mathsf{4}$.

$\textbf{rightConcat} \in \textbf{Reg} \times \textbf{Reg} \to \textbf{Reg}$ is defined by structural recursion on its first argument:

$$\begin{aligned}\textbf{rightConcat}(@(\alpha_1, \alpha_2), \beta) &= @(\alpha_1, \textbf{rightConcat}(\alpha_2, \beta)), \text{ and} \\ \textbf{rightConcat}(\alpha, \beta) &= @(\alpha, \beta), \text{ if } \alpha \text{ is not a concatenation.}\end{aligned}$$

And $\textbf{rightUnion} \in \textbf{Reg} \times \textbf{Reg} \to \textbf{Reg}$ is defined by structural recursion on its first argument:

$$\begin{aligned}\textbf{rightUnion}(+(\alpha_1, \alpha_2), \beta) &= +(\alpha_1, \textbf{rightUnion}(\alpha_2, \beta)), \text{ and} \\ \textbf{rightUnion}(\alpha, \beta) &= +(\alpha, \beta), \text{ if } \alpha \text{ is not a union.}\end{aligned}$$

E.g., $\textbf{rightConcat}(\mathsf{012}, \mathsf{345}) = \mathsf{012345}$ and $\textbf{rightUnion}(\mathsf{0} + \mathsf{1} + \mathsf{2}, \mathsf{1} + \mathsf{2} + \mathsf{3}) = \mathsf{0} + \mathsf{1} + \mathsf{2} + \mathsf{1} + \mathsf{2} + \mathsf{3}$.

$\textbf{concatsToList} \in \textbf{Reg} \to \textbf{List Reg}$ is defined by structural recursion:

$$\begin{aligned}\textbf{concatsToList}(@(\alpha, \beta)) &= [\alpha] \mathbin{@} \textbf{concatsToList}\,\beta, \text{ and} \\ \textbf{concatsToList}\,\alpha &= \alpha, \text{ if } \alpha \text{ is not a concatenation.}\end{aligned}$$

And **unionsToList** $\in \textbf{Reg} \to \textbf{List Reg}$ is defined by structural recursion:

$$\begin{aligned}\textbf{unionsToList}(+(\alpha,\beta)) &= [\alpha] \mathbin{@} \textbf{unionsToList}\,\beta, \text{ and}\\ \textbf{unionsToList}\,\alpha &= \alpha, \text{ if } \alpha \text{ is not a union.}\end{aligned}$$

E.g., $\textbf{concatsToList}((12)34) = [12,3,4]$ and $\textbf{unionsToList}((0+1)+2+3) = [0+1,2,3]$.

Finally, **sortUnions** $\in \textbf{Reg} \to \textbf{Reg}$ is defined by:

$$\textbf{sortUnions}\,\alpha = \textbf{genUnion}\,\bar{\beta},$$

where $\bar{\beta}$ is the result of sorting the elements of $\textbf{unionsToList}\,\alpha$ into strictly ascending order (without duplicates), according to our total ordering on regular expressions. E.g., $\textbf{sortUnions}(1+0+23+1) = 0+1+23$.

We define functions **allSym** $\in \textbf{Alp} \to \textbf{Reg}$ and **allStr** $\in \textbf{Alp} \to \textbf{Reg}$ as follows. Given an alphabet Σ, $\textbf{allSym}\,\Sigma$ is the *all symbols regular expression* for Σ: $a_1 + \cdots + a_n$, where $a_1, \ldots, a_n$ are the elements of Σ, listed in order and without repetition (when $n = 0$, we use \$, and when $n = 1$, we use a_1). And, given an alphabet Σ, $\textbf{allStr}\,\Sigma$ is the *all strings regular expression* for Σ: $(\textbf{allSym}\,\Sigma)^*$. For example,

$$\begin{aligned}\textbf{allSym}\,\{0,1,2\} &= 0+1+2, \text{ and}\\ \textbf{allStr}\,\{0,1,2\} &= (0+1+2)^*.\end{aligned}$$

Thus, for all $\Sigma \in \textbf{Alp}$,

$$\begin{aligned}L(\textbf{allSym}\,\Sigma) &= \{\,[a] \mid a \in \Sigma\,\} = \{\,a \mid a \in \Sigma\,\}, \text{ and}\\ L(\textbf{allStr}\,\Sigma) &= \Sigma^*.\end{aligned}$$

Now we are able to say what it means for a language to be regular: a language L is *regular* iff $L = L(\alpha)$ for some $\alpha \in \textbf{Reg}$. We define

$$\begin{aligned}\textbf{RegLan} &= \{\,L(\alpha) \mid \alpha \in \textbf{Reg}\,\}\\ &= \{\,L \in \textbf{Lan} \mid L \text{ is regular}\,\}.\end{aligned}$$

Since every regular expression can be described, e.g., in fully parenthesized form, by a finite sequence of ASCII characters, we can enumerate the regular expressions, and consequently we have that **Reg** is countably infinite. Since $\{0^0\}$, $\{0^1\}$, $\{0^2\}$, $\ldots$, are all regular languages, we have that **RegLan** is infinite. Furthermore, we can establish an injection h from **RegLan** to **Reg**: $h\,L$ is the first (in our enumeration of regular expressions) α such that $L(\alpha) = L$. Because **Reg** is countably infinite, it follows that there is an injection from **Reg** to $\mathbb{N}$. Composing these injections, gives us an injection from **RegLan** to $\mathbb{N}$. And when observing that **RegLan** is infinite, we implicity gave an injection from $\mathbb{N}$

to **RegLan**. Thus, by the Schröder-Bernstein Theorem, we have that **RegLan** and $\mathbb{N}$ have the same size, so that **RegLan** is countably infinite.

Since **RegLan** is countably infinite but **Lan** is uncountable, it follows that **RegLan** $\subsetneq$ **Lan**, i.e., there are non-regular languages. In Section **??**, we will see a concrete example of a non-regular language.

