An Introduction to Formal Language Theory that Integrates Experimentation and Proof

Allen Stoughton
Kansas State University

Draft of Fall 2004

The LaTeX source of this book and associated lecture slides, and the distribution of the Forlan toolset are available on the WWW at `http://www.cis.ksu.edu/~allen/forlan/`.

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 are set 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 many applications to computer science, e.g., to compiler construction, most computer science programs offer both undergraduate and graduate courses in this subject. Many of the results of formal language theory are proved constructively, using algorithms that are useful in practice. In typical courses on formal language theory, students apply these algorithms to toy examples by hand, and learn how they are used in applications. But they are not able to experiment with them on a larger scale.

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 an automaton that accepts some language, L. With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (perhaps) prove that it is 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 (union, intersection, difference, concatenation, closure). He or she 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. 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

Over the past several years, I have been designing and developing a computer toolset, called Forlan, for experimenting with formal languages. Forlan is implemented in the functional programming language Standard ML [MTHM97, Pau96], a language whose notation and concepts are similar to those of mathematics. Forlan is used interactively. In fact, a Forlan session is simply a Standard ML session in which the Forlan modules are pre-loaded. Users are able to extend Forlan by defining ML functions.

In Forlan, the usual objects of formal language theory—automata, regular expressions, grammars, labeled paths, parse trees, etc.—are defined as abstract types, and have concrete syntax. The standard algorithms of formal language theory are implemented in Forlan, including conversions between different kinds of automata and grammars, the usual operations on automata and grammars, equivalence testing and minimization of deterministic finite automata, etc. Support for the variant of the programming language Lisp that we use (instead of Turing machines) as a universal programming language is planned.

While developing Forlan, I have also been writing lectures notes on formal language theory that are based around Forlan, and this book is the outgrowth of those notes. I am attempting to keep the conceptual and notational distance between the textbook and toolset as small as possible. The book treats each concept or algorithm both theoretically, especially using proof, and through experimentation, using Forlan. Special proofs that are carried out assuming the correctness of Forlan's implementation are labeled "[Forlan]", and theorems that are *only* proved in this way are also so-labeled.

Readers of this book are assumed to have a significant amount of 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 core sections of the book assume no previous knowledge of Standard ML. Eventually, advanced sections covering the implementation of Forlan will be written, and these sections will assume the kind of familiarity with Standard ML that could be obtained by reading [Pau96] or [Ull98].

Outline of the Book

The book consists of five chapters. Chapter 1, *Mathematical Background*, consists of the material on set theory, induction principles for the natural numbers, 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, introduce and show how to use several string induction principles, 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, four 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 and lexical analysis.

In Chapter 4, *Context-free Languages*, we study context-free grammars and languages, algorithms for processing grammars and for converting regular expressions and finite automata to grammars, and properties of context-free languages. It turns out that the set of all context-free languages is a proper superset of the set of all regular languages.

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study a universal programming language based on Lisp, which we use 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. It turns out that the context-free languages are a proper subset of the recursive languages, that the recursive languages are a proper subset of the recursively enumerable languages, and that there are languages that are not recursively enumerable. Furthermore, there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Further Reading and Related Work

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

The existing formal language toolsets fit into two categories. In the first category are tools like JFLAP [BLP$^+$97, HR00], Pâté [BLP$^+$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 second category consists of toolsets that, like Forlan, are embedded in programming languages, and so that support sophisticated experimentation with formal languages. Toolsets in this category include Automata [Sut92], Grail+ [Yu02], HaLeX [Sar02] and Leiß's Automata Library [Lei00]. I am not aware of any other textbook/toolset packages whose toolsets are members of this second category.

Acknowledgments

It is a pleasure to acknowledge helpful conversations or e-mail exchanges relating to this textbook/toolset project with Brian Howard, Rodney Howell, John Hughes, Nathan James, Patrik Jansson, Jace Kohlmeier, Dexter Kozen, Aarne Ranta, Ryan Stejskal and Colin Stirling. Some of this work was done while I was on sabbatical at the Department of Computing Science of the University of Chalmers.

Chapter 1

Mathematical Background

This chapter consists of the material on set theory, induction principles for the natural numbers, and trees and inductive definitions that will be required in the later chapters.

1.1 Basic Set Theory

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

Let's begin by establishing notation for the standard 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.

Next, we say when one set is a subset of another set, as well as when two sets are equal. 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 *equal* to B ($A = B$) iff $A \subseteq B$ and $B \subseteq A$;
- 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, A is equal to B iff A and B have the same elements, 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, by the definition of set equality, 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.

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 $A \supsetneq C$.

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, $5.5 \notin A$, since $5.5 \notin \mathbb{N}$, and $4 \notin A$, 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, 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 $2, 1 \geq 1$.

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

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

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 a zero for intersection ($\emptyset \cap A = \emptyset = A \cap \emptyset$, for all sets A). $A - B$ 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, $\{0, 1\} \times \{1, 2\} = \{(0, 1), (0, 2), (1, 1), (1, 2)\}$. If A and B have n and m elements, respectively, 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, then $\mathcal{P}(A)$ will have 2^n elements.

We can also form products of three or more sets. For example, we write $A \times B \times C$ for the set of all ordered triples (x, y, z) such that $x \in A$, $y \in B$ and $z \in C$.

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.

We must 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 $x \in A \cap (B \cup C)$.

$\square$

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

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

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 x of our universe that are in all of the nonexistent elements of $\emptyset$, i.e., it would contain all elements of our universe. It turns out, however, that there is no such set, which is why we may only take generalized intersections of non-empty sets.

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 $\{0,1,2\}$.

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 $\{\,(x,z) \mid (x,y) \in R \text{ and } (y,z) \in S,$ for some

$y\,\}$. For example, if $R = \{(1,1),(1,2),(2,3)\}$ and $S = \{(2,3),(2,4),(3,4)\}$, then $S \circ R = \{(1,3),(1,4),(2,4)\}$.

It is easy to show, roughly speaking, 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 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 $\{(1,0),(2,1),(3,1)\}$.

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 $(y,x) \in R$;
- a *function* iff, for all x, y, z, if $(x,y) \in R$ and $(x,z) \in R$, then $y = z$.

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

- R is not reflexive on $\{0,1,2\}$, since $(0,0) \notin R$.
- 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 $(1,0) \notin R$.
- R a 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.

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 $\mathbf{range}(f) \subseteq Y$. We write $X \to Y$ for the set of all functions from X to Y.

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 $f \in \{0, 1, 2\} \to \{0, 1, 2\}$.

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 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 worked, we have, roughly speaking, 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.

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:

$$\begin{tikzcd}
 & f & \\
0 \arrow[rrdd] & & 0.5 \\
1 \arrow[rr] & & 2.6 \\
2 \arrow[rruu] & & 5.1
\end{tikzcd}$$

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. It's not hard to show that for all sets X, Y, Z:

(1) $X \cong X$;

(2) If $X \cong Y \cong Z$, then $X \cong Z$;

(3) If $X \cong Y$, then $Y \cong X$.

E.g., consider (2). By the assumptions, we have that there is a bijection f from X to Y, and there is a bijection g from Y to Z. Then $g \circ f$ is a bijection from X to Z, showing that $X \cong Z$.

We say that a set X is:

- *finite* iff $X \cong \{1, \ldots, n\}$, for some $n \in \mathbb{N}$;
- *infinite* iff it is not finite;
- *countably infinite* iff $X \cong \mathbb{N}$;
- *countable* iff X is either finite or countably infinite;
- *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;
- 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, one uses an important technique called "diagonalization", which we will see again in Chapter 5. 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. Since $\mathcal{P}(\mathbb{N})$ is not finite, it follows that there is a bijection f from $\mathbb{N}$ to

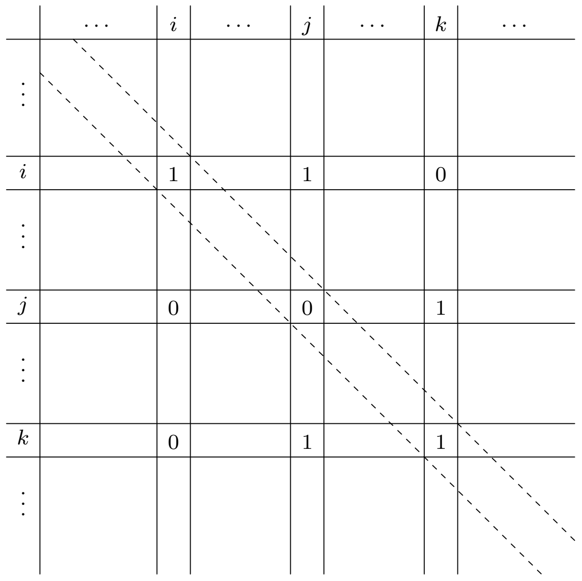


Figure 1.1: Example Diagonalization Table for Cardinality Proof

$\mathcal{P}(\mathbb{N})$. 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 (n, m) 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 $n \in \mathbb{N}$, $X \neq f(n)$.

With our example table:

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

We conclude this section by turning the above ideas into a shorter, but more opaque, proof that:

Proposition 1.1.2
$\mathcal{P}(\mathbb{N})$ is uncountable.

Proof. Suppose, toward a contradiction, that $\mathcal{P}(\mathbb{N})$ is countable. Thus, 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 $n \in X$. Because $X = f(n)$, we have that $n \in f(n)$. Hence, by the definition of X, it follows that $n \not\in X$—contradiction.

- Suppose $n \not\in X$. Because $X = f(n)$, we have that $n \not\in f(n)$. Hence, by the definition of X, it follows that $n \in X$—contradiction.

Since we obtained a contradiction in both cases, we have an overall contradiction. □

We have seen how bijections may be used to determine whether sets have the same size. But how can one 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.

A function f is an *injection* (or is *injective*) iff, for all x, y, z, if $(x, z) \in f$ and $(y, z) \in f$, then $x = y$. I.e., 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). Of course, if f is a bijection from X to Y, then f is injective.

We say that a set X is *dominated by* a set Y ($X \preceq Y$) iff there is an injective function whose domain is X and whose range is a subset of Y. For example, the injection $\mathbf{id}_{\mathbb{N}}$ shows that $\mathbb{N} \preceq \mathbb{R}$.

It's not hard to show that for all sets X, Y, Z:

(1) $X \preceq X$;

(2) If $X \preceq Y \preceq Z$, then $X \preceq Z$.

Clearly, if $X \cong Y$, then $X \preceq Y \preceq X$. A famous result of set theory, called the Schröder-Bernstein Theorem, says that, for all sets X and Y, if $X \preceq Y \preceq X$, then $X \cong Y$. And, one of the forms of the famous Axiom of Choice says that, for all sets X and Y, either $X \preceq Y$ or $Y \preceq X$. Finally, the sizes or cardinalities of sets are ordered in such a way that, for all sets X and Y, $|X| \leq |Y|$ iff $X \preceq Y$.

Given the above machinery, one can generalize Proposition 1.1.2 into Cantor's Theorem, which says that, for all sets X, $|X|$ is strictly smaller than $|\mathcal{P}(X)|$.

1.2 Induction Principles for the Natural Numbers

In this section, we consider two methods for proving that every natural number n has some property $P(n)$. The first method is the familiar principle of mathematical induction. The second method is the principle of strong (or course-of-values) induction.

The *principle of mathematical induction* says that

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

follows from showing

- (basis step)

$$P(0);$$

- (inductive step)

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

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

Let's consider a simple example of mathematical induction, involving the iterated composition of a function with itself. The nth composition f^n of a function $f \in A \to A$ with itself is defined by recursion:

$$\begin{aligned} f^0 &= id_A, \text{ for all sets } A \text{ and } f \in A \to A; \\ f^{n+1} &= f \circ f^n, \text{ for all sets } A, f \in A \to A \text{ and } n \in \mathbb{N}. \end{aligned}$$

Thus, if $f \in A \to A$, then $f^0 = \mathbf{id}_A$, $f^1 = f \circ f^0 = f \circ \mathbf{id}_A = f$, $f^2 = f \circ f^1 = f \circ f$, etc. For example, if f is the function from $\mathbb{N}$ to $\mathbb{N}$ that adds two to its input, then $f^n(m) = m + 2n$, for all $n, m \in \mathbb{N}$.

Proposition 1.2.1
For all $n, m \in \mathbb{N}$, $f^{n+m} = f^n \circ f^m$.

In other words, the proposition says that running a function $n+m$ times will produce the same result as running it m times, and then running it n times. For the proof, we have to begin by figuring whether we should do induction on n or m or both (one induction inside the other). It turns out that we can prove our result by fixing m, and then doing induction on n. Readers should consider whether another approach will work.

Proof. Suppose $m \in \mathbb{N}$. We use mathematical induction to show that, for all $n \in \mathbb{N}$, $f^{n+m} = f^n \circ f^m$. (Thus, our property $P(n)$ is "$f^{n+m} = f^n \circ f^m$".)

(Basis Step) We have that $f^{0+m} = f^m = id_A \circ f^m = f^0 \circ f^m$.

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

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

□

The *principle of strong induction* says that

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

follows from showing

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

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

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

Proposition 1.2.2
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. $\square$

It is easy to see that any proof using mathematical induction can be turned into one using strong induction. (Split into the cases where $n = 0$ and $n = m + 1$, for some m.)

Are there results that can be proven using strong induction but not using mathematical induction? The answer turns out to be "no". In fact, a proof using strong induction can be mechanically turned into one using mathematical induction, but at the cost of making the property $P(n)$ more complicated. Challenge: find a $P(n)$ that can be used to prove Lemma 1.2.2 using mathematical induction. (Hint: make use of the technique of the following proposition.)

As a matter of style, one should use mathematical induction whenever it is *convenient* to do so, since it is the more straightforward of the two principles.

Given the preceding claim, it's not surprising that we can prove the validity of the principle of strong induction using only mathematical induction:

Proposition 1.2.3
Suppose $P(n)$ is a property, and

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

Then

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

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

> 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

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

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

(Basis Step) 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

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

  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

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

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

We conclude this section by showing one more proof using strong induction. Define $f \in \mathbb{N} \to \mathbb{N}$ by: for all $n \in \mathbb{N}$,

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ 0 & \text{if } n = 1, \\ n + 1 & \text{if } n > 1 \text{ and } n \text{ is odd.} \end{cases}$$

Proposition 1.2.4
For all $n \in \mathbb{N}$, there is an $l \in \mathbb{N}$ such that $f^l(n) = 0$.

In other words, the proposition says that, for all $n \in \mathbb{N}$, one can get from n to 0 by running f some number of times.

Proof. We use strong induction to show that, for all $n \in \mathbb{N}$, there is an $l \in \mathbb{N}$ such that $f^l(n) = 0$. Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: for all $m \in \mathbb{N}$, if $m < n$, then there is an $l \in \mathbb{N}$ such that $f^l(m) = 0$. We must show that there is an $l \in \mathbb{N}$ such that $f^l(n) = 0$. There are four cases to consider.

($n = 0$) We have that $f^0(n) = \mathbf{id}_{\mathbb{N}}(0) = 0$.

($n = 1$) We have that $f^1(n) = f(1) = 0$.