3.1.3 Processing Regular Expressions in Forlan

Now, we turn to the Forlan implementation of regular expressions. The Forlan module `Reg` defines the abstract type `reg` (in the top-level environment) of regular expressions, as well as various functions and constants for processing regular expressions, including:

```
val input        : string -> reg
val output       : string * reg -> unit
val size         : reg -> int
val numLeaves    : reg -> int
val height       : reg -> int
val emptyStr     : reg
val emptySet     : reg
val fromSym      : sym -> reg
val closure      : reg -> reg
val concat       : reg * reg -> reg
val union        : reg * reg -> reg
val compare      : reg * reg -> order
val equal        : reg * reg -> bool
val fromStr      : str -> reg
val power        : reg * int -> reg
val alphabet     : reg -> sym set
val genConcat    : reg list -> reg
val genUnion     : reg list -> reg
val rightConcat  : reg * reg -> reg
val rightUnion   : reg * reg -> reg
val concatsToList : reg -> reg list
val unionsToList : reg -> reg list
val sortUnions   : reg -> reg
val allSym       : sym set -> reg
val allStr       : sym set -> reg
val fromStrSet   : str set -> reg
```

The Forlan syntax for regular expressions is the infix/postfix one introduced in the preceding subsection, where $\alpha \mathbin{@} \beta$ is always written as $\alpha\beta$, and we use parentheses to override default precedences/associativities, or simply for clarity. For example, $\mathtt{0}^*\mathtt{1}\mathtt{0}^* + \%$ and $(\mathtt{0}^*(\mathtt{1}(\mathtt{0}^*))) + \%$ are the same regular expression. And, $((\mathtt{0}^*)\mathtt{1})\mathtt{0}^* + \%$ is a different regular expression, but one with the same meaning. Furthermore, $\mathtt{0}^*\mathtt{1}(\mathtt{0}^* + \%)$ is not only different from the two preceding regular expressions, but it has a different meaning (it fails to generate %.) When regular expressions are outputted, as few parentheses as possible are used.

The functions size, numLeaves and height return the size, number of leaves and height, respectively, of a regular expression. The values emptyStr and emptySet are % and $, respectively. The function fromSym takes in a symbol a and returns the regular expression a. It is available in the top-level environment as symToReg. The function closure takes in a regular expression α and returns $*(\alpha)$. The function concat takes a pair (α, β) of regular expressions and returns $@(\alpha, \beta)$. The function union takes a pair (α, β) of regular expressions and returns $+(\alpha, \beta)$. The function compare implements our total ordering on regular expressions, and equal tests whether two regular expressions are equal. The function fromStr implements the function **strToReg**, and is also available in the top-level environment as strToReg. The function power raises a regular expression to a power, and the function alphabet returns the alphabet of a regular expression. Finally, the functions genConcat, genUnion, rightConcat, rightUnion, concatsToList, unionsToList, sortUnions, allSym and allStr implement the functions with the same names. The function fromStrSet returns $, if called with the empty set. Otherwise, it returns fromStr $x_1 + \cdots +$ fromStr x_n, where $x_1, \ldots, x_n$ are the elements of its argument, listed in strictly ascending order.

Here are some example uses of the functions of Reg:

```
- val reg = Reg.input "";
@ 0*10* + %
@ .
val reg = - : reg
- Reg.size reg;
val it = 9 : int
- Reg.numLeaves reg;
val it = 4 : int
- val reg' = Reg.fromStr(Str.power(Str.input "", 3));
@ 01
@ .
val reg' = - : reg
- Reg.output("", reg');
010101
val it = () : unit
- Reg.size reg';
val it = 11 : int
- Reg.numLeaves reg';
val it = 6 : int
- Reg.compare(reg, reg');
val it = GREATER : order
- val reg'' = Reg.concat(Reg.closure reg, reg');
val reg'' = - : reg
- Reg.output("", reg'');
(0*10* + %)*010101
val it = () : unit
- SymSet.output("", Reg.alphabet reg'');
```

```
0, 1
val it = () : unit
- val reg''' = Reg.power(reg, 3);
val reg''' = - : reg
- Reg.output("", reg''');
(0*10* + %)(0*10* + %)(0*10* + %)
val it = () : unit
- Reg.size reg''';
val it = 29 : int
- Reg.numLeaves reg''';
val it = 12 : int
- Reg.output("", Reg.fromString "(0*(1(0*))) + %");
0*10* + %
val it = () : unit
- Reg.output("", Reg.fromString "(0*1)0* + %");
(0*1)0* + %
val it = () : unit
- Reg.output("", Reg.fromString "0*1(0* + %)");
0*1(0* + %)
val it = () : unit
- Reg.equal
= (Reg.fromString "0*10* + %",
=  Reg.fromString "0*1(0* + %)");
val it = false : bool
```

We can use the functions `genConcat`, `genUnion`, `rightConcat`, `rightUnion`, `concatsToList`, `unionsToList` and `sortUnions` as follows:

```
- Reg.output("", Reg.genConcat nil);
%
val it = () : unit
- Reg.output("", Reg.genUnion nil);
$
val it = () : unit
- val regs =
=       [Reg.fromString "01", Reg.fromString "01 + 12",
=        Reg.fromString "(1 + 2)*", Reg.fromString "3 + 4"];
val regs = [-,-,-,-] : reg list
- Reg.output("", Reg.genConcat regs);
(01)(01 + 12)(1 + 2)*(3 + 4)
val it = () : unit
- Reg.output("", Reg.genUnion regs);
01 + (01 + 12) + (1 + 2)* + 3 + 4
val it = () : unit
- Reg.output
= ("",
=  Reg.rightConcat
=  (Reg.fromString "0123", Reg.fromString "4567"));
01234567
```

```
val it = () : unit
- Reg.output
= ("",
=  Reg.rightUnion
=  (Reg.fromString "0 + 1 + 2", Reg.fromString "1 + 2 + 3"));
0 + 1 + 2 + 1 + 2 + 3
val it = () : unit
- map
= Reg.toString
= (Reg.concatsToList(Reg.fromString "0(12)34"));
val it = ["0","12","3","4"] : string list
- map
= Reg.toString
= (Reg.unionsToList(Reg.fromString "0 + (1 + 2) + 3 + 0"));
val it = ["0","1 + 2","3","0"] : string list
- Reg.output
= ("", Reg.sortUnions(Reg.fromString "12 + 0 + 3 + 0"));
0 + 3 + 12
val it = () : unit
```

We can use the functions `allSym`, `allStr` and `fromStrSet` like this:

```
- Reg.output("", Reg.allSym(SymSet.fromString ""));
$
val it = () : unit
- Reg.output("", Reg.allSym(SymSet.fromString "0"));
0
val it = () : unit
- Reg.output("", Reg.allSym(SymSet.fromString "0, 1, 2"));
0 + 1 + 2
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString ""));
$*
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString "0"));
0*
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString "2, 1, 0"));
(0 + 1 + 2)*
val it = () : unit
- Reg.output
= ("",
=  Reg.fromStrSet(StrSet.fromString "one, two, three, four"));
one + two + four + three
val it = () : unit
```

3.1.4 JForlan

The Java program JForlan can be used to view and edit regular expression trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

3.1.5 Notes

A novel feature of this book is that regular expressions are trees, so that our linear syntax for regular expressions is derived rather than primary. Thus regular expression equality is just tree equality, and it's easy to explain when parentheses are necessary in a linear description of a regular expression. Furthermore, tree-oriented concepts, notation and operations automatically apply to regular expressions, letting us, e.g., give definitions by structural recursion.