($n > 1$ and n is even) Since n is even, we have that $n = 2i$, for some $i \in \mathbb{N}$. And, because $2i = n > 1$, we can conclude that $i \geq 1$. Hence $i < i+i$, with the consequence that

$$\frac{n}{2} = \frac{2i}{2} = i < i + i = 2i = n.$$

Hence $n/2 < n$. Thus, by the inductive hypothesis, it follows that there is an $l \in \mathbb{N}$ such that $f^l(n/2) = 0$. Hence,

$$\begin{aligned}
f^{l+1}(n) &= (f^l \circ f^1)(n) && \text{(Proposition 1.2.1)} \\
&= f^l(f(n)) \\
&= f^l(n/2) && \text{(definition of } f(n)\text{, since } n \text{ is even)} \\
&= 0.
\end{aligned}$$

($n > 1$ and n is odd) Since n is odd, we have that $n = 2i + 1$, for some $i \in \mathbb{N}$. And, because $2i + 1 = n > 1$, we can conclude that $i \geq 1$. Hence $i + 1 < i + i + 1$, with the consequence that

$$\frac{n+1}{2} = \frac{(2i+1)+1}{2} = \frac{2i+2}{2} = \frac{2(i+1)}{2} = i + 1 < i + i + 1 = 2i + 1 = n.$$

Hence $(n + 1)/2 < n$. Thus, by the inductive hypothesis, there is an $l \in \mathbb{N}$ such that $f^l((n + 1)/2) = 0$. Hence,

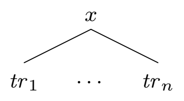
$$\begin{aligned}
f^{l+2}(n) &= (f^l \circ f^2)(n) && \text{(Proposition 1.2.1)} \\
&= f^l(f(f(n))) \\
&= f^l(f(n+1)) && \text{(definition of } f(n)\text{, since } n > 1 \text{ and } n \text{ is odd)} \\
&= f^l((n+1)/2) && \text{(definition of } f(n+1)\text{, since } n+1 \text{ is even)} \\
&= 0.
\end{aligned}$$

□

1.3 Trees and Inductive Definitions

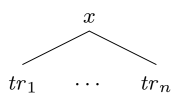
In this section, we will introduce and study ordered trees of arbitrary (finite) arity whose nodes are labeled by elements of some set. The definition of the set of such trees will be our first example of an inductive definition. In later chapters, we will define regular expressions (in Chapter 3) and parse trees (in Chapter 4) as restrictions of the trees we consider here.

Suppose X is a set. The set $\mathbf{Tree}_X$ of *X-trees* is the least set such that, (†) for all $x \in X$, $n \in \mathbb{N}$ and $tr_1, \ldots, tr_n \in \mathbf{Tree}_X$,

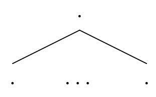


$\in$ $\mathbf{Tree}_X$.

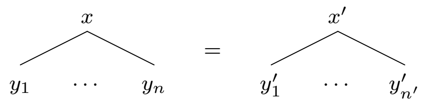
The *root label* of the tree



is x, and tr_1 is the tree's first *child*, etc. We are treating



as a constructor, so that



iff $x = x'$, $n = n'$, $y_1 = y'_1$, $\ldots$, $y_n = y'_{n'}$.

When we say that $\mathbf{Tree}_X$ is the "least" set satisfying property (†), we mean least with respect to $\subseteq$. I.e., we are saying that $\mathbf{Tree}_X$ is the unique set such that:

- $\mathbf{Tree}_X$ satisfies property (†); and
- if A is a set satisfying property (†), then $\mathbf{Tree}_X \subseteq A$.

In other words:

- $\mathbf{Tree}_X$ satisfies (†) and doesn't contain any extraneous elements; and
- $\mathbf{Tree}_X$ consists of precisely those values that can be constructed in some number of steps using (†).

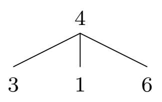
The definition of $\mathbf{Tree}_X$ is our first example of an *inductive definition*, a definition in which we collect together all of the values that can be constructed using some set of rules.

Here are some example elements of $\mathbf{Tree}_{\mathbb{N}}$:

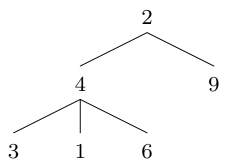
- (remember that n can be 0)

  3

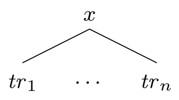
-



•



We sometimes use linear notation for trees, writing an X-tree

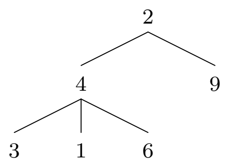


as

$$x(tr_1, \ldots, tr_n).$$

We often abbreviate $x()$ (the childless tree whose root label is x) to x.

For example, we can write the $\mathbb{N}$-tree



as $2(4(3,1,6),9)$.

Every inductive definition gives rise to an induction principle, and the definition of $\mathbf{Tree}_X$ is no exception. The *induction principle for* $\mathbf{Tree}_X$ says that

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

follows from showing

for all $x \in X$, $n \in \mathbb{N}$ and $tr_1, \ldots, tr_n \in \mathbf{Tree}_X$,
if (†) $P(tr_1), \ldots, P(tr_n)$,
then $P(x(tr_1, \ldots, tr_n))$.

We refer to (†) as the inductive hypothesis.

When we draw a tree, we can point at a position in the drawing and call it a *node*. The formal analogue of this graphical notion is called a path. The set **Path** of *paths* is the least set such that

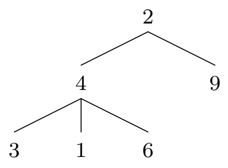
- $\mathbf{nil} \in \mathbf{Path}$;
- For all $n \in \mathbb{N}$ and pat in **Path**, $n \rightarrow pat \in \mathbf{Path}$.

(Here, **nil** and $\rightarrow$ are constructors, which tells us when paths are equal.) A path

$$n_1 \rightarrow \cdots \rightarrow n_l \rightarrow \mathbf{nil},$$

consists of directions to a node in the drawing of a tree: one starts at the *root node* of a tree, goes from there to the n_1'th child, …, goes from there to the n_l'th child, and then stops.

Some examples of paths and corresponding nodes for the $\mathbb{N}$-tree



are:

- **nil** corresponds to the node labeled 2;
- $1 \rightarrow \mathbf{nil}$ corresponds to the node labeled 4;
- $1 \rightarrow 2 \rightarrow \mathbf{nil}$ corresponds to the node labeled 1.

We consider a path pat to be *valid* for a tree tr iff following the directions of pat never causes us to try to select a nonexistent child. E.g., the path $1 \rightarrow 2 \rightarrow nil$ isn't valid for the tree $6(7(8))$, since the tree $7(8)$ lacks a second child.

As usual, if the sub-tree at position pat in tr has no children, then we call the sub-tree's root node a *leaf* or *external node*; otherwise, the sub-tree's root node is called an *internal node*. Note that we can form a tree tr' from a tree tr by replacing the sub-tree at position pat in tr by a tree tr''.

We define the *size* of an X-tree tr to be the number of elements of

$$\{\, pat \mid pat \text{ is a valid path for } tr \,\}.$$

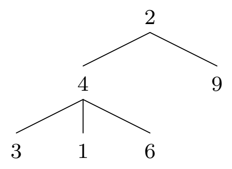
The *length* of a path *pat* ($|pat|$) is defined recursively by:

$$\begin{aligned} |\mathbf{nil}| &= 0; \\ |n \to pat| &= 1 + |pat|, \text{ for all } n \in \mathbb{N} \text{ and } pat \in \mathbf{Path}. \end{aligned}$$

Given this definition, we can define the *height* of an X-tree *tr* to be the largest element of

$$\{\, |pat| \mid pat \text{ is a valid path for } tr \,\}.$$

For example, the tree



has:

- size 6, since exactly six paths are valid for this tree; and
- height 2, since the path $1 \to 1 \to \mathbf{nil}$ is valid for this tree and has length 2, and there are no paths of greater length that are valid for this tree.

Chapter 2

Formal Languages

In this chapter, we say what symbols, strings, alphabets and (formal) languages are, introduce several string induction principles, and give an introduction to the Forlan toolset.

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 subsequent chapters, we will study four more restricted kinds of languages: the regular (Chapter 3), context-free (Chapter 4), recursive and recursively enumerable (Chapter 5) 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, then we wouldn't be able to have notation in Forlan for all strings and symbols. These considerations lead us to the following definition.

A *symbol* is one of the following finite sequences of ASCII characters:

- One of the *digits* $\mathsf{0}$–$\mathsf{9}$;
- One of the *upper case letters* A–Z;
- One of the *lower case letters* a–z;
- A $\langle$, followed by any finite sequence of printable ASCII characters in which $\langle$ and $\rangle$ are properly nested, followed by a $\rangle$.

For example, $\langle\mathsf{id}\rangle$ and $\langle\langle\mathsf{a}\rangle\mathsf{b}\rangle$ are symbols. On the other hand, $\langle\mathsf{a}\rangle\rangle$ is not a symbol since $\langle$ and $\rangle$ are not properly nested in $\mathsf{a}\rangle$.

Whenever possible, we will use the mathematical variables a, b and c to name symbols. To avoid confusion, we will try to avoid situations in which we must simultaneously use, e.g., the symbol a and the mathematical variable a.

We write **Sym** for the set of all symbols. We order **Sym** by length (number of ASCII characters) 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? To see that the answer is "yes", let's first see that it is possible to enumerate (list in some order, without repetition) all of the finite sequences of ASCII characters. We can list these sequences first according to length, and then according to lexicographic order. Thus the set of all such sequences is countably infinite. And since every symbol is such a sequence, it follows that **Sym** is countably infinite, too.

Now that we know what symbols are, we can define strings in the standard way. A *string* is a finite sequence of symbols. We write the string with no symbols (the *empty string*) as %, instead of the conventional ϵ, since this symbol can also be used in Forlan. Some other examples of strings are ab, $\mathsf{0110}$ and $\langle\mathsf{id}\rangle\langle\mathsf{num}\rangle$. Whenever possible, we will use the mathematical variables u, v, w, x, y and z to name strings.

The *length* of a string x ($|x|$) is the number of symbols in the string. For example: $|\%| = 0$, $|\mathsf{ab}| = 2$, $|\mathsf{0110}| = 4$ and $|\langle\mathsf{id}\rangle\langle\mathsf{num}\rangle| = 2$.

We write **Str** for 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 is a finite sequence of ASCII characters, it follows that **Str** is countably infinite.

The *concatenation* of strings x and y ($x \mathbin{@} y$) is the string consisting of the symbols of x followed by the symbols of y. For example, $\% \mathbin{@} \mathsf{abc} = \mathsf{abc}$ and $\mathsf{01} \mathbin{@} \mathsf{10} = \mathsf{0110}$. Concatenation is associative: for all $x, y, z \in \mathbf{Str}$,

$$(x \mathbin{@} y) \mathbin{@} z = x \mathbin{@} (y \mathbin{@} z).$$

And, % is the identify for concatenation: for all $x \in \mathbf{Str}$,

$$\% \mathbin{@} x = x \mathbin{@} \% = x.$$

We often abbreviate $x \mathbin{@} y$ to xy. This abbreviation introduces some harmless ambiguity. For example, all of $\mathsf{0} \mathbin{@} \mathsf{10}$, $\mathsf{01} \mathbin{@} \mathsf{0}$ and $\mathsf{0} \mathbin{@} \mathsf{1} \mathbin{@} \mathsf{0}$ are abbreviated to $\mathsf{010}$. Fortunately, all of these expressions have the same value, so this kind of ambiguity is not a problem.

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

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

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

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

□

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 $u \in \mathbf{Str}$;
- 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;
- ab is a (non-proper) prefix, suffix and substring of ab.

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 $\mathbf{alphabet} \in \mathbf{Str} \to \mathbf{Alp}$ by *right* recursion on strings:

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

(We would have called it *left* recursion, if the recursive call had been $\mathbf{alphabet}(xa) = \{a\} \cup \mathbf{alphabet}(x)$.) I.e., $\mathbf{alphabet}(w)$ consists of all of the symbols occurring in the string w. E.g., $\mathbf{alphabet}(\mathsf{01101}) = \{\mathsf{0}, \mathsf{1}\}$. We say that $\mathbf{alphabet}(x)$ is the *alphabet of x*.

If Σ is an alphabet, then we write Σ^* for

$$\{\, w \in \mathbf{Str} \mid \mathbf{alphabet}(w) \subseteq \Sigma \,\}.$$

I.e., Σ^* consists of all of the strings that can be built using the symbols of Σ. For example, the elements of $\{\mathsf{0}, \mathsf{1}\}^*$ are:

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

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

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

(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 $\mathcal{P}(\{\mathsf{0},\mathsf{1}\}^*)$, the set of all $\{\mathsf{0},\mathsf{1}\}$-languages, has the same size as $\mathcal{P}(\mathbb{N})$, and is thus uncountable.

Given a language L, we write $\textbf{alphabet}(L)$ for the *alphabet*

$$\bigcup\{\, \textbf{alphabet}(w) \mid w \in L\,\}.$$

of L. I.e., $\textbf{alphabet}(L)$ consists of all of the symbols occurring in the strings of L. For example,

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

If A is an infinite subset of **Sym** (and so is not an alphabet), we allow ourselves to write A^* for

$$\{\, x \in \textbf{Str} \mid \textbf{alphabet}(x) \subseteq A\,\}.$$

I.e., A^* consists of all of the strings that can be built using the symbols of A. For example, $\textbf{Sym}^* = \textbf{Str}$.

2.2 String Induction Principles

In this section, we introduce three string induction principles: left string induction, right string induction and strong string induction. These induction principles are ways of showing that every string $w \in A^*$ 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 $A^* = \mathbf{Str}$.

The first two of our string induction principles are similar to mathematical induction, whereas the third principle is similar to strong induction. In fact, we could easily turn proofs using the first two string induction principles into proofs by mathematical induction on the length of w, and could turn proofs using the third string induction principle into proofs using strong induction on the length of w.

In this section, we will also see two more examples of how inductive definitions give rise to induction principles.

Suppose $A \subseteq \mathbf{Sym}$. The *principle of left string induction* for A says that

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

follows from showing

- (basis step)
$$P(\%);$$
- (inductive step)
$$\text{for all } a \in A \text{ and } w \in A^*, \text{ if } \ (\dagger)\ P(w), \text{then } P(wa).$$

We refer to the formula (†) as the *inductive hypothesis*. This principle is called "left" string induction, because w is on the left of wa.

In other words, to show that every $w \in A^*$ has property P, we show that the empty string has property P, assume that $a \in A$, $w \in A^*$ and that (the inductive hypothesis) w has property P, and then show that wa has property P.

By switching wa to aw in the inductive step, we get the principle of right string induction. Suppose $A \subseteq \mathbf{Sym}$. The *principle of right string induction* for A says that

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

follows from showing

- (basis step)

$$P(\%);$$

- (inductive step)

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

Before going on to strong string induction, we look at some examples of how left/right string induction can be used. We define the *reversal* x^R of a string x by right recursion on strings:

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

Thus, e.g., $(\mathsf{021})^R = \mathsf{120}$. 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 2.2.1
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 string 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,

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

□

Proposition 2.2.2
For all $x \in \mathbf{Str}$, $(x^R)^R = x$.

Proof. Follows by an easy right string induction, making use of Proposition 2.2.1. $\square$

In Section 2.1, we used right string recursion to define the function $\mathbf{alphabet} \in \mathbf{Str} \to \mathbf{Alp}$. Thus, we can use right string induction to show that:

Proposition 2.2.3
For all $x, y \in \mathbf{Str}$, $\mathbf{alphabet}(xy) = \mathbf{alphabet}(x) \cup \mathbf{alphabet}(y)$.

Now we come to the string induction principle that is analogous to strong induction. Suppose $A \subseteq \mathbf{Sym}$. The *principle of strong string induction* for A says that

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

follows from showing

for all $w \in A^*$,
if (‡) for all $x \in A^*$, if $|x| < |w|$, then $P(x)$,
then $P(w)$.

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

In other words, to show that every $w \in A^*$ has property P, we let $w \in A^*$, and assume (the inductive hypothesis) that every $x \in A^*$ that is strictly shorter than w has property P. Then, we must show that w has property P.

Let's consider a first—and very simple—example of strong string induction. Let X be the least subset of $\{\mathsf{0}, \mathsf{1}\}^*$ such that:

(1) $\% \in X$;

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

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

$$\%, \mathsf{00}, \mathsf{11}, \mathsf{0000}, \mathsf{0110}, \mathsf{1001}, \mathsf{1111}, \ldots$$

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

Lemma 2.2.4
$Y \subseteq X$.

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 $|x| < |w|$, 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 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 $|w| \geq 2$, we have that $w = axb$ for some $a, b \in \{0,1\}$ and $x \in \{0,1\}^*$. And, $|x|$ 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 $|x| < |w|$, 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 $w = axa \in X$. □

Lemma 2.2.5
$X \subseteq Y$.

We could prove this lemma by strong string induction. But it is simpler and more elegant to use an alternative approach. The inductive definition of X gives rise to the following induction principle. The *principle of induction on X* says that

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

follows from showing

(1)

$$P(\%)$$

(by Part (1) of the definition of X, $\% \in X$, and thus we should expect to have to show $P(\%)$);

(2)

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

(by Part (2) of the definition of X, if $a \in \{\mathtt{0},\mathtt{1}\}$ and $x \in X$, then $axa \in X$; when proving that the "new" element axa has property P, we're allowed to assume that the "old" element x has the property).

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

We will use induction on X to prove Lemma 2.2.5.

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 % 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: $x \in Y$. 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 2.2.6
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.4 and 2.2.5. □

We end this section by proving a more complex proposition concerning a "difference" function on strings, which we will use a number of times in later chapters. Given a string $w \in \{\mathtt{0},\mathtt{1}\}^*$, we write $\mathbf{diff}(w)$ for

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

Then:

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

Note that, for all $w \in \{\mathtt{0},\mathtt{1}\}^*$, $\mathbf{diff}(w) = 0$ iff w has an equal number of $\mathtt{0}$'s and $\mathtt{1}$'s.

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

(1) $\% \in X$;

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

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

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

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

For example, since $\% \in X$, it follows, by (3) and (4) that $\mathtt{01} = \mathtt{0}\%\mathtt{1} \in X$ and $\mathtt{10} = \mathtt{1}\%\mathtt{0} \in X$. Thus, by (2), we have that $\mathtt{0110} = (\mathtt{01})(\mathtt{10}) \in X$. And, Y consists of all strings of $\mathtt{0}$'s and $\mathtt{1}$'s with an equal number of $\mathtt{0}$'s and $\mathtt{1}$'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 $\mathtt{0}$'s and $\mathtt{1}$'s; and (the hard direction) that every string of $\mathtt{0}$'s and $\mathtt{1}$'s with an equal number of $\mathtt{0}$'s and $\mathtt{1}$'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 2.2.7
$X \subseteq Y$.

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 $\% \in \{\mathtt{0},\mathtt{1}\}^*$ and $\mathbf{diff}(\%) = 0$, we have that $\% \in Y$.

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

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

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

□

Lemma 2.2.8
$Y \subseteq X$.

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 $|x| < |w|$, 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.

- ($w = \%$) Then $w = \% \in X$, by Part (1) of the definition of X.
- ($w = 0t$ 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 $|y| < |w|$ and $|z| < |w|$, 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 $w = 0y1z = (0y1)z \in X$.

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

  Let u be the shortest prefix of t such that $\mathbf{diff}(u) \leq -1$. (Since t is a prefix of itself and $\mathbf{diff}(t) = -1 \leq -1$, 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 $\mathbf{diff}(y) \geq 0$.

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

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

  Because $|y| < |w|$ and $|z| < |w|$, 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 $w = 1y0z = (1y0)z \in X$.

$\square$

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 2.2.9
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.7 and 2.2.8. $\square$

2.3 Introduction to Forlan

The Forlan toolset is implemented as a set of Standard ML (SML) modules. It's used interactively. In fact, a Forlan session is nothing more than a Standard ML session in which the Forlan modules are available.

Instructions for installing Forlan on machines running Linux and Windows can be found on the WWW at `http://www.cis.ksu.edu/~allen/forlan/`.

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

To invoke Forlan under Linux, type the command `forlan`:

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

To invoke Forlan under Windows, (double-)click on the Forlan icon.

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

The simplest way of using SML is as a calculator:

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

SML responds to each expression by printing its value and type, and noting that the expression's value has been bound to the identifier `it`. Expressions must be terminated with semicolons.

SML also has the types `string` and `bool`, as well as product types $t_1 * \cdots * t_n$, whose values consist of n-tuples:

```
- "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
```

The operator `^` is string concatenation.

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
```

One can even give names to the components of a tuple:

```
- val (x, y, z) = (3 + 1, 4 = 4, "a" ^ "b");
val x = 4 : int
val y = true : bool
val z = "ab" : string
```

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

```
- fun f n = n + 1;
val f = fn : int -> int
- f 3;
val it = 4 : int
- f(4 + 5);
val it = 10 : 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` maps its input n to its output $n + 1$. 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. Note that SML infers the types of functions, and that the type operator `*` has higher precedence than the operator `->`. When applying a funtion to a single argument, the argument may be enclosed in parentheses, but doesn't have to be parenthesized.

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
```

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

One can load the contents of a file into SML 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 values of an option type t `option` are built using the type's two constructors: `NONE` of type t `option`, and `SOME` of type t `->` t `option`. So, 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`.

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`)`.

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

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

These 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` compares two symbols, yielding `LESS`, `EQUAL` or `GREATER`.

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`. Whitespace characters are ignored by Forlan's input routines. Interactive input is terminated by a line consisting of a single "`.`" (dot, period). Forlan's 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 future, 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 "";
@ <id>
@ .
val a = - : sym
- val b = Sym.input "";
@ <num>
@ .
val b = - : sym
- Sym.output("", a);
<id>
val it = () : unit
- Sym.compare(a, b);
val it = LESS : order
```

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

Expressions in SML are evaluated from left to right, which explains why the following transcript results in the value `GREATER`, rather than `LESS`:

```
- Sym.compare(Sym.input "", Sym.input "");
@ <can>
@ .
@ <be>
@ .
```

```
val it = GREATER : order
```

The module `Set` defines an abstract type

```
type 'a set
```

of finite sets of elements of type `'a`. It is bound in the top-level environment. E.g., `int set` is the type of sets of integers. `Set` also defines a variety of functions for processing sets. But we will only make direct use of a few of them, including:

```
val toList : 'a set -> 'a list
val size   : 'a set -> int
val empty  : 'a set
val sing   : 'a -> 'a set
```

These functions are "polymorphic": they are applicable to values of type `int set`, `sym set`, etc. The function `sing` makes a value x into the singleton set $\{x\}$.

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
```

Sets of symbols are expressed in Forlan as sequences of symbols, separated by commas. When a set is outputted, or converted to a list, its elements are listed in ascending order.

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
```

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, as well as 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 prefix   : str * str -> bool
val suffix   : str * str -> bool
val substr   : str * str -> bool
val power    : str * int -> str
```

`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.

The type `str` is bound in the top-level environment, and is equal to `sym list`, the type of lists of symbols. Every value of type `str` has the form `[`a_1`, ..., `a_n`]`, where $n \in \mathbb{N}$ and the a_i are symbols. The usual list processing functions, such as `@` (append) and `length`, are applicable to elements of type `str`, and the empty string can be written as either `[]` or `nil`.

Every string can be expressed in Forlan's input syntax as either a single `%` or a nonempty sequence of symbols. For convenience, though, string expressions may be built up from symbols and `%` using parentheses (for grouping) and concatenation. During input processing, the parentheses are removed and the concatenations are carried out, producing lists of symbols. E.g., `%(hell)%o` describes the same string as `hello`.

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.input "";
@ %(hell)%o
@ .
val y = [-,-,-,-,-] : str
- Str.output("", y);
hello
val it = () : unit
- Str.compare(x, y);
val it = GREATER : order
- Str.output("", x @ y);
hello<there>hello
val it = () : unit
- Str.prefix(y, x);
val it = true : bool
- Str.substr(y, x);
val it = true : bool
```

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 alphabet : str set -> sym set
```

Sets of strings are expressed in Forlan as sequences of strings, separated by commas. When a set is outputted, or converted to a list, its elements are listed in ascending order. 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>, ano%ther
@ .
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
- SymSet.output("", StrSet.alphabet zs);
a, e, h, l, n, o, r, t, <id>, <num>
val it = () : unit
```

The module `SymRel` defines a 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. `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 domain      : sym_rel -> sym set
val range       : sym_rel -> sym set
val reflexive     : sym_rel * sym set -> bool
val symmetric     : sym_rel -> bool
val transitive    : sym_rel -> bool
val function      : sym_rel -> bool
val applyFunction : sym_rel -> sym -> sym
```

Relations on symbols are expressed in Forlan as sequences of ordered pairs `(`*a*`,`*b*`)` of symbols, separated by commas. When a relation is outputted, or converted to a list, its pairs are listed in ascending order, first according to their left-sides, and then according to their right sides. `reflexive(`*rel*`, `*bs*`)` tests whether *rel* is reflexive on *bs*. The function `applyFunction` is *curried*, i.e., it is a function that returns a function. Given a relation *rel*, it 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*.

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

```
- val rel = SymRel.input "";
@ (1, 2), (2, 3), (3, 4), (4, 5)
@ .
val rel = - : sym_rel
- SymRel.output("", rel);
(1, 2), (2, 3), (3, 4), (4, 5)
val it = () : unit
- SymSet.output("", SymRel.domain rel);
1, 2, 3, 4
val it = () : unit
- SymSet.output("", SymRel.range rel);
2, 3, 4, 5
val it = () : unit
- SymRel.reflexive(rel, SymSet.fromString "1, 2");
val it = false : bool
- SymRel.symmetric rel;
val it = false : bool
- SymRel.transitive rel;
val it = false : bool
- SymRel.function rel;
val it = true : bool
- val f = SymRel.applyFunction rel;
val f = fn : sym -> sym
- Sym.output("", f(Sym.fromString "3"));
4
val it = () : unit
- Sym.output("", f(Sym.fromString "4"));
```

```
5
val it = () : unit
- Sym.output("", f(Sym.fromString "5"));
argument not in domain

uncaught exception Error
```

Chapter 3

Regular Languages

In this chapter, we study: regular expressions and languages; four kinds of finite automata; algorithms for processing regular expressions and finite automata; properties of regular languages; and applications of regular expressions and finite automata to searching in text files and lexical analysis.

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.

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,

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

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

$$(L_1 \mathbin{@} L_2) \mathbin{@} L_3 = L_1 \mathbin{@} (L_2 \mathbin{@} L_3).$$

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

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

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

$$\emptyset \mathbin{@} L = L \mathbin{@} \emptyset = \emptyset.$$

We often abbreviate $L_1 \mathbin{@} L_2$ 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 language L to a power $n \in \mathbb{N}$* by recursion on n:

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

We assign this operation higher precedence than concatenation, so that LL^n means $L(L^n)$ in the above definition. 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. $\square$

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

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

and

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

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. $\square$

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:

- $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$;
- 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,

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

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 $\{\mathsf{0}, \mathsf{1}\}$ (Section 2.1); or
- The closure of the language $\{\mathsf{0}, \mathsf{1}\}$.

Fortunately, these languages are equal (both are all strings of $\mathsf{0}$'s and $\mathsf{1}$'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);
- 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 $(A \cup (BC))^*$.

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
```

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

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

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

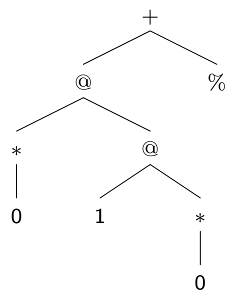
- (empty string) $\% \in \mathbf{Reg}$;
- (empty set) $\$ \in \mathbf{Reg}$;
- (symbol) for all $a \in \mathbf{Sym}$, $a \in \mathbf{Reg}$;
- (closure) for all $\alpha \in \mathbf{Reg}$, $*(\alpha) \in \mathbf{Reg}$;
- (concatenation) for all $\alpha, \beta \in \mathbf{Reg}$, $@(\alpha, \beta) \in \mathbf{Reg}$;
- (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,

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

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.

Because **Reg** is defined inductively, it gives rise to an induction principle. The *principle of induction on* **Reg** says that

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

follows from showing

- $P(\%)$;

- $P(\$)$;
- 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 $P(@(\alpha, \beta))$;
- for all $\alpha, \beta \in \mathbf{Reg}$, if $P(\alpha)$ and $P(\beta)$, then $P(+(\alpha, \beta))$.

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

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

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

- Highest: $(\cdot)^*$;
- Intermediate: @ (right associative);
- 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 $(\mathsf{0} + (\mathsf{1}\mathsf{2}))^* = (\mathsf{0} + \mathsf{1}\mathsf{2})^*$.

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

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

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

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

i.e.,

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

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:

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

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 $a \in$ **Sym**, **symToReg**(a) is the regular expression that looks like a. And, given symbols $a_1, \ldots, a_n$, for $n \in \mathbb{N}$, **strToReg**$(a_1 \ldots a_n)$ is the regular expression $\%$, if $n = 0$, and is the regular expression $a_1 \ldots a_n$, otherwise (remember that this is a tree, of size $n + (n - 1)$). It is easy to see that, for all $a \in$ **Sym**, $L($**symToReg**$(a)) = \{a\}$, and, for all $x \in$ **Str**, $L($**strToReg**$(x)) = \{x\}$.

We define the *regular expression α^n formed by raising a regular expres-*

sion α to a power $n \in \mathbb{N}$ by recursion on n:

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

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 made use of two base cases, so that α^1 is α, not $\alpha\%$. For example, $(\mathsf{0} + \mathsf{1})^3 = (\mathsf{0} + \mathsf{1})(\mathsf{0} + \mathsf{1})(\mathsf{0} + \mathsf{1})$.

Proposition 3.1.3
*For all $\alpha \in$ **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 lemma is that $L((\mathsf{0} + \mathsf{1})^3) = L(\mathsf{0} + \mathsf{1})^3 = \{\mathsf{0}, \mathsf{1}\}^3$.

We define the *alphabet of* a regular expression α (**alphabet**(α)) by recursion:

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

This is a good definition, since the union of two alphabets is an alphabet. For example, **alphabet**$(\mathsf{0}^*\mathsf{1}\mathsf{0}^* + \%) = \{\mathsf{0}, \mathsf{1}\}$.

Proposition 3.1.4
*For all $\alpha \in$ **Reg**, **alphabet**$(L(\alpha)) \subseteq$ **alphabet**(α).*

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

Proof. An easy induction on α, i.e., a proof using the principle of induction on **Reg**. □

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

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

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 by a finite sequence of ASCII characters, we have that **Reg** is countably infinite. Since $\{\mathsf{0}^0\}$, $\{\mathsf{0}^1\}$, $\{\mathsf{0}^2\}$, ..., are all regular languages, we have that **RegLan** is infinite.

To see that **RegLan** is countably infinite, imagine the following way of listing all of the regular languages. One works through the regular expressions, one after the other. Given a regular expression α, one asks whether the language $L(\alpha)$ has already appeared in our list. If not, we add it to the list, and then go on to the next regular expression. Otherwise, we simply go on to the next regular expression. It is easy to see that each regular language will appear exactly once in this infinite list. Thus **RegLan** is countably infinite.

Since **Lan** is uncountable, it follows that **RegLan** $\subsetneq$ **Lan**, i.e., there are non-regular languages. In Section 3.13, we will see a concrete example of a non-regular language.

Let's consider the problem of finding a regular expression that generates the set X of all strings of 0's and 1's with an even number of 0's. A string with this property would begin with some number of 1's (possibly none). After this, the string would have some number of parts (possibly none), each consisting of a 0, followed by some number of 1's, followed by a 0, followed by some number of 1's. The above considerations lead us to the regular expression $\alpha = \mathsf{1}^*(\mathsf{0}\mathsf{1}^*\mathsf{0}\mathsf{1}^*)^*$. To convince ourselves that this answer is correct, we must think about why $L(\alpha) = X$, i.e., why $L(\alpha) \subseteq X$ (everything generated by α is in X) and $X \subseteq L(\alpha)$ (everything in X is generated by α). In the next section, we'll consider proof methods for showing the correctness of regular expressions.