3.2 Equivalence and Correctness of Regular Expressions

In this section, we say what it means for regular expressions to be equivalent, show a series of results about regular expression equivalence, and consider how regular expressions may be designed and proved correct.

3.2.1 Equivalence of Regular Expressions

Regular expressions α and β are *equivalent* iff $L(\alpha) = L(\beta)$. In other words, α and β are equivalent iff α and β generate the same language. We define a relation $\approx$ on **Reg** by: $\alpha \approx \beta$ iff α and β are equivalent. For example, $L((\mathsf{00})^* + \%) = L((\mathsf{00})^*)$, and thus $(\mathsf{00})^* + \% \approx (\mathsf{00})^*$.

One approach to showing that $\alpha \approx \beta$ is to show that $L(\alpha) \subseteq L(\beta)$ and $L(\beta) \subseteq L(\alpha)$. The following proposition is useful for showing language inclusions, not just ones involving regular languages.

Proposition 3.2.1

*(1) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 \cup A_2 \subseteq B_1 \cup B_2$.*

*(2) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 \cap A_2 \subseteq B_1 \cap B_2$.*

*(3) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $B_2 \subseteq A_2$, then $A_1 - A_2 \subseteq B_1 - B_2$.*

*(4) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 A_2 \subseteq B_1 B_2$.*

*(5) For all $A, B \in$ **Lan** and $n \in \mathbb{N}$, if $A \subseteq B$, then $A^n \subseteq B^n$.*

*(6) For all $A, B \in$ **Lan**, if $A \subseteq B$, then $A^* \subseteq B^*$.*

In Part (3), note that the second part of the sufficient condition for knowing $A_1 - A_2 \subseteq B_1 - B_2$ is $B_2 \subseteq A_2$, not $A_2 \subseteq B_2$.

Proof. (1) and (2) are straightforward. We show (3) as an example, below. (4) is easy. (5) is proved by mathematical induction, using (4). (6) is proved using (5).

For (3), suppose that $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$, $A_1 \subseteq B_1$ and $B_2 \subseteq A_2$. To show that $A_1 - A_2 \subseteq B_1 - B_2$, suppose $w \in A_1 - A_2$. We must show that $w \in B_1 - B_2$. It will suffice to show that $w \in B_1$ and $w \notin B_2$.

Since $w \in A_1 - A_2$, we have that $w \in A_1$ and $w \notin A_2$. Since $A_1 \subseteq B_1$, it follows that $w \in B_1$. Thus, it remains to show that $w \notin B_2$.

Suppose, toward a contradiction, that $w \in B_2$. Since $B_2 \subseteq A_2$, it follows that $w \in A_2$—contradiction. Thus we have that $w \notin B_2$. □

Next we show that our relation $\approx$ has some of the familiar properties of equality.

Proposition 3.2.2
(1) $\approx$ is reflexive on **Reg***, symmetric and transitive.*

(2) For all $\alpha, \beta \in$ **Reg***, if $\alpha \approx \beta$, then $\alpha^* \approx \beta^*$.*

(3) For all $\alpha_1, \alpha_2, \beta_1, \beta_2 \in$ **Reg***, if $\alpha_1 \approx \beta_1$ and $\alpha_2 \approx \beta_2$, then $\alpha_1\alpha_2 \approx \beta_1\beta_2$.*

(4) For all $\alpha_1, \alpha_2, \beta_1, \beta_2 \in$ **Reg***, if $\alpha_1 \approx \beta_1$ and $\alpha_2 \approx \beta_2$, then $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$.*

Proof. Follows from the properties of =. As an example, we show Part (4).

Suppose $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$, and assume that $\alpha_1 \approx \beta_1$ and $\alpha_2 \approx \beta_2$. Then $L(\alpha_1) = L(\beta_1)$ and $L(\alpha_2) = L(\beta_2)$, so that

$$\begin{aligned} L(\alpha_1 + \alpha_2) &= L(\alpha_1) \cup L(\alpha_2) = L(\beta_1) \cup L(\beta_2) \\ &= L(\beta_1 + \beta_2). \end{aligned}$$

Thus $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$. □

A consequence of Proposition 3.2.2 is the following proposition, which says that, if we replace a subtree of a regular expression α by an equivalent regular expression, that the resulting regular expression is equivalent to α.

Proposition 3.2.3
Suppose $\alpha, \beta, \beta' \in$ **Reg***, $\beta \approx \beta'$, $pat \in$* **Path** *is valid for α, and β is the subtree of α at position pat. Let α' be the result of replacing the subtree at position pat in α by β'. Then $\alpha \approx \alpha'$.*

Proof. By induction on α. □

Next, we state and prove some equivalences involving union.

Proposition 3.2.4
(1) For all $\alpha, \beta \in$ **Reg***, $\alpha + \beta \approx \beta + \alpha$.*

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$.

(3) For all $\alpha \in \mathbf{Reg}$, $\$ + \alpha \approx \alpha$.

(4) For all $\alpha \in \mathbf{Reg}$, $\alpha + \alpha \approx \alpha$.

(5) If $L(\alpha) \subseteq L(\beta)$, *then* $\alpha + \beta \approx \beta$.

Proof.

(1) Follows from the commutativity of $\cup$.

(2) Follows from the associativity of $\cup$.

(3) Follows since $\emptyset$ is the identity for $\cup$.

(4) Follows since $\cup$ is idempotent: $A \cup A = A$, for all sets A.

(5) Follows since, if $L_1 \subseteq L_2$, then $L_1 \cup L_2 = L_2$.

□

Next, we consider equivalences for concatenation.

Proposition 3.2.5
(1) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $(\alpha\beta)\gamma \approx \alpha(\beta\gamma)$.

(2) For all $\alpha \in \mathbf{Reg}$, $\%\alpha \approx \alpha \approx \alpha\%$.

(3) For all $\alpha \in \mathbf{Reg}$, $\$\alpha \approx \$ \approx \alpha\$$.

Proof.

(1) Follows from the associativity of language concatenation.

(2) Follows since $\{\%\}$ is the identity for language concatenation.

(3) Follows since $\emptyset$ is the zero for language concatenation.

□

Next we consider the distributivity of concatenation over union. First, we prove a proposition concerning languages. Then, we use this proposition to show the corresponding proposition for regular expressions.

Proposition 3.2.6
(1) For all $L_1, L_2, L_3 \in \mathbf{Lan}$, $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$.

(2) For all $L_1, L_2, L_3 \in \mathbf{Lan}$, $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$.

Proof. We show the proof of Part (1); the proof of the other part is similar. Suppose $L_1, L_2, L_3 \in \mathbf{Lan}$. It will suffice to show that

$$L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3).$$

To see that $L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3$, suppose $w \in L_1(L_2 \cup L_3)$. We must show that $w \in L_1L_2 \cup L_1L_3$. By our assumption, $w = xy$ for some $x \in L_1$ and $y \in L_2 \cup L_3$. There are two cases to consider.