Now, we turn to the Forlan implementation of regular expressions. The Forlan module `Reg` defines an 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 compare  : reg * reg -> order
val alphabet : reg -> sym set
val emptyStr : reg
val emptySet : reg
val fromSym  : sym -> reg
val fromStr  : str -> reg
val closure  : reg -> reg
val concat   : reg * reg -> reg
val union    : reg * reg -> reg
val power    : reg * int -> reg
```

The Forlan syntax for regular expressions is our abbreviated linear notation. E.g., one must write `0*1` instead of `@(*(0),1)`. When regular expressions are outputted, as few parentheses as possible are used. The values `emptyStr` and `emptySet` represent % and $, respectively. The functions `fromSym` and `fromStr` implement the functions **symToReg** and **strToReg**, respectively, and are also bound in the top-level environment as `symToReg` and `strToReg`. The function `closure` takes in a regular expression α and returns $*(\alpha)$, and `concat` and `union` work similarly.

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
- 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.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
```

3.2 Equivalence and Simplification 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; look at an example of regular expression synthesis/proof of correctness; and describe two algorithms for the simplification of regular expressions. The first algorithm is weak, but efficient; the second is stronger, but inefficient. We also show how these algorithms can be used in Forlan.

We begin by saying what it means for two regular expressions to be equivalent. Regular expressions α and β are *equivalent* iff $L(\alpha) = L(\beta)$. In other words, α and β are equivalent iff α and β denote the same language. We define a relation $\approx$ on **Reg** by: $\alpha \approx \beta$ iff α and β are equivalent.

For example, $L((00)^* + \%) = L((00)^*)$, and thus $(00)^* + \% \approx (00)^*$.

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 $A_1, A_2, B_1, B_2 \in \textbf{Lan}$, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1A_2 \subseteq B_1B_2$.

(5) For all $A, B \in \textbf{Lan}$ and $n \in \mathbb{N}$, if $A \subseteq B$, then $A^n \subseteq B^n$.

(6) For all $A, B \in \textbf{Lan}$, if $A \subseteq B$, then $A^* \subseteq B^*$.

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 $A_1, A_2, B_1, B_2 \in \textbf{Lan}$, $A_1 \subseteq B_1$ 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 $\textbf{Reg}$, symmetric and transitive.

(2) For all $\alpha, \beta \in \textbf{Reg}$, if $\alpha \approx \beta$, then $\alpha^* \approx \beta^*$.

(3) For all $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \textbf{Reg}$, 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 $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \textbf{Reg}$, 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 $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \textbf{Reg}$, 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. $\square$

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 $\alpha, \beta, \gamma \in$ **Reg**, $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$.*

*(3) For all $\alpha \in$ **Reg**, $\$ + \alpha \approx \alpha$.*

*(4) For all $\alpha \in$ **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 $L_1 \cup L_2 = L_2$.

$\square$

Next, we consider equivalences for concatenation.

Proposition 3.2.5
*(1) For all $\alpha, \beta, \gamma \in$ **Reg**, $(\alpha\beta)\gamma \approx \alpha(\beta\gamma)$.*

*(2) For all $\alpha \in$ **Reg**, $\%\alpha \approx \alpha \approx \alpha\%$.*

*(3) For all $\alpha \in$ **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.

$\square$

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 $w = xy \in L_1(L_2 \cup L_3)$.

$\square$

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 $\alpha(\beta+\gamma)\approx\alpha\beta+\alpha\gamma$. $\square$

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
(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 $L\in\mathbf{Lan}$, $(L^*)^* = L^*$.

Proof. The five parts can be proven in order using Proposition 3.2.1. All parts but (2) and (5) can be proved without using induction.

As an example, we show the proof of Part (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 $L^* = (L^*)^1\subseteq(L^*)^*$. $\square$

By Proposition 3.2.8(4), we have that, for all $L \in \mathbf{Lan}$, $LL^* \subseteq L^*$ and $L^*L \subseteq L^*$. ($LL^* = L^1L^* \subseteq L^*L^* = L^*$, and the other inclusion follows similarly).

Proposition 3.2.9
(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 $\alpha \in \mathbf{Reg}$, $(\alpha^*)^* \approx \alpha^*$.

Proof. Follows from Proposition 3.2.8. Consider, e.g., the proof of Part (5). By Proposition 3.2.8(5), we have that

$$L((\alpha^*)^*) = L(\alpha^*)^* = (L(\alpha)^*)^* = L(\alpha)^* = L(\alpha^*).$$

Thus $(\alpha^*)^* \approx \alpha^*$. $\square$

Before going on to regular expression simplification, let's consider an example regular expression synthesis/proof of correctness problem. Let

$$\begin{aligned} A &= \{\mathsf{001}, \mathsf{011}, \mathsf{101}, \mathsf{111}\}, \\ B &= \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid \text{every occurrence of 0 in } w \\ &\qquad \text{is immediately followed by an element of } A \,\}. \end{aligned}$$

The elements of A can be thought of as the odd numbers between 1 and 7, expressed in binary. E.g., $\% \in B$, since the empty string has no occurrences of 0, and 00111 is in B, since its first 0 is followed by 011 and its second 0 is followed by 111. But 0000111 is not in B, since its first 0 is followed by 000, which is not in A. And 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 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 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 0 in y that wasn't followed by an element of A, then this same occurrence of 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 denotes 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, 1 is basic, so there will be no more basic strings that begin with 1. But what about the basic strings beginning with 0? No sequence of 0's is basic, and any string that begins with four or more 0's will not be basic. It is easy to see that 000111 is basic. In fact, it is the only basic string of the form $\mathsf{0001}u$. (The second 0 forces u to begin with 1, and the third forces u to begin with 11. And, if $|u| > 2$, then the overall string would have a nonempty, proper prefix in B, and so wouldn't be basic.) Similarly, 00111 is the only basic string beginning with 001. But what about the basic strings beginning with 01? It's not hard to see that there are infinitely many such strings: 0111, 010111, 01010111, 0101010111, etc. Fortunately, there is a simple pattern here: we have all strings of the form $\mathsf{0(10)}^n\mathsf{111}$ for $n \in \mathbb{N}$.

By the above considerations, it seems that we should let our regular expression be

$$(\mathsf{1} + \mathsf{0(10)}^*\mathsf{111} + \mathsf{00111} + \mathsf{000111})^*.$$

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

$$X = \{\mathsf{0}\} \cup \{\mathsf{00}\} \cup \{\mathsf{10}\}^*, \qquad Y = \{\mathsf{1}\} \cup \{\mathsf{0}\}X\{\mathsf{111}\}.$$

Then, we have that

$$\begin{aligned} X &= L(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*), \\ Y &= L(\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111}), \\ Y^* &= L((\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111})^*). \end{aligned}$$

Thus, it will suffice to show that $Y^* = B$. We will show that $Y^* \subseteq B \subseteq Y^*$.

Let

$$C = \{\, w \in B \mid w \text{ begins with } \mathsf{01} \,\}.$$

Lemma 3.2.10
For all $n \in \mathbb{N}$, $\{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq C$.

Proof. We proceed by mathematical induction.

(Basis Step) 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, every 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 3.2.11
$Y \subseteq B$.

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 3.2.10, we have that $w = \mathsf{0}x\mathsf{111} \in \{\mathsf{0}\}\{\mathsf{10}\}^n\{\mathsf{111}\} \subseteq C \subseteq B$.

□

Lemma 3.2.12
For all $n \in \mathbb{N}$, $Y^n \subseteq B$.

Proof. 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

$$\begin{aligned} Y^{n+1} &= YY^n \\ &\subseteq BB && \text{(Lemma 3.2.11 and the inductive hypothesis)} \\ &\subseteq B. \end{aligned}$$

□

Lemma 3.2.13
$Y^* \subseteq B$.

Proof. Suppose $w \in Y^*$. We must show that $w \in B$. We have that $w \in Y^n$ for some $n \in \mathbb{N}$. By Lemma 3.2.12, we have that $w \in Y^n \subseteq B$. □

Lemma 3.2.14
$B \subseteq Y^*$.

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 $|x| < |w|$, 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.

(1) Suppose $w = \%$. Then $w = \% \in \{\%\} = Y^0 \subseteq Y^*$.

(2) 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 $|x| < |w|$, the inductive hypothesis tells us that $x \in Y^*$. Thus $w = \mathsf{1}x \in YY^* \subseteq Y^*$.

(3) 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 $x = \mathsf{1}y$ for some $y \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{0}x = \mathsf{01}y$. Since $w \in B$, we have that $y \neq \%$. Thus, there are two cases to consider.
  - 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 $|u| < |w|$, the inductive hypothesis tells us that $u \in Y^*$. Hence $w = (\mathsf{0111})u \in YY^* \subseteq Y^*$.
  - Suppose $y = \mathsf{0}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{010}z$. Let u be the longest prefix of z that is in $\{\mathsf{10}\}^*$. (Since $\%$ is a prefix of z and is in $\{\mathsf{10}\}^*$, it follows that u is well-defined.) Let $v \in \{\mathsf{0},\mathsf{1}\}^*$ be such that $z = uv$. Thus $w = \mathsf{010}z = \mathsf{010}uv$.

    Suppose, toward a contradiction, that v begins with $\mathsf{10}$. Then $u\mathsf{10}$ is a prefix of $z = uv$ that is longer than $|u|$. Furthermore $u\mathsf{10} \in \{\mathsf{10}\}^*\{\mathsf{10}\} \subseteq \{\mathsf{10}\}^*$, contradicting the definition of u. Thus we have that v does not begin with $\mathsf{10}$.

    Next, we show that $\mathsf{010}u$ ends with $\mathsf{010}$. Since $u \in \{\mathsf{10}\}^*$, we have that $u \in \{\mathsf{10}\}^n$ for some $n \in \mathbb{N}$. There are three cases to consider.

    * Suppose $n = 0$. Since $u \in \{\mathsf{10}\}^0 = \{\%\}$, we have that $u = \%$. Thus $\mathsf{010}u = \mathsf{010}$ ends with $\mathsf{010}$.
    * Suppose $n = 1$. Since $u \in \{\mathsf{10}\}^1 = \{\mathsf{10}\}$, we have that $u = \mathsf{10}$. Hence $\mathsf{010}u = \mathsf{01010}$ ends with $\mathsf{010}$.
    * Suppose $n \geq 2$. Then $n - 2 \geq 0$, so that $u \in \{\mathsf{10}\}^{(n-2)+2} = \{\mathsf{10}\}^{n-2}\{\mathsf{10}\}^2$. Hence u ends with $\mathsf{1010}$, showing that $\mathsf{010}u$ ends with $\mathsf{010}$.

    Summarizing, we have that $w = \mathsf{010}uv$, $u \in \{\mathsf{10}\}^*$, $\mathsf{010}u$ ends with $\mathsf{010}$, and v does not begin with $\mathsf{10}$. Since the second-to-last $\mathsf{0}$ in $\mathsf{010}u$ is followed in w by an element of A, and $\mathsf{101}$ is the only element of A that begins with $\mathsf{10}$, we have that $v = \mathsf{1}s$ for some $s \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{010}u\mathsf{1}s$, and $\mathsf{010}u\mathsf{1}$ ends with $\mathsf{0101}$. Since the second-to-last symbol of $\mathsf{010}u\mathsf{1}$ is a $\mathsf{0}$, 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 11, we have that $t = 1r$ for some $r \in \{0,1\}^*$. Thus $w = 010u111r$. Since $(10)u \in \{10\}\{10\}^* \subseteq \{10\}^* \subseteq X$, we have that $010u111 = (0)((10)u)111 \in \{0\}X\{111\} \subseteq Y$. Since r is a suffix of w, it follows that $r \in B$. Thus, the inductive hypothesis tells us that $r \in Y^*$. Hence $w = (010u111)r \in YY^* \subseteq Y^*$.

- Suppose $x = 0y$ for some $y \in \{0,1\}^*$. Thus $w = 0x = 00y$. Since $00y = w \in B$, we have that $y \neq \%$. Thus, there are two cases to consider.
  - 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 $y = 0z$ for some $z \in \{0,1\}^*$. Thus $w = 00y = 000z$. Since the first 0 in $000z = w$ is followed by an element of A, and the only element of A that begins with 00 is 001, we have that $z = 1u$ for some $u \in \{0,1\}^*$. Thus $w = 0001u$. Since the second 0 in $0001u = w$ is followed by an element of A, and 011 is the only element of A that begins with 01, we have that $u = 1v$ for some $v \in \{0,1\}^*$. Thus $w = 00011v$. Since the third 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 $w = (000111)t \in YY^* \subseteq Y^*$.

□

By Lemmas 3.2.13 and 3.2.14, we have that $Y^* \subseteq B \subseteq Y^*$, i.e., $Y^* = B$. This completes our regular expression synthesis/proof of correctness example.

Next, we consider our first simplification algorithm—a weak, but efficient one. We define a function **weakSimplify** $\in$ **Reg** $\to$ **Reg** by recursion. For all $\alpha \in \textbf{Reg}$, $\textbf{weakSimplify}(\alpha)$ is defined as follows.

- If $\alpha = \%$, then $\textbf{weakSimplify}(\alpha) = \%$.
- If $\alpha = \$$, then $\textbf{weakSimplify}(\alpha) = \$$.
- If $\alpha \in \textbf{Sym}$, then $\textbf{weakSimplify}(\alpha) = \alpha$.
- Suppose $\alpha = \beta^*$, for some $\beta \in \textbf{Reg}$. Let $\beta' = \textbf{weakSimplify}(\beta)$. There are four cases to consider.
  - If $\beta' = \%$, then $\textbf{weakSimplify}(\alpha) = \%$.
  - If $\beta' = \$$, then $\textbf{weakSimplify}(\alpha) = \%$.
  - If $\beta' = \gamma^*$, for some $\gamma \in \textbf{Reg}$, then $\textbf{weakSimplify}(\alpha) = \beta'$.
  - Otherwise, $\textbf{weakSimplify}(\alpha) = \beta'^*$.
- Suppose $\alpha = \beta\gamma$, for some $\beta, \gamma \in \textbf{Reg}$. Let $\beta' = \textbf{weakSimplify}(\beta)$ and $\gamma' = \textbf{weakSimplify}(\gamma)$. There are four cases to consider.
  - If $\beta' = \%$, then $\textbf{weakSimplify}(\alpha) = \gamma'$.
  - Otherwise, if $\gamma' = \%$, then $\textbf{weakSimplify}(\alpha) = \beta'$.
  - Otherwise, if $\beta' = \$$ or $\gamma' = \$$, then $\textbf{weakSimplify}(\alpha) = \$$.
  - Otherwise, let $\beta'_1, \ldots, \beta'_n$, for $n \geq 1$, be such that $\beta' = \beta'_1 \cdots \beta'_n$ and β'_n is not a concatenation, and let $\gamma'_1, \ldots, \gamma'_m$, for $m \geq 1$, be such that $\gamma' = \gamma'_1 \cdots \gamma'_m$ and γ'_m is not a concatenation. Then $\textbf{weakSimplify}(\alpha)$ is the result of repeatedly walking down $\beta'_1 \cdots \beta'_n \gamma'_1 \cdots \gamma'_m$ and replacing adjacent regular expressions of the form $\alpha'^*\alpha'$ by $\alpha'\alpha'^*$.

    (For example, if $\beta' = \mathtt{011}^*$ and $\gamma' = \mathtt{10}$, then $\textbf{weakSimplify}(\alpha)$ is $\mathtt{0111}^*\mathtt{0}$, not $(\mathtt{011}^*)\mathtt{10} = (\mathtt{011}^*)(\mathtt{10})$.)
- Suppose $\alpha = \beta + \gamma$, for some $\beta, \gamma \in \textbf{Reg}$. Let $\beta' = \textbf{weakSimplify}(\beta)$ and $\gamma' = \textbf{weakSimplify}(\gamma)$. There are three cases to consider.
  - If $\beta' = \$$, then $\textbf{weakSimplify}(\alpha) = \gamma'$.
  - Otherwise, if $\gamma' = \$$, then $\textbf{weakSimplify}(\alpha) = \beta'$.
  - Otherwise, let $\beta'_1, \ldots, \beta'_n$, for $n \geq 1$, be such that $\beta' = \beta'_1 + \cdots + \beta'_n$ and β'_n is not a union, and let $\gamma'_1, \ldots, \gamma'_m$, for $m \geq 1$, be such that $\gamma' = \gamma'_1 + \cdots + \gamma'_m$ and γ'_m is not a union. Then $\textbf{weakSimplify}(\alpha)$ is the result of putting the summands in

$$\beta'_1 + \cdots + \beta'_n + \gamma'_1 + \cdots + \gamma'_m$$

in order without duplicates.

(For example, if $\beta' = 1 + 2 + 3$ and $\gamma' = 0 + 1$, then **weakSimplify**$(\alpha) = 0 + 1 + 2 + 3$.)

On the one hand, **weakSimplify** is just a mathematical function. But, because we have defined it recursively, we can use its definition to compute the result of calling it on a regular expression. Thus, we may regard the definition of **weakSimplify** as an algorithm.

Proposition 3.2.15
*For all $\alpha \in$ **Reg**:*

(1) **weakSimplify**$(\alpha) \approx \alpha$*;*

(2) **alphabet**(**weakSimplify**(α)) $\subseteq$ **alphabet**(α)*;*

(3) The size of **weakSimplify**(α) *is less-than-or-equal-to the size of* α*;*

(4) The number of concatenations in **weakSimplify**(α) *is less-than-or-equal-to the number of concatenations of* α*.*

Proof. By induction on **Reg**. □

We say that a regular expression α is *weakly simplified* iff none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$;
- $(\beta_1 + \beta_2) + \beta_3$;
- $\beta_1 + \beta_2$, where $\beta_1 \geq \beta_2$, or $\beta_1 + (\beta_2 + \beta_3)$, where $\beta_1 \geq \beta_2$;
- $\%\beta$ or $\beta\%$;
- $\$\beta$ or $\beta\$$;
- $(\beta_1\beta_2)\beta_3$;
- $\beta^*\beta$ or $\beta^*(\beta\gamma)$;
- $\%^*$ or $\$^*$ or $(\beta^*)^*$.

Thus, if a regular expression α is weakly simplified, then each of its subtrees will also be weakly simplified.

Proposition 3.2.16
For all $\alpha \in \mathbf{Reg}$, $\mathbf{weakSimplify}(\alpha)$ is weakly simplified.

Proof. By induction on **Reg**. $\square$

It turns out the weakly simplified regular expressions have some pleasing properties:

Proposition 3.2.17
For all $\alpha \in \mathbf{Reg}$:

(1) If α is weakly simplified and $L(\alpha) = \{\%\}$, then $\alpha = \%$;

(2) If α is weakly simplified and $L(\alpha) = \emptyset$, then $\alpha = \$$;

(3) For all $a \in \mathbf{Sym}$, if α is weakly simplified and $L(\alpha) = \{a\}$, then $\alpha = a$.

E.g., Part (1) of the proposition says that, if α is weakly simplified and $L(\alpha)$ is the language whose only string is %, then α is %.

Proof. By simultaneous induction on **Reg**, i.e., using the principle of induction on **Reg**. We show part of the proof of the concatenation case. Suppose $\alpha, \beta \in \mathbf{Reg}$ and assume the inductive hypothesis, that Parts (1)–(3) hold for α and β. One must show that Parts (1)–(3) hold for $\alpha\beta$. We will show that Part (3) holds for $\alpha\beta$. Suppose $a \in \mathbf{Sym}$, and assume that $\alpha\beta$ is weakly simplified and $L(\alpha\beta) = \{a\}$. We must show that $\alpha\beta = a$.

Since $L(\alpha)L(\beta) = L(\alpha\beta) = \{a\}$, there are two cases to consider.

- Suppose $L(\alpha) = \{a\}$ and $L(\beta) = \{\%\}$. Since β is weakly simplified and $L(\beta) = \{\%\}$, Part (1) of the inductive hypothesis on β tells us that $\beta = \%$. But this means that $\alpha\beta = \alpha\%$ is not weakly simplified after all—contradiction. Thus we can conclude that $\alpha\beta = a$.

- Suppose $L(\alpha) = \{\%\}$ and $L(\beta) = \{a\}$. The proof of this case is similar to that of the other one.

$\square$

The next proposition says that \$ need only be used at the top-level of a regular expression.

Proposition 3.2.18
For all $\alpha \in \mathbf{Reg}$, if α is weakly simplified and α has one or more occurrences of \$, then $\alpha = \$$.

Proof. An easy induction on **Reg**. □

Using our simplification algorithm, we can define an algorithm for calculating the language generated by a regular expression, when this language is finite, and for announcing that this language is infinite, otherwise.

First, we weakly simplify our regular expression, α, and call the resulting regular expression β. If β contains no closures, then we compute its meaning in the usual way. But, if β contains one or more closures, then its language will be infinite (see below), and thus we can output a message saying that $L(\alpha)$ is infinite.

We can use induction on **Reg** to prove that, for all $\alpha \in \mathbf{Reg}$, if α is weakly simplified and contains one more closures, then $L(\alpha)$ is infinite. The interesting cases are when α is a closure or a concatenation.

If α^* is weakly simplified, then α is weakly simplified and is not % or \$. Thus, by Proposition 3.2.17, $L(\alpha)$ contains at least one non-empty string, and thus $L(\alpha^*) = L(\alpha)^*$ is infinite.

And, if $\alpha\beta$ is weakly simplified and contains one or more closures, then α and β are weakly simplified, and either α or β will have a closure. Let's consider the case when α has a closure, the other case being similar. Then $L(\alpha)$ will be infinite. Since $\alpha\beta$ is weakly simplified, it follows that β is not \$. Thus, by Proposition 3.2.17, $L(\beta)$ contains at least one string, and thus $L(\alpha\beta) = L(\alpha)L(\beta)$ is infinite.

In preparation for the definition of our stronger simplification algorithm, we must define some auxiliary functions (algorithms). First, we show how we can recursively test whether $\% \in L(\alpha)$ for a regular expression α. We define a function

$$\mathbf{hasEmp} \in \mathbf{Reg} \to \{\mathbf{true}, \mathbf{false}\}$$

by recursion:

$$\begin{aligned}
\mathbf{hasEmp}(\%) &= \mathbf{true};\\
\mathbf{hasEmp}(\$) &= \mathbf{false};\\
\mathbf{hasEmp}(a) &= \mathbf{false},\ \text{for all } a \in \mathbf{Sym};\\
\mathbf{hasEmp}(\alpha^*) &= \mathbf{true},\ \text{for all } \alpha \in \mathbf{Reg};\\
\mathbf{hasEmp}(\alpha\beta) &= \mathbf{hasEmp}(\alpha)\ \mathbf{and}\ \mathbf{hasEmp}(\beta),\ \text{for all } \alpha, \beta \in \mathbf{Reg};\\
\mathbf{hasEmp}(\alpha + \beta) &= \mathbf{hasEmp}(\alpha)\ \mathbf{or}\ \mathbf{hasEmp}(\beta),\ \text{for all } \alpha, \beta \in \mathbf{Reg}.
\end{aligned}$$

Proposition 3.2.19
For all $\alpha \in \mathbf{Reg}$, $\% \in L(\alpha)$ iff $\mathbf{hasEmp}(\alpha) = \mathbf{true}$.

Proof. By induction on α. $\Box$

Next, we show how we can recursively test whether $a \in L(\alpha)$ for a symbol a and a regular expression α. We define a function

$$\mathbf{hasSym} \in \mathbf{Sym} \times \mathbf{Reg} \to \{\mathbf{true}, \mathbf{false}\}$$

by recursion:

$$\begin{aligned}
\mathbf{hasSym}(a, \%) &= \mathbf{false}, \text{ for all } a \in \mathbf{Sym}; \\
\mathbf{hasSym}(a, \$) &= \mathbf{false}, \text{ for all } a \in \mathbf{Sym}; \\
\mathbf{hasSym}(a, b) &= a = b, \text{ for all } a, b \in \mathbf{Sym}; \\
\mathbf{hasSym}(a, \alpha^*) &= \mathbf{hasSym}(a, \alpha), \text{ for all } a \in \mathbf{Sym} \text{ and } \alpha \in \mathbf{Reg}; \\
\mathbf{hasSym}(a, \alpha\beta) &= (\mathbf{hasSym}(a, \alpha) \textbf{ and } \mathbf{hasEmp}(\beta)) \textbf{ or} \\
&\quad\;\; (\mathbf{hasEmp}(\alpha) \textbf{ and } \mathbf{hasSym}(a, \beta)), \\
&\quad\;\; \text{for all } a \in \mathbf{Sym} \text{ and } \alpha, \beta \in \mathbf{Reg}; \\
\mathbf{hasSym}(a, \alpha + \beta) &= \mathbf{hasSym}(a, \alpha) \textbf{ or } \mathbf{hasSym}(a, \beta), \\
&\quad\;\; \text{for all } a \in \mathbf{Sym} \text{ and } \alpha, \beta \in \mathbf{Reg}.
\end{aligned}$$

Proposition 3.2.20
For all $a \in \mathbf{Sym}$ *and* $\alpha \in \mathbf{Reg}$, $a \in L(\alpha)$ *iff* $\mathbf{hasSym}(a, \alpha) = \mathbf{true}$.

Proof. By induction on **Reg**, using Proposition 3.2.19. $\Box$

Next, we define a function

$$\mathbf{weakSubset} \in \mathbf{Reg} \times \mathbf{Reg} \to \{\mathbf{true}, \mathbf{false}\}$$

that meets the following specification: for all $\alpha, \beta \in \mathbf{Reg}$, if $\mathbf{weakSubset}(\alpha, \beta) = \mathbf{true}$, then $L(\alpha) \subseteq L(\beta)$. I.e., this function conservatively approximates a test for $L(\alpha) \subseteq L(\beta)$. The function that always returns **false** would meet this specification, but our function will do much better than this, and will be reasonably efficient. In Section 3.12, we will learn of a less efficient algorithm that will provide a complete test for $L(\alpha) \subseteq L(\beta)$.

Given $\alpha, \beta \in \mathbf{Reg}$, we define $\mathbf{weakSubset}(\alpha, \beta)$ as follows. First, we let $\alpha' = \mathbf{weakSimplify}(\alpha)$ and $\beta' = \mathbf{weakSimplify}(\beta)$. Then we return $\mathbf{weakSub}(\alpha', \beta')$, where

$$\mathbf{weakSub} \in \mathbf{Reg} \times \mathbf{Reg} \to \{\mathbf{true}, \mathbf{false}\}$$

is the function defined below.

Given $\alpha, \beta \in \mathbf{Reg}$, we define $\mathbf{weakSub}(\alpha, \beta)$ by recursion on the sum of the sizes of α and β. If $\alpha = \beta$, then we return **true**; otherwise, we consider the possible forms of α.

- ($\alpha = \%$) We return $\mathbf{hasEmp}(\beta)$.
- ($\alpha = \$$) We return **true**.
- ($\alpha = a$, for some $a \in \mathbf{Sym}$) We return $\mathbf{hasSym}(a, \beta)$.
- ($\alpha = {\alpha_1}^*$, for some $\alpha_1 \in \mathbf{Reg}$) Here we must look at the form of β.
  - ($\beta = \%$) We return **false**. (In practice, α will be weakly simplified, and so α won't denote $\{\%\}$.)
  - ($\beta = \$$) We return **false**.
  - ($\beta = a$, for some $a \in \mathbf{Sym}$) We return **false**.
  - (β is a closure) We return $\mathbf{weakSub}(\alpha_1, \beta)$.
  - ($\beta = \beta_1\beta_2$, for some $\beta_1, \beta_2 \in \mathbf{Reg}$) If $\mathbf{hasEmp}(\beta_1) = \mathbf{true}$ and $\mathbf{weakSub}(\alpha, \beta_2)$, then we return **true**. Otherwise, if $\mathbf{hasEmp}(\beta_2) = \mathbf{true}$ and $\mathbf{weakSub}(\alpha, \beta_1)$, then we return **true**. Otherwise, we return **false**, even though the answer sometimes should be **true**.
  - ($\beta = \beta_1 + \beta_2$, for some $\beta_1, \beta_2 \in \mathbf{Reg}$) We return

    $$\mathbf{weakSub}(\alpha, \beta_1) \mathbf{\ or\ } \mathbf{weakSub}(\alpha, \beta_2),$$

    even though this is **false** too often.
- ($\alpha = \alpha_1\alpha_2$, for some $\alpha_1, \alpha_2 \in \mathbf{Reg}$) Here we must look at the form of β.
  - ($\beta = \%$) We return **false**. (In practice, α will be weakly simplified, and so α won't denote $\{\%\}$.)
  - ($\beta = \$$) We return **false**. (In practice, α will be weakly simplified, and so α won't denote $\emptyset$.)
  - ($\beta = a$, for some $a \in \mathbf{Sym}$) We return **false**. (In practice, α will be weakly simplified, and so α won't denote $\{a\}$.)

  - ($\beta = {\beta_1}^*$, for some $\beta_1 \in \mathbf{Reg}$) We return

    $$\mathbf{weakSub}(\alpha, \beta_1) \textbf{ or}$$
    $$(\mathbf{weakSub}(\alpha_1, \beta) \textbf{ and } \mathbf{weakSub}(\alpha_2, \beta)),$$

    even though this returns **false** too often.
  - ($\beta = \beta_1\beta_2$, for some $\beta_1, \beta_2 \in \mathbf{Reg}$) If $\mathbf{weakSub}(\alpha_1, \beta_1) = \mathbf{true}$ and $\mathbf{weakSub}(\alpha_2, \beta_2) = \mathbf{true}$, then we return **true**. Otherwise, if $\mathbf{hasEmp}(\beta_1) = \mathbf{true}$ and $\mathbf{weakSub}(\alpha, \beta_2) = \mathbf{true}$, then we return **true**. Otherwise, if $\mathbf{hasEmp}(\beta_2) = \mathbf{true}$ and $\mathbf{weakSub}(\alpha, \beta_1) = \mathbf{true}$, then we return **true**. Otherwise, if β_1 is a closure, then we return

    $$\mathbf{weakSub}(\alpha_1, \beta_1) \textbf{ and } \mathbf{weakSub}(\alpha_2, \beta),$$

    even though this returns **false** too often. Otherwise, we return **false**, even though sometimes we would like the answer to be **true**.
  - ($\beta = \beta_1 + \beta_2$, for some $\beta_1, \beta_2 \in \mathbf{Reg}$) We return

    $$\mathbf{weakSub}(\alpha, \beta_1) \textbf{ or } \mathbf{weakSub}(\alpha, \beta_2),$$

    even though this is **false** too often.

- ($\alpha = \alpha_1 + \alpha_2$) We return

  $$\mathbf{weakSub}(\alpha_1, \beta) \textbf{ and } \mathbf{weakSub}(\alpha_2, \beta).$$

Proposition 3.2.21
For all $\alpha, \beta \in \mathbf{Reg}$, if $\mathbf{weakSubset}(\alpha, \beta) = \mathbf{true}$, then $L(\alpha) \subseteq L(\beta)$.

Proof. First, we use induction on the sum of the sizes of α and β to show that, for all $\alpha, \beta \in \mathbf{Reg}$, if $\mathbf{weakSub}(\alpha, \beta) = \mathbf{true}$, then $L(\alpha) \subseteq L(\beta)$. Then result then follows by Proposition 3.2.15. $\square$

On the positive side, we have that, e.g., $\mathbf{weakSubset}(\mathsf{0}^*\mathsf{011}^*\mathsf{1}, \mathsf{0}^*\mathsf{1}^*)$. On the other hand, $\mathbf{weakSubset}((\mathsf{01})^*, (\% + \mathsf{0})(\mathsf{10})^*(\% + \mathsf{1})) = \mathbf{false}$, even though $L((\mathsf{01})^*) \subseteq L((\% + \mathsf{0})(\mathsf{10})^*(\% + \mathsf{1}))$.