- Suppose $y \in L_2$. Then $w = xy \in L_1L_2 \subseteq L_1L_2 \cup L_1L_3$.
- Suppose $y \in L_3$. Then $w = xy \in L_1L_3 \subseteq L_1L_2 \cup L_1L_3$.

To see that $L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3)$, suppose $w \in L_1L_2 \cup L_1L_3$. We must show that $w \in L_1(L_2 \cup L_3)$. There are two cases to consider.

- Suppose $w \in L_1L_2$. Then $w = xy$ for some $x \in L_1$ and $y \in L_2$. Thus $y \in L_2 \cup L_3$, so that $w = xy \in L_1(L_2 \cup L_3)$.
- Suppose $w \in L_1L_3$. Then $w = xy$ for some $x \in L_1$ and $y \in L_3$. Thus $y \in L_2 \cup L_3$, so that $w = xy \in L_1(L_2 \cup L_3)$.

□

Proposition 3.2.7
(1) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$.

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $(\alpha + \beta)\gamma \approx \alpha\gamma + \beta\gamma$.

Proof. Follows from Proposition 3.2.6. Consider, e.g., the proof of Part (1). By Proposition 3.2.6(1), we have that

$$\begin{aligned}
L(\alpha(\beta + \gamma)) &= L(\alpha)L(\beta + \gamma) \\
&= L(\alpha)(L(\beta) \cup L(\gamma)) \\
&= L(\alpha)L(\beta) \cup L(\alpha)L(\gamma) \\
&= L(\alpha\beta) \cup L(\alpha\gamma) \\
&= L(\alpha\beta + \alpha\gamma)
\end{aligned}$$

Thus $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$. □

Finally, we turn our attention to equivalences for Kleene closure, first stating and proving some results for languages, and then stating and proving the corresponding results for regular expressions.

Proposition 3.2.8
- *For all $L \in \mathbf{Lan}$, $LL^* \subseteq L^*$.*
- *For all $L \in \mathbf{Lan}$, $L^*L \subseteq L^*$.*

Proof. E.g., to see that $LL^* \subseteq L^*$, suppose $w \in LL^*$. Then $w = xy$ for some $x \in L$ and $y \in L^*$. Hence $y \in L^n$ for some $n \in \mathbb{N}$. Thus $w = xy \in LL^n = L^{n+1} \subseteq L^*$. □

Proposition 3.2.9

(1) $\emptyset^* = \{\%\}$.

(2) $\{\%\}^* = \{\%\}$.

(3) For all $L \in \mathbf{Lan}$, $L^*L = LL^*$.

(4) For all $L \in \mathbf{Lan}$, $L^*L^* = L^*$.

(5) For all $L \in \mathbf{Lan}$, $(L^*)^* = L^*$.

(6) For all $L_1 L_2 \in \mathbf{Lan}$, $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$.

Proof. The six parts can be proven in order using Proposition 3.2.1. All parts but (2), (5) and (6) can be proved without using induction.

As an example, we show the proof of (5). To show that $(L^*)^* = L^*$, it will suffice to show that $(L^*)^* \subseteq L^* \subseteq (L^*)^*$.

To see that $(L^*)^* \subseteq L^*$, we use mathematical induction to show that, for all $n \in \mathbb{N}$, $(L^*)^n \subseteq L^*$.

(Basis Step) We have that $(L^*)^0 = \{\%\} = L^0 \subseteq L^*$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $(L^*)^n \subseteq L^*$. We must show that $(L^*)^{n+1} \subseteq L^*$. By the inductive hypothesis, Proposition 3.2.1(4) and Part (4), we have that $(L^*)^{n+1} = L^*(L^*)^n \subseteq L^*L^* = L^*$.

Now, we use the result of the induction to prove that $(L^*)^* \subseteq L^*$. Suppose $w \in (L^*)^*$. We must show that $w \in L^*$. Since $w \in (L^*)^*$, we have that $w \in (L^*)^n$ for some $n \in \mathbb{N}$. Thus, by the result of the induction, $w \in (L^*)^n \subseteq L^*$.

Finally, for the other inclusion, we have that $L^* = (L^*)^1 \subseteq (L^*)^*$. □

Exercise 3.2.10

Prove Proposition 3.2.9(6), i.e., for all $L_1 L_2 \in \mathbf{Lan}$, $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$.

Proposition 3.2.11

(1) $\$^* \approx \%$.

(2) $\%^* \approx \%$.

(3) For all $\alpha \in \mathbf{Reg}$, $\alpha^*\alpha \approx \alpha\,\alpha^*$.

(4) For all $\alpha \in \mathbf{Reg}$, $\alpha^*\alpha^* \approx \alpha^*$.

(5) For all $\alpha \in \mathbf{Reg}$, $(\alpha^*)^* \approx \alpha^*$.

*(6) For all $\alpha, \beta \in$ **Reg**, $(\alpha\beta)^*\alpha \approx \alpha(\beta\alpha)^*$.*

Proof. Follows from Proposition 3.2.9. Consider, e.g., the proof of Part (5). By Proposition 3.2.9(5), we have that

$$L((\alpha^*)^*) = L(\alpha^*)^* = (L(\alpha)^*)^* = L(\alpha)^* = L(\alpha^*).$$

Thus $(\alpha^*)^* \approx \alpha^*$. □

3.2.2 Proving the Correctness of Regular Expressions

In this subsection, we use two examples to show how regular expressions can be designed and proved correct.

For our first example, define a function **zeros** $\in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{N}$ by recursion:

$$\begin{aligned}\mathbf{zeros}\,\% &= 0,\\ \mathbf{zeros}(0w) &= \mathbf{zeros}\,w + 1, \text{for all } w \in \{\mathsf{0},\mathsf{1}\}^*, \text{ and}\\ \mathbf{zeros}(1w) &= \mathbf{zeros}\,w, \text{for all } w \in \{\mathsf{0},\mathsf{1}\}^*.\end{aligned}$$

Thus $\mathbf{zeros}\,w$ is the number of occurrences of $\mathsf{0}$ in w. It is easy to show that:

- $\mathbf{zeros}\,\mathsf{0} = 1$;
- $\mathbf{zeros}\,\mathsf{1} = 0$;
- for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{zeros}(xy) = \mathbf{zeros}\,x + \mathbf{zeros}\,y$;
- for all $n \in \mathbb{N}$, $\mathbf{zeros}(\mathsf{0}^n) = n$; and
- for all $n \in \mathbb{N}$, $\mathbf{zeros}(\mathsf{1}^n) = 0$.

Let

$$X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{zeros}\,w \text{ is even} \,\},$$

so that X is all strings of $\mathsf{0}$'s and $\mathsf{1}$'s with an even number of $\mathsf{0}$'s. Clearly, $\% \in X$ and $\{\mathsf{1}\}^* \subseteq X$.