Now, we give the definition of our stronger simplification function (algorithm):

$$\mathbf{simplify} \in (\mathbf{Reg} \times \mathbf{Reg} \to \{\mathbf{true}, \mathbf{false}\}) \to \mathbf{Reg} \to \mathbf{Reg}.$$

This function takes in a function *sub* (like **weakSubset**) that conservatively approximates the test for one regular expression's language being a subset of another expression's language, and returns a function that uses *sub* in order to simplify regular expressions.

Our definition of **simplify** is based on the following twenty-one *simplification rules*, which may be applied to arbitrary subtrees of regular expressions. Each simplification rule either:

- strictly decreases the size of a regular expression; or
- preserves the size of a regular expression, but decreases the number of concatenations in the regular expression (see Rules 7 and 8).

In the rules, we abbreviate $\mathbf{hasEmp}(\alpha) = \mathbf{true}$ and $\mathit{sub}(\alpha,\beta) = \mathbf{true}$ to $\mathbf{hasEmp}(\alpha)$ and $\mathit{sub}(\alpha,\beta)$, respectively.

(1) $\alpha^*(\beta\alpha^*)^* \to (\alpha+\beta)^*$.

(2) $(\alpha^*\beta)^*\alpha^* \to (\alpha+\beta)^*$.

(3) If $\mathbf{hasEmp}(\alpha)$ and $\mathit{sub}(\alpha,\beta^*)$, then $\alpha\beta^* \to \beta^*$.

(4) If $\mathbf{hasEmp}(\beta)$ and $\mathit{sub}(\beta,\alpha^*)$, then $\alpha^*\beta \to \alpha^*$.

(5) If $\mathit{sub}(\alpha,\beta^*)$, then $(\alpha+\beta)^* \to \beta^*$.

(6) $(\alpha+\beta^*)^* \to (\alpha+\beta)^*$.

(7) If $\mathbf{hasEmp}(\alpha)$ and $\mathbf{hasEmp}(\beta)$, then $(\alpha\beta)^* \to (\alpha+\beta)^*$.

(8) If $\mathbf{hasEmp}(\alpha)$ and $\mathbf{hasEmp}(\beta)$, then $(\alpha\beta+\gamma)^* \to (\alpha+\beta+\gamma)^*$.

(9) If $\mathbf{hasEmp}(\alpha)$ and $\mathit{sub}(\alpha,\beta^*)$, then $(\alpha\beta)^* \to \beta^*$.

(10) If $\mathbf{hasEmp}(\beta)$ and $\mathit{sub}(\beta,\alpha^*)$, then $(\alpha\beta)^* \to \alpha^*$.

(11) If $\mathbf{hasEmp}(\alpha)$ and $\mathit{sub}(\alpha,(\beta+\gamma)^*)$, then $(\alpha\beta+\gamma)^* \to (\beta+\gamma)^*$.

(12) If $\mathbf{hasEmp}(\beta)$ and $\mathit{sub}(\beta,(\alpha+\gamma)^*)$, then $(\alpha\beta+\gamma)^* \to (\alpha+\gamma)^*$.

(13) If $\mathit{sub}(\alpha,\beta)$, then $\alpha+\beta \to \beta$.

(14) $\alpha\beta_1+\alpha\beta_2 \to \alpha(\beta_1+\beta_2)$.

(15) $\alpha_1\beta+\alpha_2\beta \to (\alpha_1+\alpha_2)\beta$.

(16) If $sub(\alpha\beta_1, \alpha\beta_2)$, then $\alpha(\beta_1 + \beta_2) \to \alpha\beta_2$.

(17) If $sub(\alpha_1\beta, \alpha_2\beta)$, then $(\alpha_1 + \alpha_2)\beta \to \alpha_2\beta$.

(18) If $sub(\alpha\alpha^*, \beta)$, then $\alpha^* + \beta \to \% + \beta$.

(19) If $\mathbf{hasEmp}(\beta)$ and $sub(\alpha\alpha\alpha^*, \beta)$, then $\alpha^* + \beta \to \alpha + \beta$.

(20) If $\mathbf{hasEmp}(\beta)$, then $\alpha\alpha^* + \beta \to \alpha^* + \beta$.

(21) If $n \geq 1$ and $sub(\alpha^n, \beta)$, then $\alpha^{n+1}\alpha^* + \beta \to \alpha^n\alpha^* + \beta$.

Consider, e.g., rule (3). Suppose $\mathbf{hasEmp}(\alpha) = \mathbf{true}$ and $sub(\alpha, \beta^*) = \mathbf{true}$, so that that $\% \in L(\alpha)$ and $L(\alpha) \subseteq L(\beta^*)$. We need that $\alpha\beta^* \approx \beta^*$ and that the size of β^* is strictly less than the size of $\alpha\beta^*$.

To obtain $\alpha\beta^* \approx \beta^*$, it will suffice to show that, for all $A, B \in \mathbf{Lan}$, if $\% \in A$ and $A \subseteq B^*$, then $AB^* = B^*$. Suppose $A, B \in \mathbf{Lan}$, $\% \in A$ and $A \subseteq B^*$. We show that $AB^* \subseteq B^* \subseteq AB^*$. Suppose $w \in AB^*$, so that $w = xy$, for some $x \in A$ and $y \in B^*$. Since $A \subseteq B^*$, it follows that $w = xy \in B^*B^* = B^*$. Suppose $w \in B^*$. Then $w = \%w \in AB^*$.

The size of $\alpha\beta^*$ is the size of α plus the size of β plus 1 (for the closure) plus 1 (for the concatenation). But the size of β^* is the size of β plus one, and thus the size of β^* is strictly less than the size of $\alpha\beta^*$.

We also make use of the following nine *closure rules*, which may be applied to any subtree of a regular expression, and which preserve the alphabet, size and number of concatenations of a regular expression:

(1) $(\alpha + \beta) + \gamma \to \alpha + (\beta + \gamma)$.

(2) $\alpha + (\beta + \gamma) \to (\alpha + \beta) + \gamma$.

(3) $\alpha(\beta\gamma) \to (\alpha\beta)\gamma$.

(4) $(\alpha\beta)\gamma \to \alpha(\beta\gamma)$.

(5) $\alpha + \beta \to \beta + \alpha$.

(6) $\alpha^*\alpha \to \alpha\alpha^*$.

(7) $\alpha\alpha^* \to \alpha^*\alpha$.

(8) $\alpha(\beta\alpha)^* \to (\alpha\beta)^*\alpha$.

(9) $(\alpha\beta)^*\alpha \to \alpha(\beta\alpha)^*$.

Our simplification algorithm works as follows, given a regular expression α. We first replace α by $\mathbf{weakSimplify}(\alpha)$. Then we enter our main loop:

- We start working our way through all of the finitely many regular expressions β that α can be transformed to using our closure rules. (At each point in the generation of this sequence of regular expressions, we have a list of regular expressions that have already been chosen (initially, empty), plus a sorted (without duplicates) list of regular expressions that we have yet to process (initially, β). When the second of these lists becomes empty, we are done. Otherwise, we process the first element, γ, of this list. If γ is in our list of already chosen regular expressions, then we do nothing. Otherwise, γ is our next regular expression. We then add γ to the list of already chosen regular expressions, compute a sorted list consisting of all the ways of changing γ by single applications of closure rules, and add this sorted list at the end of the list of regular expressions that we have yet to process.)

- If one of our simplification rules applies to such a β, then we apply the rule to β, yielding the result γ, set α to $\mathbf{weakSimplify}(\gamma)$, and branch back to the beginning of our loop. (We start by working through the simplification rules, in order, looking for one that applied to the top-level of β. If we don't find one, the we carry out this process, recursively, on the children of β, working from left to right.)

- Otherwise, we select the next value of β, and continue this process.

- If we exhaust all of the β's, then we return α as our answer.

Each iteration of our loop either decreases the size of our regular expression, or maintains the size, but decreases the number of its concatenations. This explains why our algorithm always terminates. On the other hand, in some cases there will be so many ways of reorganizing a regular expression using the closure rules that one won't be able to wait for the algorithm to terminate. For example, if $\alpha = \alpha_1 + \cdots + \alpha_n$, then there are at least $n!$ ways of reorganizing α using Closure Rules (1), (2) and (5) alone. On the other hand, the algorithm will terminate sufficiently quickly on some large regular expressions, especially since the closure rules are applied lazily.

We say that a regular expression α is *sub-simplified* iff

- α is weakly simplified, and

- α can't be transformed by our closure rules into a regular expression to which one of our simplification rules applies.

Thus, if α is *sub*-simplified, then every subtree of α is also *sub*-simplified.

Theorem 3.2.22
For all $\alpha \in \mathbf{Reg}$:

(1) $\mathbf{simplify}(sub)(\alpha) \approx \alpha$*;*

(2) $\mathbf{alphabet}(\mathbf{simplify}(sub)(\alpha)) \subseteq \mathbf{alphabet}(\alpha)$*;*

(3) The size of $\mathbf{simplify}(sub)(\alpha)$ *is less-than-or-equal-to the size of* α*;*

(4) $\mathbf{simplify}(sub)(\alpha)$ *is sub-simplified.*

Now, we turn out attention to the implementation of regular expression simplification in Forlan. The Forlan module `Reg` also defines the functions:

```
val weakSimplify  : reg -> reg
val weakSubset    : reg * reg -> bool
val simplify      : (reg * reg -> bool) -> reg -> reg
val traceSimplify : (reg * reg -> bool) -> reg -> reg
val fromStrSet    : str set -> reg
val toStrSet      : reg -> str set
```

The function `traceSimplify` is like `simplify`, except that it outputs a trace of the simplification process. The function `fromStrSet` converts a finite language into a regular expression denoting that language in the most obvious way, and the function `toStrSet` returns the language generated by a regular expression, when that language is finite, and informs the user that the language is infinite, otherwise.

Here are some example uses of these functions:

```
- val reg = Reg.input "";
@ (% + $0)(% + 00*0 + 0**)*
@ .
val reg = - : reg
- Reg.output("", Reg.weakSimplify reg);
(% + 0* + 000*)*
val it = () : unit
- Reg.output("", Reg.simplify Reg.weakSubset reg);
0*
val it = () : unit
- Reg.toStrSet reg;
language is infinite

uncaught exception Error
```

```
- val reg'' = Reg.input "";
@ (1+%)(2+$)(3+%*)(4+$*)
@ .
val reg'' = - : reg
- StrSet.output("", Reg.toStrSet reg'');
2, 12, 23, 24, 123, 124, 234, 1234
val it = () : unit
- Reg.output("", Reg.weakSimplify reg'');
(% + 1)2(% + 3)(% + 4)
val it = () : unit
- Reg.output("", Reg.fromStrSet(StrSet.input ""));
@ hello, there, again
@ .
again + hello + there
val it = () : unit
- val reg''' = Reg.input "";
@ 1 + (% + 0 + 2)(% + 0 + 2)*1 +
@ (1 + (% + 0 + 2)(% + 0 + 2)*1)
@ (% + 0 + 2 + 1(% + 0 + 2)*1)
@ (% + 0 + 2 + 1(% + 0 + 2)*1)*
@ .
val reg''' = - : reg
- Reg.size reg''';
val it = 68 : int
- Reg.size(Reg.weakSimplify reg''');
val it = 68 : int
- Reg.output("", Reg.simplify Reg.weakSubset reg''');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
```

The last of these regular expressions denotes the set of all strings of 0's, 1's and 2's with an odd number of 1's. Here is an example use of the `traceSimplify` function:

```
- Reg.traceSimplify Reg.weakSubset (Reg.input "");
@ (0+1)*0*0
@ .
(0 + 1)*0*0
weakly simplifies to
(0 + 1)*00*
is transformed by closure rules to
((0 + 1)*0*)0
is transformed by simplification rule 4 to
(0 + 1)*0
weakly simplifies to
```

```
(0 + 1)*0
is simplified
val it = - : reg
```

For even some surprisingly small regular expressions, like

$$0001(001)^*01(001)^*,$$

working through all the ways that the regular expressions may be transformed using our closure rules may take too long. Consequently there is a Forlan parameter that controls the number of closure rule steps these functions are willing to carry out, at each iteration of the main simplification loop. The default maximum number of closure rule steps is 3000. This limit may be changed or eliminated using the function

```
val setRegClosureSteps : int option -> unit
```

of the `Params` module. Calling this function with argument `NONE` causes the limit to be eliminated. Calling it with argument `SOME` n causes the limit to be set to n. When the application of closure rules is aborted, the answer will still have all of the properties of Theorem 3.2.22, except for being completely *sub*-simplified.

Here is how the above regular expression is handled by `simplify` and `traceSimplify`:

```
- Reg.simplify Reg.weakSubset (Reg.input "");
@ 0001(001)*01(001)*
@ .
val it = - : reg
- Reg.traceSimplify Reg.weakSubset (Reg.input "");
@ 0001(001)*01(001)*
@ .
0001(001)*01(001)*
weakly simplifies to
0001(001)*01(001)*
is simplified (rule closure aborted)
val it = - : reg
```

If one eliminates the limit on the number of closure steps that may be applied at each iteration of the main simplification loop, then, after a fairly long time (e.g., about half an hour of CPU time on my laptop), one learns that

$$0001(001)^*01(001)^*$$

is **weakSubset**-simplified (assuming, of course, that Forlan is correct).

3.3 Finite Automata and Labeled Paths

In this section, we: say what finite automata (FA) are, and give an introduction to how they can be processed using Forlan; say what labeled paths are, and show how they can be processed using Forlan; and use the notion of labeled path to say what finite automata mean.

First, we say what finite automata are. A *finite automaton* (FA) M consists of:

- a finite set Q_M of symbols (we call the elements of Q_M the *states* of M);
- an element s_M of Q_M (we call s_M the *start state* of M);
- a subset A_M of Q_M (we call the elements of A_M the *accepting states* of M);
- a finite subset T_M of $\{\, (q, x, r) \mid q, r \in Q_M \text{ and } x \in \mathbf{Str}\,\}$ (we call the elements of T_M the *transitions* of M).

In a context where we are only referring to a single FA, M, we sometimes abbreviate Q_M, s_M, A_M and T_M to Q, s, A and T, respectively. Whenever possible, we will use the mathematical variables p, q and r to name states. We write **FA** for the set of all finite automata, which is a countably infinite set. Two FAs are *equal* iff they have the same states, start states, accepting states, and transitions.

As an example, we can define an FA M as follows:

- $Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$;
- $s_M = \mathsf{A}$;
- $A_M = \{\mathsf{A}, \mathsf{C}\}$;
- $T_M = \{(\mathsf{A}, \mathsf{1}, \mathsf{A}), (\mathsf{B}, \mathsf{11}, \mathsf{B}), (\mathsf{C}, \mathsf{111}, \mathsf{C}), (\mathsf{A}, \mathsf{0}, \mathsf{B}), (\mathsf{A}, \mathsf{2}, \mathsf{B}), (\mathsf{A}, \mathsf{0}, \mathsf{C}),$ $(\mathsf{A}, \mathsf{2}, \mathsf{C}), (\mathsf{B}, \mathsf{0}, \mathsf{C}), (\mathsf{B}, \mathsf{2}, \mathsf{C})\}$.

Shortly, we will use the notion of labeled path to formally explain what finite automata mean. Before we are able to do that, however, it is useful to have an informal understanding of the meaning of FAs. Finite automata are *nondeterministic* machines that take strings as inputs. When a machine is run on a given input, it begins in its start state.

If, after some number of steps, the machine is in state p, the machine's remaining input begins with x, and one of the machine's transitions is (p, x, q), then the machine *may* read x from its input and switch to state q. If (p, y, r) is also a transition, and the remaining input begins with y, then consuming y and switching to state r will also be possible, etc.

If *at least one* execution sequence consumes all of the machine's input and takes it to one of its accepting states, then we say that the input is *accepted* by the machine; otherwise, we say that the input is *rejected*. The meaning of a machine is the language consisting of all strings that it accepts.