Let's consider the problem of finding a regular expression that generates X. A string with this property would begin with some number of $\mathsf{1}$'s (possibly none). After this, the string would have some number of parts (possibly none), each consisting of a $\mathsf{0}$, followed by some number of $\mathsf{1}$'s, followed by a $\mathsf{0}$, followed by some number of $\mathsf{1}$'s. The above considerations lead us to the regular expression

$$\alpha = \mathsf{1}^*(\mathsf{0}\mathsf{1}^*\mathsf{0}\mathsf{1}^*)^*.$$

To prove $L(\alpha) = X$, it's helpful to give names to the meanings of two parts of α. Let

$$Y = \{\mathsf{0}\}\{\mathsf{1}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^* \quad \text{and} \quad Z = \{\mathsf{1}\}^*Y^*,$$

so that $L(\mathsf{0}\mathsf{1}^*\mathsf{0}\mathsf{1}^*) = Y$ and $L(\alpha) = Z$. Thus it will suffice to prove that $Z = X$, and we do this by showing $Z \subseteq X \subseteq Z$. We begin by showing $Z \subseteq X$.

Lemma 3.2.12

(1) $Y \subseteq X$.

(2) $XX \subseteq X$.

Proof.

(1) Suppose $w \in Y$, so that $w = \mathsf{0}x\mathsf{0}y$ for some $x, y \in \{\mathsf{1}\}^*$. Thus $\mathbf{zeros}\, w = \mathbf{zeros}(\mathsf{0}x\mathsf{0}y) = \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, x + \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, y = 1 + 0 + 1 + 0 = 2$ is even, so that $w \in X$.

(2) Suppose $w \in XX$, so that $w = xy$ for some $x, y \in X$. Then $\mathbf{zeros}\, x$ and $\mathbf{zeros}\, y$ are even, so that $\mathbf{zeros}\, w = \mathbf{zeros}\, x + \mathbf{zeros}\, y$ is even. Hence $w \in X$.

□

Lemma 3.2.13
$Y^* \subseteq X$.

Proof. It will suffice to show that, for all $n \in \mathbb{N}$, $Y^n \subseteq X$, and we show this using mathematical induction.

(Basis Step) We have that $Y^0 = \{\%\} \subseteq X$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $Y^n \subseteq X$. Then $Y^{n+1} = YY^n \subseteq XX \subseteq X$, by Lemma 3.2.12

□

Lemma 3.2.14
$Z \subseteq X$.

Proof. By Lemmas 3.2.12 and 3.2.13, we have that $Z = \{\mathsf{1}\}^* Y^* \subseteq XX \subseteq X$. □

To prove $X \subseteq Z$, it's helpful to define another language:

$$U = \{\, w \in X \mid \mathsf{1} \text{ is not a prefix of } w \,\}.$$

Lemma 3.2.15
$U \subseteq Y^*$.

Proof. Because $U \subseteq \{\mathsf{0}, \mathsf{1}\}^*$, it will suffice to show that, for all $w \in \{\mathsf{0}, \mathsf{1}\}^*$,

$$\text{if } w \in U, \text{ then } w \in Y^*.$$

We proceed by strong string induction. Suppose $w \in \{\mathsf{0}, \mathsf{1}\}^*$, and assume the inductive hypothesis: for all $x \in \{\mathsf{0}, \mathsf{1}\}^*$, if x is a proper substring of w, then

$$\text{if } x \in U, \text{ then } x \in Y^*.$$

We must show that

$$\text{if } w \in U, \text{ then } w \in Y^*.$$

Suppose $w \in U$, so that $\mathbf{zeros}\, w$ is even and $\mathsf{1}$ is not a prefix of w. We must show that $w \in Y^*$. If $w = \%$, then $w \in Y^*$. Otherwise, $w = \mathsf{0}x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since $1 + \mathbf{zeros}\, x = \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, x = \mathbf{zeros}(\mathsf{0}x) = \mathbf{zeros}\, w$ is even, we have that $\mathbf{zeros}\, x$ is odd. Let n be the largest element of $\mathbb{N}$ such that $\mathsf{1}^n$ is a prefix of x. (n is well-defined, since $\mathsf{1}^0 = \%$ is a prefix of x.) Thus $x = \mathsf{1}^n y$ for some $y \in \{\mathsf{0},\mathsf{1}\}^*$. Since $\mathbf{zeros}\, y = 0 + \mathbf{zeros}\, y = \mathbf{zeros}\, \mathsf{1}^n + \mathbf{zeros}\, y = \mathbf{zeros}(\mathsf{1}^n y) = \mathbf{zeros}\, x$ is odd, we have that $y \neq \%$. And, by the definition of n, $\mathsf{1}$ is not a prefix of y. Hence $y = \mathsf{0}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Since $1 + \mathbf{zeros}\, z = \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, z = \mathbf{zeros}(\mathsf{0}z) = \mathbf{zeros}\, y$ is odd, we have that $\mathbf{zeros}\, z$ is even. Let m be the largest element of $\mathbb{N}$ such that $\mathsf{1}^m$ is a prefix of z. Thus $z = \mathsf{1}^m u$ for some $u \in \{\mathsf{0},\mathsf{1}\}^*$, and $\mathsf{1}$ is not a prefix of u. Since $\mathbf{zeros}\, u = 0 + \mathbf{zeros}\, u = \mathbf{zeros}\, \mathsf{1}^m + \mathbf{zeros}\, u = \mathbf{zeros}(\mathsf{1}^m u) = \mathbf{zeros}\, z$ is even, it follows that $u \in U$. Summarizing, we have that $w = \mathsf{0}x = \mathsf{01}^n y = \mathsf{01}^n\mathsf{0}z = \mathsf{01}^n\mathsf{01}^m u$ and $u \in U$. Since u is a proper substring of w, the inductive hypothesis tells us that $u \in Y^*$. Hence $w = \mathsf{01}^n\mathsf{01}^m u = (\mathsf{01}^n\mathsf{01}^m)u \in YY^* \subseteq Y^*$. □

Lemma 3.2.16
$X \subseteq Z$.

Proof. Suppose $w \in X$. Let n be the largest element of $\mathbb{N}$ such that $\mathsf{1}^n$ is a prefix of w. Thus $w = \mathsf{1}^n x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since $w \in X$, we have that $\mathbf{zeros}\, x = 0 + \mathbf{zeros}\, x = \mathbf{zeros}\, \mathsf{1}^n + \mathbf{zeros}\, x = \mathbf{zeros}\, w$ is even, so that $x \in X$. By the definition of n, we have that $\mathsf{1}$ is not a prefix of x, and thus $x \in U$. Hence $w = \mathsf{1}^n x \in \{\mathsf{1}\}^* U \subseteq \{\mathsf{1}\}^* Y^* = Z$, by Lemma 3.2.15. □

By Lemmas 3.2.14 and 3.2.16, we have that $Z \subseteq X \subseteq Z$, completing our proof that α is correct.

Our second example of regular expression design and proof of correction involves the languages

$$\begin{aligned} A &= \{\mathsf{001}, \mathsf{011}, \mathsf{101}, \mathsf{111}\}, \text{ and} \\ B &= \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \text{for all } x, y \in \{\mathsf{0},\mathsf{1}\}^*, \text{ if } w = x\mathsf{0}y, \text{then there is a } z \in A \\ &\qquad \text{such that } z \text{ is a prefix of } y \,\}. \end{aligned}$$

The elements of A can be thought of as the odd numbers between 1 and 7, expressed in binary, and B consists of those strings of $\mathsf{0}$'s and $\mathsf{1}$'s in which every occurrence of $\mathsf{0}$ is immediately followed by an element of A.

E.g., $\% \in B$, since the empty string has no occurrences of $\mathsf{0}$, and $\mathsf{00111}$ is in B, since its first $\mathsf{0}$ is followed by $\mathsf{011}$ and its second $\mathsf{0}$ is followed by $\mathsf{111}$. But $\mathsf{0000111}$ is not in B, since its first $\mathsf{0}$ is followed by $\mathsf{000}$, which is not in A. And $\mathsf{011}$ is not in B, since $|11| < 3$.

Note that, for all $x, y \in B$, $xy \in B$, i.e., $BB \subseteq B$. This holds, since: each occurrence of $\mathsf{0}$ in x is followed by an element of A in x, and is thus followed by the same element of A in xy; and each occurrence of $\mathsf{0}$ in y is followed by an element of A in y, and is thus followed by the same element of A in xy.

Furthermore, for all strings x, y, if $xy \in B$, then y is in B, i.e., every suffix of an element of B is also in B. This holds since if there was an occurrence of $\mathsf{0}$ in y that wasn't followed by an element of A, then this same occurrence of $\mathsf{0}$ in the suffix y of xy would also not be followed by an element of A, contradicting $xy \in B$.

How should we go about finding a regular expression α such that $L(\alpha) = B$? Because $\% \in B$, for all $x, y \in B$, $xy \in B$, and for all strings x, y, if $xy \in B$ then $y \in B$, our regular expression can have the form β^*, where β generates all the strings that are *basic* in the sense that they are nonempty elements of B with no non-empty proper prefixes that are in B.

Let's try to understand what the basic strings look like. Clearly, $\mathsf{1}$ is basic, so there will be no more basic strings that begin with $\mathsf{1}$. But what about the basic strings beginning with $\mathsf{0}$? No sequence of $\mathsf{0}$'s is basic, and any string that begins with four or more $\mathsf{0}$'s will not be basic. It is easy to see that $\mathsf{000111}$ is basic. In fact, it is the only basic string of the form $\mathsf{000}u$. (The first $\mathsf{0}$ forces u to begin with $\mathsf{1}$, the second $\mathsf{0}$ forces u to continue with $\mathsf{1}$, and the third forces u to continue with $\mathsf{1}$. And, if $|u| > 3$, then the overall string would have a nonempty, proper prefix in B, and so wouldn't be basic.) Similarly, $\mathsf{00111}$ is the only basic string beginning with $\mathsf{001}$. But what about the basic strings beginning with $\mathsf{01}$? It's not hard to see that there are infinitely many such strings: $\mathsf{0111}$, $\mathsf{010111}$, $\mathsf{01010111}$, $\mathsf{0101010111}$, etc. Fortunately, there is a simple pattern here: we have all strings of the form $\mathsf{0}(\mathsf{10})^n\mathsf{111}$ for $n \in \mathbb{N}$.

By the above considerations, it seems that we can let our regular expression be

$$(\mathsf{1} + \mathsf{0}(\mathsf{10})^*\mathsf{111} + \mathsf{00111} + \mathsf{000111})^*.$$

But, using some of the equivalences we learned about above, we can turn this regular expression into

$$(\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111})^*,$$

which we take as our α. Now, we prove that $L(\alpha) = B$.

Let

$$X = \{\mathsf{0}\} \cup \{\mathsf{00}\} \cup \{\mathsf{10}\}^* \quad \text{and} \quad Y = \{\mathsf{1}\} \cup \{\mathsf{0}\}X\{\mathsf{111}\}.$$

Then, we have that

$$\begin{aligned} X &= L(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*), \\ Y &= L(\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111}), \text{ and} \\ Y^* &= L((\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111})^*) = L(\alpha). \end{aligned}$$

Thus, it will suffice to show that $Y^* = B$. We will show that $Y^* \subseteq B \subseteq Y^*$.

To begin with, we would like to use mathematical induction to prove that, for all $n \in \mathbb{N}$, $\{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq B$. But in order for the inductive step to succeed, we must prove something stronger. Let

$$C = \{\, w \in B \mid \mathsf{01} \text{ is a prefix of } w \,\}.$$

Lemma 3.2.17
For all $n \in \mathbb{N}$, $\{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq C$.

Proof. We proceed by mathematical induction.

(Basis Step) Since $\mathsf{01}$ is a prefix of $\mathsf{0111}$, and $\mathsf{0111} \in B$, we have that $\mathsf{0111} \in C$. Hence $\{\mathsf{0}\}\{\mathsf{10}\}^0\{\mathsf{111}\} = \{\mathsf{0}\}\{\%\}\{\mathsf{111}\} = \{\mathsf{0}\}\{\mathsf{111}\} = \{\mathsf{0111}\} \subseteq C$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $\{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq C$. We must show that $\{\mathsf{0}\}\{\mathsf{10}\}^{n+1}\{\mathsf{111}\} \subseteq C$. Since

$$\begin{aligned}\{\mathsf{0}\}\{\mathsf{10}\}^{n+1}\{\mathsf{111}\} &= \{\mathsf{0}\}\{\mathsf{10}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \\ &= \{\mathsf{01}\}\{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \\ &\subseteq \{\mathsf{01}\}C \qquad\qquad \text{(inductive hypothesis)},\end{aligned}$$

it will suffice to show that $\{\mathsf{01}\}C \subseteq C$. Suppose $w \in \{\mathsf{01}\}C$. We must show that $w \in C$. We have that $w = \mathsf{01}x$ for some $x \in C$. Thus w begins with $\mathsf{01}$. It remains to show that $w \in B$. Since $x \in C$, we have that x begins with $\mathsf{01}$. Thus the first occurrence of $\mathsf{0}$ in $w = \mathsf{01}x$ is followed by $\mathsf{101} \in A$. Furthermore, any other occurrence of $\mathsf{0}$ in $w = \mathsf{01}x$ is within x, and so is followed by an element of A because $x \in C \subseteq B$. Thus $w \in B$.