The Forlan syntax for FAs can be explained using an example. Here is how our example FA M can be expressed in Forlan's syntax:

```
{states}
A, B, C
{start state}
A
{accepting states}
A, C
{transitions}
A, 1 -> A; B, 11 -> B; C, 111 -> C;
A, 0 -> B; A, 2 -> B;
A, 0 -> C; A, 2 -> C;
B, 0 -> C; B, 2 -> C
```

Since whitespace characters are ignored by Forlan's input routines, the preceding description of M could have been formatted in many other ways. States are separated by commas, and transitions are separated by semicolons. The order of states and transitions is irrelevant.

Transitions that only differ in their right-hand states can be merged into single transition families. E.g., we can merge

```
A, 0 -> B
```

and

```
A, 0 -> C
```

into the transition family

```
A, 0 -> B | C
```

The Forlan module `FA` defines an abstract type `fa` (in the top-level environment) of finite automata, as well as a large number of functions and constants for processing FAs, including:

```
val input  : string -> fa
val output : string * fa -> unit
```

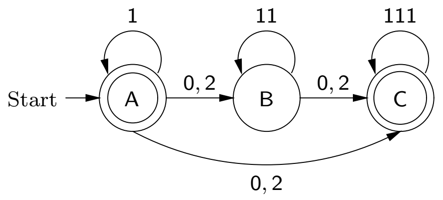
As usual, the `input` and `output` functions can be given either the names of the files they should read from or write to, or the null string `""`, which stands for the standard input or output. During printing, Forlan merges transitions into transition families whenever possible.

Suppose that our example FA is in the file `3.3-fa`. We can input this FA into Forlan, and then output it to the standard output, as follows:

```
- val fa = FA.input "3.3-fa";
val fa = - : fa
- FA.output("", fa);
{states}
A, B, C
{start state}
A
{accepting states}
A, C
{transitions}
A, 0 -> B | C; A, 1 -> A; A, 2 -> B | C; B, 0 -> C;
B, 2 -> C; B, 11 -> B; C, 111 -> C
val it = () : unit
```

We also make use of graphical notation for finite automata. Each of the states of a machine is circled, and its accepting states are double-circled. The machine's start state is pointed to by an arrow coming from "Start", and each transition (p, x, q) is drawn as an arrow from state p to state q that is labeled by the string x. Multiple labeled arrows from one state to another can be abbreviated to a single arrow, whose label consists of the comma-separated list of the labels of the original arrows.

For example, here is how our FA M can be described graphically:



The *alphabet of* a finite automaton M ($\mathbf{alphabet}(M)$) is $\{\, a \in \mathbf{Sym} \mid$ there are q, x, r such that $(q, x, r) \in T_M$ and $a \in \mathbf{alphabet}(x)\,\}$. I.e., $\mathbf{alphabet}(M)$ is all of the symbols appearing in the strings of M's transitions. For example, the alphabet of our example FA M is $\{0, 1, 2\}$.

The Forlan module `FA` contains the functions

```
val alphabet       : fa -> sym set
val numStates      : fa -> int
val numTransitions : fa -> int
val equal          : fa * fa -> bool
```

The function `alphabet` returns the alphabet of an FA, the functions `numStates` and `numTransitions` count the number of states and transitions, respectively, of an FA, and the function `equal` tests whether two FAs are identical, i.e., have the same states, start states, accepting states and transitions.

We will explain when strings are accepted by finite automata using the notion of a labeled path. A *labeled path* *lp* has the form

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

where $n \in \mathbb{N} - \{0\}$, the q_i's (which we think of as states) are symbols, and the x_i's are strings. We can think of a path of this form as describing a way of getting from state q_1 to state q_n, in some unspecified machine, by reading the strings $x_1, \ldots, x_{n-1}$ from the machine's input. We start out in state q_1, make use of the transition (q_1, x_1, q_2) to read x_1 from the input and switch to state q_2, etc. We write **LP** for the set of all labeled paths, which is a countably infinite set.

Let *lp* be the labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

We say that:

- the *start state* of *lp* (**startState**(*lp*)) is q_1;
- the *end state* of *lp* (**endState**(*lp*)) is q_n;
- the *length* of *lp* ($|lp|$) is $n - 1$;
- the *label* of *lp* (**label**(*lp*)) is $x_1 x_2 \cdots x_{n-1}$ (%, when $n = 1$).

For example

$$\mathsf{A}$$

is a labeled path whose start and end states are both A, whose length is 0, and whose label is %. And

$$\mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B} \overset{\mathsf{2}}{\Rightarrow} \mathsf{C}$$

is a labeled path whose start state is A, end state is C, length is 3, and label is $\mathsf{0(11)2} = \mathsf{0112}$. Note that every labeled path of length 0 has % as its label.

Paths

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n \quad \text{and} \quad p_1 \overset{y_1}{\Rightarrow} p_2 \overset{y_2}{\Rightarrow} \cdots p_{m-1} \overset{y_{m-1}}{\Rightarrow} p_m$$

are equal iff

- $n = m$;
- for all $1 \leq i \leq n$, $q_i = p_i$; and
- for all $1 \leq i \leq n-1$, $x_i = y_i$.

We sometimes (e.g., when using Forlan) write a path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n$$

as

$$q_1, x_1 \Rightarrow q_2, x_2 \Rightarrow \cdots q_{n-1}, x_{n-1} \Rightarrow q_n.$$

If lp_1 and lp_2 are the labeled paths

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n \quad \text{and} \quad p_1 \overset{y_1}{\Rightarrow} p_2 \overset{y_2}{\Rightarrow} \cdots p_{m-1} \overset{y_{m-1}}{\Rightarrow} p_m,$$

respectively, and $q_n = p_1$, then the *join* of lp_1 and lp_2 ($\mathbf{join}(lp_1, lp_2)$) is the labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n \overset{y_1}{\Rightarrow} p_2 \overset{y_2}{\Rightarrow} \cdots p_{m-1} \overset{y_{m-1}}{\Rightarrow} p_m.$$

For example, the join of

$$\mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B} \overset{\mathsf{2}}{\Rightarrow} \mathsf{C} \quad \text{and} \quad \mathsf{C} \overset{\mathsf{111}}{\Rightarrow} \mathsf{C}$$

is

$$\mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B} \overset{\mathsf{2}}{\Rightarrow} \mathsf{C} \overset{\mathsf{111}}{\Rightarrow} \mathsf{C}.$$

A labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

is *valid for* an FA M iff, for all $1 \leq i \leq n - 1$,

$$(q_i, x_i, q_{i+1}) \in T_M,$$

and $q_n \in Q_M$. When $n > 1$, the requirement that $q_n \in Q_M$ is redundant, since it will be implied by $q_{n-1}, x_{n-1} \Rightarrow q_n \in T_M$. But, when $n = 1$, there is no i such that $1 \leq i \leq n - 1$. Thus, if we didn't require that $q_n \in Q_M$, then all labeled paths of length 0 would be valid for all FAs.

For example, the labeled paths

$$\mathsf{A} \qquad \text{and} \qquad \mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B} \overset{\mathsf{2}}{\Rightarrow} \mathsf{C}$$

are valid for our example FA M. But the labeled path

$$\mathsf{A} \overset{\%}{\Rightarrow} \mathsf{A}$$

is not valid for M, since $(\mathsf{A}, \%, \mathsf{A}) \notin T_M$.

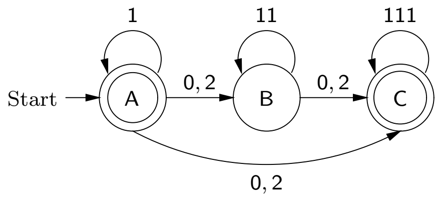
Now we are in a position to say what finite automata mean. A string w is *accepted by* a finite automaton M iff there is a labeled path *lp* such that

- the label of *lp* is w;
- *lp* is valid for M;
- the start state of *lp* is the start state of M; and
- the end state of *lp* is an accepting state of M.

Clearly, if w is accepted by M, then $\textbf{alphabet}(w) \subseteq \textbf{alphabet}(M)$. The *language accepted by* a finite automaton M ($L(M)$) is

$$\{\, w \in \textbf{Str} \mid w \text{ is accepted by } M \,\}.$$

Consider our example FA M:



We have that

$$\begin{aligned}L(M) = {} & \{\mathsf{1}\}^* \cup \\ & \{\mathsf{1}\}^*\{\mathsf{0},\mathsf{2}\}\{\mathsf{11}\}^*\{\mathsf{0},\mathsf{2}\}\{\mathsf{111}\}^* \cup \\ & \{\mathsf{1}\}^*\{\mathsf{0},\mathsf{2}\}\{\mathsf{111}\}^*.\end{aligned}$$

For example, %, 11, 110112111 and 2111111 are accepted by M.

Proposition 3.3.1
Suppose M is a finite automaton. Then $\mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}(M)$.

In other words, the proposition says that every symbol of every string that is accepted by M comes from the alphabet of M, i.e., appears in the label of one of M's transitions.

We say that finite automata M and N are *equivalent* iff $L(M) = L(N)$. In other words, M and N are equivalent iff M and N accept the same language. We define a relation $\approx$ on **FA** by: $M \approx N$ iff M and N are equivalent. It is easy to see that $\approx$ is reflexive on **FA**, symmetric and transitive.

The Forlan module `LP` defines an abstract type `lp` (in the top-level environment) of labeled paths, as well as various functions for processing labeled paths, including:

```
val input       : string -> lp
val output      : string * lp -> unit
val equal       : lp * lp -> bool
val startState  : lp -> sym
val endState    : lp -> sym
val label       : lp -> str
val length      : lp -> int
val join        : lp * lp -> lp
val sym         : sym -> lp
val cons        : sym * str * lp -> lp
val divideAfter : lp * int -> lp * lp
```

The specification of most of these functions is obvious. The function `join` issues an error message, if the end state of its first argument isn't the same as the start state of its second argument. The function `sym` turns a symbol a into the labeled path of length 0 whose start and end states are both a. The function `cons` adds a new transition to the left of a labeled path. And, `divideAfter(`lp`, `n`)` splits lp into a labeled path of length n and a labeled path of length $|lp| - n$, when $0 \leq n \leq |lp|$, and issues an error message, otherwise.

The module `FA` also defines the functions

```
val checkLP : fa -> lp -> unit
val validLP : fa -> lp -> bool
```

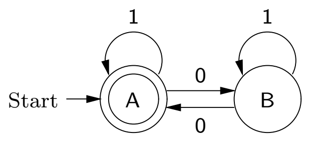
for checking whether a labeled path is valid in a finite automaton. These are curried functions—functions that return functions as their results. The function `checkLP` takes in an FA M and returns a function that checks whether a labeled path *lp* is valid for M. When *lp* is not valid for M, the function explains why it isn't; otherwise, it prints nothing. And, the function `validLP` takes in an FA M and returns a function that tests whether a labeled path *lp* is valid for M, silently returning `true`, if it is, and silently returning `false`, otherwise.

Here are some examples of labeled path and FA processing (`fa` is still our example FA):

```
- val lp = LP.input "";
@ A, 1 => A, 0 => B, 11 => B, 2 => C, 111 => C
@ .
val lp = - : lp
- Sym.output("", LP.startState lp);
A
val it = () : unit
- Sym.output("", LP.endState lp);
C
val it = () : unit
- LP.length lp;
val it = 5 : int
- Str.output("", LP.label lp);
10112111
val it = () : unit
- val checkLP = FA.checkLP fa;
val checkLP = fn : lp -> unit
- checkLP lp;
val it = () : unit
- val lp' = LP.fromString "A";
val lp' = - : lp
- LP.length lp';
val it = 0 : int
- Str.output("", LP.label lp');
%
val it = () : unit
- checkLP lp';
val it = () : unit
- checkLP(LP.input "");
@ A, % => A, 1 => A
@ .
invalid transition : "A, % -> A"

uncaught exception Error
- val lp'' = LP.join(lp, LP.input "");
@ C, 111 => C
@ .
val lp'' = - : lp
- LP.output("", lp'');
A, 1 => A, 0 => B, 11 => B, 2 => C, 111 => C, 111 => C
val it = () : unit
- checkLP lp'';
val it = () : unit
- val (lp1, lp2) = LP.divideAfter(lp'', 2);
val lp1 = - : lp
val lp2 = - : lp
- LP.output("", lp1);
A, 1 => A, 0 => B
val it = () : unit
- LP.output("", lp2);
B, 11 => B, 2 => C, 111 => C, 111 => C
val it = () : unit
```

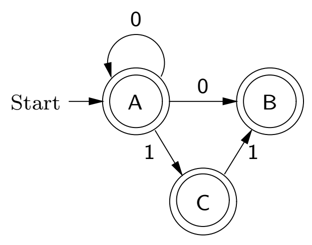
To conclude this section, let's consider the problem of finding a finite automaton that accepts the set of all strings of 0's and 1's with an even number of 0's. It seems reasonable that our machine have two states: a state A corresponding to the strings of 0's and 1's with an even number of zeros, and a state B corresponding to the strings of 0's and 1's with an odd number of zeros. Processing a 1 in either state should cause us to stay in that state, but processing a 0 in one of the states should cause us to switch to the other state. Because % has an even number of 0's, the start state, and only accepting state, will be A. The above considerations lead us to the FA:



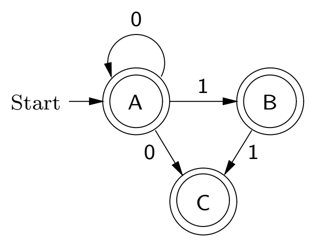
In Section 3.7, we'll study techniques for proving the correctness of FAs.

3.4 Isomorphism of Finite Automata

Let M and N be the finite automata



and



(M) (N)

How are M and N related? Although they are not equal, they do have the same "structure", in that M can be turned into N by replacing A, B and C by A, C and B, respectively. When FAs have the same structure, we will say they are "isomorphic".

In order to say more formally what it means for two FAs to be isomorphic, we define the notion of an isomorphism from one FA to another. An *isomorphism* h from an FA M to an FA N is a bijection from Q_M to Q_N such that

- $h(s_M) = s_N$;
- $\{\, h(q) \mid q \in A_M \,\} = A_N$;
- $\{\, (h(q), x, h(r)) \mid (q, x, r) \in T_M \,\} = T_N$.

We define a relation **iso** on **FA** by: $M\,\mathbf{iso}\,N$ iff there is an isomorphism from M to N. We say that M and N are *isomorphic* iff $M\,\mathbf{iso}\,N$.

Consider our example FAs M and N, and let h be the function

$$\{(\mathsf{A},\mathsf{A}), (\mathsf{B},\mathsf{C}), (\mathsf{C},\mathsf{B})\}.$$

Then it is easy to check that h is an isomorphism from M to N. Hence $M\,\mathbf{iso}\,N$.

Proposition 3.4.1
The relation **iso** *is reflexive on* **FA**, *symmetric and transitive.*

Proof. If M is an FA, then the identity function on Q_M is an isomorphism from M to M.

If M, N are FAs, and h is a isomorphism from M to N, then the inverse of h is an isomorphism from N to M.

If M_1, M_2, M_3 are FAs, f is an isomorphism from M_1 to M_2, and g is an isomorphism from M_2 to M_3, then the composition of g and f is an isomorphism from M_1 to M_3. $\square$

Next, we see that, if M and N are isomorphic, then every string accepted by M is also accepted by N.

Proposition 3.4.2
Suppose M and N are isomorphic FAs. Then $L(M) \subseteq L(N)$.

Proof. Let h be an isomorphism from M to N. Suppose $w \in L(M)$. Then, there is a labeled path

$$lp = q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

such that $w = x_1 x_2 \cdots x_{n-1}$, lp is valid for M, $q_1 = s_M$ and $q_n \in A_M$. Let

$$lp' = h(q_1) \overset{x_1}{\Rightarrow} h(q_2) \overset{x_2}{\Rightarrow} \cdots h(q_{n-1}) \overset{x_{n-1}}{\Rightarrow} h(q_n).$$

Then the label of lp' is w, lp' is valid for N, $h(q_1) = h(s_M) = s_N$ and $h(q_n) \in A_N$, showing that $w \in L(N)$. $\Box$

A consequence of the two preceding propositions is that isomorphic FAs are equivalent. Of course, the converse is not true, in general, since there are many FAs that accept the same language and yet don't have the same structure.