□

Lemma 3.2.18
$Y \subseteq B$.

Proof. Suppose $w \in Y$. We must show that $w \in B$. If $w = \mathsf{1}$, then $w \in B$. Otherwise, we have that $w = \mathsf{0}x\mathsf{111}$ for some $x \in X$. There are three cases to consider.

- Suppose $x = \mathsf{0}$. Then $w = \mathsf{00111}$ is in B.
- Suppose $x = \mathsf{00}$. Then $w = \mathsf{000111}$ is in B.
- Suppose $x \in \{\mathsf{10}\}^*$. Then $x \in \{\mathsf{10}\}^n$ for some $n \in \mathbb{N}$. By Lemma 3.2.17, we have that $w = \mathsf{0}x\mathsf{111} \in \{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq C \subseteq B$.

□

Lemma 3.2.19
$Y^* \subseteq B$.

Proof. It will suffice to show that, for all $n \in \mathbb{N}$, $Y^n \subseteq B$, and we proceed by mathematical induction.

(Basis Step) Since $\% \in B$, we have that $Y^0 = \{\%\} \subseteq B$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $Y^n \subseteq B$. Then $Y^{n+1} = YY^n \subseteq BB \subseteq B$, by Lemma 3.2.18 and the inductive hypothesis.

□

Lemma 3.2.20
$B \subseteq Y^*$.

Proof. Since $B \subseteq \{\mathsf{0},\mathsf{1}\}^*$, it will suffice to show that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

$$\text{if } w \in B, \text{ then } w \in Y^*.$$

We proceed by strong string induction. Suppose $w \in \{\mathsf{0},\mathsf{1}\}^*$, and assume the inductive hypothesis: for all $x \in \{\mathsf{0},\mathsf{1}\}^*$, if x is a proper substring of w, then

$$\text{if } x \in B, \text{ then } x \in Y^*.$$

We must show that

$$\text{if } w \in B, \text{ then } w \in Y^*.$$

Suppose $w \in B$. We must show that $w \in Y^*$. There are three main cases to consider.

- Suppose $w = \%$. Then $w \in \{\%\} = Y^0 \subseteq Y^*$.
- Suppose $w = \mathsf{0}x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since $w \in B$, the first $\mathsf{0}$ of w must be followed by an element of A. Hence $x \neq \%$, so that there are two cases to consider.
  - Suppose $x = \mathsf{0}y$ for some $y \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{0}x = \mathsf{00}y$. Since $\mathsf{00}y = w \in B$, we have that $y \neq \%$. Thus, there are two cases to consider.
    - ∗ Suppose $y = \mathsf{0}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{00}y = \mathsf{000}z$. Since the first $\mathsf{0}$ in $\mathsf{000}z = w$ is followed by an element of A, and the only element of A that begins with $\mathsf{00}$ is $\mathsf{001}$, we have that $z = \mathsf{1}u$ for some $u \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{0001}u$. Since the second $\mathsf{0}$ in $\mathsf{0001}u = w$ is followed by an element of A, and $\mathsf{011}$ is the only element of A that begins with $\mathsf{01}$, we have that $u = \mathsf{1}v$ for some $v \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{00011}v$. Since the third $\mathsf{0}$ in

  $00011v = w$ is followed by an element of A, and 111 is the only element of A that begins with 11, we have that $v = 1t$ for some $t \in \{0,1\}^*$. Thus $w = 000111t$. Since $00 \in X$, we have that $000111 = (0)(00)(111) \in \{0\}X\{111\} \subseteq Y$. Because t is a suffix of w, it follows that $t \in B$. Thus the inductive hypothesis tells us that $t \in Y^*$. Hence $w = (000111)t \in YY^* \subseteq Y^*$.

  - $*$ Suppose $y = 1z$ for some $z \in \{0,1\}^*$. Thus $w = 00y = 001z$. Since the first 0 in $001z = w$ is followed by an element of A, and the only element of A that begins with 01 is 011, we have that $z = 1u$ for some $u \in \{0,1\}^*$. Thus $w = 0011u$. Since the second 0 in $0011u = w$ is followed by an element of A, and 111 is the only element of A that begins with 11, we have that $u = 1v$ for some $v \in \{0,1\}^*$. Thus $w = 00111v$. Since $0 \in X$, we have that $00111 = (0)(0)(111) \in \{0\}X\{111\} \subseteq Y$. Because v is a suffix of w, it follows that $v \in B$. Thus the inductive hypothesis tells us that $v \in Y^*$. Hence $w = (00111)v \in YY^* \subseteq Y^*$.

- – Suppose $x = 1y$ for some $y \in \{0,1\}^*$. Thus $w = 0x = 01y$. Since $w \in B$, we have that $y \neq \%$. Thus, there are two cases to consider.

  - $*$ Suppose $y = 0z$ for some $z \in \{0,1\}^*$. Thus $w = 010z$. Let u be the longest prefix of z that is in $\{10\}^*$. (Since $\%$ is a prefix of z and is in $\{10\}^*$, it follows that u is well-defined.) Let $v \in \{0,1\}^*$ be such that $z = uv$. Thus $w = 010z = 010uv$.

    Suppose, toward a contradiction, that v begins with 10. Then $u10$ is a prefix of $z = uv$ that is longer than u. Furthermore $u10 \in \{10\}^*\{10\} \subseteq \{10\}^*$, contradicting the definition of u. Thus we have that v does not begin with 10.

    Next, we show that $010u$ ends with 010. Since $u \in \{10\}^*$, we have that $u \in \{10\}^n$ for some $n \in \mathbb{N}$. There are three cases to consider.

    - · Suppose $n = 0$. Since $u \in \{10\}^0 = \{\%\}$, we have that $u = \%$. Thus $010u = 010$ ends with 010.
    - · Suppose $n = 1$. Since $u \in \{10\}^1 = \{10\}$, we have that $u = 10$. Hence $010u = 01010$ ends with 010.
    - · Suppose $n \geq 2$. Then $n - 2 \geq 0$, so that $u \in \{10\}^{(n-2)+2} = \{10\}^{n-2}\{10\}^2$. Hence u ends with 1010, showing that $010u$ ends with 010.

    Summarizing, we have that $w = 010uv$, $u \in \{10\}^*$, $010u$ ends with 010, and v does not begin with 10. Since the second-to-last 0 in $010u$ is followed in w by an element of A, and 101 is the only element of A that begins with 10, we have that $v = 1s$ for some $s \in \{0,1\}^*$. Thus $w = 010u1s$, and $010u1$ ends with 0101. Since the second-to-last symbol of $010u1$ is a 0, we have that

$s \neq \%$. Furthermore, s does not begin with $\mathsf{0}$, since, if it did, then $v = \mathsf{1}s$ would begin with $\mathsf{10}$. Thus we have that $s = \mathsf{1}t$ for some $t \in \{\mathsf{0},\mathsf{1}\}^*$. Hence $w = \mathsf{010}u\mathsf{11}t$. Since $\mathsf{010}u\mathsf{11}$ ends with $\mathsf{011}$, it follows that the last $\mathsf{0}$ in $\mathsf{010}u\mathsf{11}$ must be followed in w by an element of A. Because $\mathsf{111}$ is the only element of A that begins with $\mathsf{11}$, we have that $t = \mathsf{1}r$ for some $r \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{010}u\mathsf{111}r$. Since $(\mathsf{10})u \in \{\mathsf{10}\}\{\mathsf{10}\}^* \subseteq \{\mathsf{10}\}^* \subseteq X$, we have that $\mathsf{010}u\mathsf{111} = (\mathsf{0})((\mathsf{10})u)\mathsf{111} \in \{\mathsf{0}\}X\{\mathsf{111}\} \subseteq Y$. Since r is a suffix of w, it follows that $r \in B$. Thus, the inductive hypothesis tells us that $r \in Y^*$. Hence $w = (\mathsf{010}u\mathsf{111})r \in YY^* \subseteq Y^*$.

$*$ Suppose $y = \mathsf{1}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{011}z$. Since the first $\mathsf{0}$ of w is followed by an element of A, and $\mathsf{111}$ is the only element of A that begins with $\mathsf{11}$, we have that $z = \mathsf{1}u$ for some $u \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{0111}u$. Since $\% \in \{\mathsf{10}\}^* \subseteq X$, we have that $\mathsf{0111} = (\mathsf{0})(\%)(\mathsf{111}) \in \{\mathsf{0}\}X\{\mathsf{111}\} \subseteq Y$. Because u is a suffix of w, it follows that $u \in B$. Thus, since u is a proper substring of w, the inductive hypothesis tells us that $u \in Y^*$. Hence $w = (\mathsf{0111})u \in YY^* \subseteq Y^*$.

- Suppose $w = \mathsf{1}x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since x is a suffix of w, we have that $x \in B$. Because x is a proper substring of w, the inductive hypothesis tells us that $x \in Y^*$. Thus $w = \mathsf{1}x \in YY^* \subseteq Y^*$.

□

By Lemmas 3.2.19 and 3.2.20, we have that $Y^* \subseteq B \subseteq Y^*$, so that $Y^* = B$. This completes our regular expression design and proof of correctness example.

Exercise 3.2.21
Let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathsf{010}$ is not a substring of $w \,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

Exercise 3.2.22
Define $\mathbf{diff} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{Z}$ as in Section 2.2, so that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

$$\mathbf{diff}\, w = \text{the number of } \mathsf{1}\text{'s in } w - \text{the number of } \mathsf{0}\text{'s in } w.$$

Thus $\mathbf{diff}\,\% = 0$, $\mathbf{diff}\,\mathsf{0} = -1$, $\mathbf{diff}\,\mathsf{1} = 1$, and for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\,x + \mathbf{diff}\,y$. Let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{diff}\,w = 3m,$ for some $m \in \mathbb{Z} \,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

Exercise 3.2.23
Define a function $\mathbf{diff} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{Z}$ by: for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

$$\mathbf{diff}\, w = \text{the number of } \mathsf{0}\text{'s in } w - 2(\text{the number of } \mathsf{1}\text{'s in } w).$$

Thus $\mathbf{diff}\, w = 0$ iff w has twice as many $\mathsf{1}$'s as $\mathsf{0}$'s. Furthermore $\mathbf{diff}\, \% = 0$, $\mathbf{diff}\, \mathsf{0} = 1$, $\mathbf{diff}\, \mathsf{1} = -2$, and, for all $x, y \in \{\mathsf{0}, \mathsf{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\, x + \mathbf{diff}\, y$. Let $X = \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid \mathbf{diff}\, w = 0$ and, for all prefixes v of w, $0 \leq \mathbf{diff}\, v \leq 3 \,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

3.2.3 Notes

Our approach in this section is somewhat more formal than is common, but is otherwise standard.

Bibliography

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1991.

[BE93] J. Barwise and J. Etchemendy. *Turing's World 3.0 for Mac: An Introduction to Computability Theory.* Cambridge University Press, 1993.

[BLP$^+$97] A. O. Bilska, K. H. Leider, M. Procopiuc, O. Procopiuc, S. H. Rodger, J. R. Salemme, and E. Tsang. A collection of tools for making automata theory and formal languages come alive. In *Twenty-eighth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 15–19. ACM Press, 1997.

[End77] H. B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, second edition, 2001.

[HR00] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, pages 6–10. ACM Press, 2000.

[Jon97] N. J. Jones. *Computability and Complexity: From a Programming Perspective.* The MIT Press, 1997.

[Koz97] D. C. Kozen. *Automata and Computability.* Springer-Verlag, 1997.

[Lei10] H. Leiß. The Automata Library. `http://www.cis.uni-muenchen.de/~leiss/sml-automata.html`, 2010.

[Lin01] P. Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, 2001.

[LP98] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, second edition, 1998.

[LRGS04] S. Lombardy, Y. Régis-Gianas, and J. Sakarovitch. Introducing vaucanson. *Theoretical Computer Science*, 328:77–96, 2004.

[Mar91] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw Hill, second edition, 1991.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised 1997*. The MIT Press, 1997.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[RHND99] M. B. Robinson, J. A. Hamshar, J. E. Novillo, and A. T. Duchowski. A Java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In *Thirtieth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 105–109. ACM Press, 1999.

[Rod06] S. H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, 2006.

[RW94] D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[RWYC17] D. Raymond, D. Wood, S. Yu, and C. Câmpeanu. Grail+: A symbolic computation environment for finite-state machines, regular expressions, and finite languages. `http://www.csit.upei.ca/~ccampeanu/Grail/`, 2017.

[Sar02] J. Saraiva. HaLeX: A Haskell library to model, manipulate and animate regular languages. In *ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)*, Pittsburgh, October 2002.

[Sto05] A. Stoughton. Experimenting with formal languages. In *Thirty-sixth ACM SIGCSE Technical Symposium on Computer Science Education*, page 566. ACM Press, 2005.

[Sto08] A. Stoughton. Experimenting with formal languages using Forlan. In *FDPE '08: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 41–50, New York, NY, USA, 2008. ACM.

[Sut92] K. Sutner. Implementing finite state machines. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 15, pages 347–363. American Mathematical Society, 1992.

[Ull98] J. D. Ullman. *Elements of ML Programming: ML97 Edition*. Prentice Hall, 1998.

Index