Proposition 3.4.3
Suppose M and N are isomorphic FAs. Then $M \approx N$.

Proof. Since M **iso** N, we have that N **iso** M, by Proposition 3.4.1. Thus, by Proposition 3.4.2, we have that $L(M) \subseteq L(N) \subseteq L(M)$. Hence $L(M) = L(N)$, i.e., $M \approx N$. $\Box$

Let $X = \{\, (M, f) \mid M \in$ **FA** and f is a bijection from Q_M to some set of symbols $\}$. The function **renameStates** $\in X \to$ **FA** takes in a pair (M, f) and returns the **FA** produced from M by renaming M's states using the bijection f.

Proposition 3.4.4
Suppose M is an FA and f is a bijection from Q_M to some set of symbols. Then **renameStates**(M, f) **iso** M.

The following function is a special case of **renameStates**. The function **renameStatesCanonically** $\in$ **FA** $\to$ **FA** renames the states of an FA M to:

- A, B, etc., when the automaton has no more than 26 states (the smallest state of M will be renamed to A, the next smallest one to B, etc.); or
- $\langle 1\rangle$, $\langle 2\rangle$, etc., otherwise.

Of course, the resulting automaton will always be isomorphic to the original one.

Next, we consider an algorithm that finds an isomorphism from an FA M to an FA N, if one exists, and that indicates that no such isomorphism exists, otherwise.

Our algorithm is based on the following lemma.

Lemma 3.4.5
Suppose that h is a bijection from Q_M to Q_N. Then

$$\{\,(h(q), x, h(r)) \mid (q, x, r) \in T_M\,\} = T_N$$

iff, for all $(q, r) \in h$ and $x \in \mathbf{Str}$, there is a subset of h that is a bijection from

$$\{\,p \in Q_M \mid (q, x, p) \in T_M\,\}$$

to

$$\{\,p \in Q_N \mid (r, x, p) \in T_N\,\}.$$

If any of the following conditions are true, then we report that there is no isomorphism from M to N:

- $|Q_M| \neq |Q_N|$;
- $|A_M| \neq |A_N|$;
- $|T_M| \neq |T_N|$;
- $s_M \in A_M$, but $s_N \not\in A_N$;
- $s_N \in A_N$, but $s_M \not\in A_M$.

Otherwise, we call our main, recursive function, **findIso**, which takes the following data:

- A bijection F from a subset of Q_M to a subset of Q_N.

- A list $C_1, \ldots, C_n$ of *constraints* of the form (X, Y), where $X \subseteq Q_M$, $Y \subseteq Q_N$ and $|X| = |Y|$.

We say that a bijection *satisfies* a constraint (X, Y) iff it has a subset that is a bijection from X to Y. **findIso** is supposed to return an isomorphism from M to N that is a superset of F and satisfies the constraints $C_1, \ldots, C_n$, if such an isomorphism exists; otherwise, it must return indicating failure.

We say that the *weight* of a constraint (X, Y) is $3^{|X|}$. Thus, we have the following facts:

- If (X, Y) is a constraint, then its weight is at least $3^0 = 1$.
- If $(\{p\} \cup X, \{q\} \cup Y)$ is a constraint, $p \not\in X$, $q \not\in Y$ and $|X| \geq 1$, then the weight of $(\{p\} \cup X, \{q\} \cup Y)$ is $3^{1+|X|} = 3 \cdot 3^{|X|}$, the weight of $(\{p\}, \{q\})$ is $3^1 = 3$, and the weight of (X, Y) is $3^{|X|}$. Because $|X| \geq 1$, it follows that the sum of the weights of $(\{p\}, \{q\})$ and (X, Y) $(3+3^{|X|})$ is strictly less-than the weight of $(\{p\} \cup X, \{q\} \cup Y)$.

Each argument to a recursive call of **findIso** will be strictly smaller than the argument to the original call in the *termination order* in which data $F, C_1, \ldots, C_n$ is less-than data $F', C'_1, \ldots, C'_m$ iff either:

- $|F| > |F'|$ (remember that $|F| \leq |Q_M| = |Q_N|$); or
- $|F| = |F'|$ but the sum of the weights of the constraints $C_1, \ldots, C_n$ is strictly less-than the sum of the weights of the constraints $C'_1, \ldots, C'_m$.

Thus every call of **findIso** will terminate.

When **findIso** is called with data $F, C_1, \ldots, C_n$, we will have that the following property, which we call (*), holds: for all bijections h from a subset of Q_M to a subset of Q_N, if $F \subseteq h$ and h satisfies all of the C_i's, then:

- h is a bijection from Q_M to Q_N; and
- $h(s_M) = s_N$;
- $\{\, h(q) \mid q \in A_M \,\} = A_N$;
- for all $(q, r) \in F$ and $x \in \textbf{Str}$, there is a subset of h that is a bijection from $\{\, p \in Q_M \mid (q, x, p) \in T_M \,\}$ to $\{\, p \in Q_N \mid (r, x, p) \in T_N \,\}$.

Thus, if **findIso** is called with a bijection F and an empty list of constraints, then it will follow, by Lemma 3.4.5, that F is an isomorphism from M to N, and **findIso** will simply return F.

Initially, we call **findIso** with the following data:

- The bijection $F = \emptyset$;
- The list of constraints consisting of $(\{s_M\}, \{s_N\})$, (A_1, A_2), (B_1, B_2), where A_1 and A_2 are the accepting but non-start states of M and N, respectively, and B_1 and B_2 are the non-accepting, non-start states of M and N, respectively.

If **findIso** is called with data $F, (\emptyset, \emptyset), C_2, \ldots, C_n$, then it calls itself recursively with data $F, C_2, \ldots, C_n$. (The size of the bijection has been preserved, but the sum of the weights of the constraints has gone down by one.)

If **findIso** is called with data $F, (\{q\}, \{r\}), C_2, \ldots, C_n$, then it proceeds as follows:

- If $(q, r) \in F$, then it calls itself recursively with data $F, C_2, \ldots, C_n$ and returns what the recursive call returns. (The size of the bijection has been preserved, but the sum of the weights of the constraints has gone down by three.)
- Otherwise, if $q \in \mathbf{domain}(F)$ or $r \in \mathbf{range}(F)$, then **findIso** returns indicating failure.
- Otherwise, it works its way through the strings appearing in the transitions of M and N, forming a list of new constraints, $C'_1, \ldots, C'_m$. Given such a string, x, it lets $A^x_1 = \{\, p \in Q_M \mid (q, x, p) \in T_M \,\}$ and $A^x_2 = \{\, p \in Q_N \mid (r, x, p) \in T_N \,\}$. If $|A^x_1| \neq |A^x_2|$, then it returns indicating failure. Otherwise, it adds the constraint (A^x_1, A^x_2) to our list of new constraints. When all such strings have been exhausted, it calls itself recursively with data $F \cup \{(q, r)\}, C'_1, \ldots, C'_m, C_2, \ldots, C_n$ and returns what this recursive call returns. (The size of the bijection has been increased by one.)

If **findIso** is called with data $F, (A_1, A_2), C_2, \ldots, C_n$, where $|A_1| > 1$, then it proceeds as follows. It picks the smallest symbol $q \in A_1$, and lets $B_1 = A_1 - \{q\}$. Then, it works its way through the elements of A_2. Given $r \in A_2$, it lets $B_2 = A_2 - \{r\}$. Then, it tries calling itself recursively with data $F, (\{q\}, \{r\}), (B_1, B_2), C_2, \ldots, C_n$. If this call returns an isomorphism h, then it returns it to its caller. (The size of the bijection has been preserved, but the sum of the sizes of the weights of the constraints has gone down by $2 \cdot 3^{|B_1|} - 3 \geq 3$.) Otherwise, if this recursive call indicates failure, then it tries the next element of A_2. If it exhausts the elements of A_2, then it returns indicating failure.

Lemma 3.4.6
If **findIso** *is called with data $F, C_1, \ldots, C_n$ satisfying property (*), then it returns an isomorphism from M to N that is a superset of F and satisfies the constraints C_i, if one exists, and returns indicating failure, otherwise.*

Proof. By *well-founded induction* on our termination ordering. I.e., when proving the result for $F, C_1, \ldots, C_n$, we may assume that the result holds for all data $F', C'_1, \ldots, C'_m$ that is strictly smaller in our termination ordering. □

Theorem 3.4.7
If **findIso** *is called with its initial data, then it returns an isomorphism from M to N, if one exists, and returns indicating failure, otherwise.*

Proof. Follows easily from Lemma 3.4.6. □

The Forlan module `FA` also defines the functions

```
val isomorphism             : fa * fa * sym_rel -> bool
val findIsomorphism         : fa * fa -> sym_rel
val isomorphic              : fa * fa -> bool
val renameStates            : fa * sym_rel -> fa
val renameStatesCanonically : fa -> fa
```

The function `isomorphism` checks whether a relation on symbols is an isomorphism from one FA to another. The function `findIsomorphism` tries to find an isomorphism from one FA to another; it issues an error message if it fails to find one. The function `isomorphic` checks whether two FAs are isomorphic. The function `renameStates` issues an error message if the supplied relation isn't a bijection from the set of states of the supplied FA to some set; otherwise, it returns the result of **renameStates**. And the function `renameStatesCanonically` acts like **renameStatesCanonically**.

Suppose that `fa1` and `fa2` have been bound to our example finite automata M and N, respectively. Then, here are some example uses of the above functions:

```
- val rel = FA.findIsomorphism(fa1, fa2);
val rel = - : sym_rel
- SymRel.output("", rel);
(A, A), (B, C), (C, B)
val it = () : unit
- FA.isomorphism(fa1, fa2, rel);
val it = true : bool
- FA.isomorphic(fa1, fa2);
```

```
val it = true : bool
- val rel' = FA.findIsomorphism(fa1, fa1);
val rel' = - : sym_rel
- SymRel.output("", rel');
(A, A), (B, B), (C, C)
val it = () : unit
- FA.isomorphism(fa1, fa1, rel');
val it = true : bool
- FA.isomorphism(fa1, fa2, rel');
val it = false : bool
- val rel'' = SymRel.input "";
@ (A, 2), (B, 1), (C, 0)
@ .
val rel'' = - : sym_rel
- val fa3 = FA.renameStates(fa1, rel'');
val fa3 = - : fa
- FA.output("", fa3);
{states}
0, 1, 2
{start state}
2
{accepting states}
0, 1, 2
{transitions}
0, 1 -> 1; 2, 0 -> 1 | 2; 2, 1 -> 0
val it = () : unit
- val fa4 = FA.renameStatesCanonically fa3;
val fa4 = - : fa
- FA.output("", fa4);
{states}
A, B, C
{start state}
C
{accepting states}
A, B, C
{transitions}
A, 1 -> B; C, 0 -> B | C; C, 1 -> A
val it = () : unit
- FA.equal(fa4, fa1);
val it = false : bool
- FA.isomorphic(fa4, fa1);
val it = true : bool
```

3.5 Algorithms for Checking Acceptance and Finding Accepting Paths

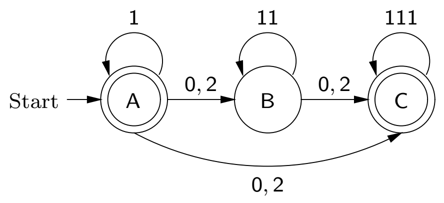
In this section we study algorithms for: checking whether a string is accepted by a finite automaton; and finding a labeled path that explains why a string is accepted by a finite automaton.

Suppose M is a finite automaton. We define a function $\Delta_M \in \mathcal{P}(Q_M) \times \mathbf{Str} \to \mathcal{P}(Q_M)$ by: $\Delta_M(P, w)$ is the set of all $r \in Q_M$ such that there is an $lp \in \mathbf{LP}$ such that

- w is the label of lp;
- lp is valid for M;
- the start state of lp is in P;
- r is the end state of lp.

In other words, $\Delta_M(P, w)$ consists of all of the states that can be reached from elements of P by labeled paths that are labeled by w and valid for M. When the FA M is clear from the context, we sometimes abbreviate Δ_M to Δ.

Suppose M is the finite automaton



Then, $\Delta_M(\{\mathsf{A}\}, 12111111) = \{\mathsf{B}, \mathsf{C}\}$, since

$$\mathsf{A} \overset{1}{\Rightarrow} \mathsf{A} \overset{2}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \quad \text{and} \quad \mathsf{A} \overset{1}{\Rightarrow} \mathsf{A} \overset{2}{\Rightarrow} \mathsf{C} \overset{111}{\Rightarrow} \mathsf{C} \overset{111}{\Rightarrow} \mathsf{C}$$

are all of the labeled paths that are labeled by 12111111, valid in M and whose start states are A. Furthermore, $\Delta_M(\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}, 11) = \{\mathsf{A}, \mathsf{B}\}$, since

$$\mathsf{A} \overset{1}{\Rightarrow} \mathsf{A} \overset{1}{\Rightarrow} \mathsf{A} \quad \text{and} \quad \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B}$$

are all of the labeled paths that are labeled by 11 and valid in M.

Suppose M is a finite automaton, $P \subseteq Q_M$ and $w \in Str$. We can calculate $\Delta_M(P, w)$ as follows.

Let S be the set of all suffixes of w. Given $y \in S$, we write $\mathbf{pre}(y)$ for the unique x such that $w = xy$.

First, we generate the least subset X of $Q_M \times S$ such that:

(1) for all $p \in P$, $(p, w) \in X$;

(2) for all $q, r \in Q_M$ and $x, y \in \mathbf{Str}$, if $(q, xy) \in X$ and $(q, x, r) \in T_M$, then $(r, y) \in X$.

We start by using rule (1), adding (p, w) to X, whenever $p \in P$. Then X (and any superset of X) will satisfy property (1). Then, rule (2) is used repeatedly to add more pairs to X. Since $Q_M \times S$ is a finite set, eventually X will satisfy property (2).

If M is our example finite automaton, then here are the elements of X, when $P = \{\mathsf{A}\}$ and $w = \mathsf{2111}$:

- $(\mathsf{A}, \mathsf{2111})$;
- $(\mathsf{B}, \mathsf{111})$, because of $(\mathsf{A}, \mathsf{2111})$ and the transition $(\mathsf{A}, \mathsf{2}, \mathsf{B})$;
- $(\mathsf{C}, \mathsf{111})$, because of $(\mathsf{A}, \mathsf{2111})$ and the transition $(\mathsf{A}, \mathsf{2}, \mathsf{C})$ (now, we're done with $(\mathsf{A}, \mathsf{2111})$);
- $(\mathsf{B}, \mathsf{1})$, because of $(\mathsf{B}, \mathsf{111})$ and the transition $(\mathsf{B}, \mathsf{11}, \mathsf{B})$ (now, we're done with $(\mathsf{B}, \mathsf{111})$);
- $(\mathsf{C}, \%)$, because of $(\mathsf{C}, \mathsf{111})$ and the transition $(\mathsf{C}, \mathsf{111}, \mathsf{C})$ (now, we're done with $(\mathsf{C}, \mathsf{111})$);
- nothing can be added using $(\mathsf{B}, \mathsf{1})$ and $(\mathsf{C}, \%)$, and so we've found all the elements of X.

The following lemma explains when pairs show up in X.

Lemma 3.5.1
For all $q \in Q_M$ and $y \in S$,

$$(q, y) \in X \quad \textit{iff} \quad q \in \Delta_M(P, \mathbf{pre}(y)).$$

Proof. The "only if" (left-to-right) direction is by induction on X: we show that, for all $(q, y) \in X$, $q \in \Delta_M(P, \mathbf{pre}(y))$.

- Suppose $p \in P$. Then $p \in \Delta_M(P, \%)$. But $\mathbf{pre}(w) = \%$, so that $p \in \Delta_M(P, \mathbf{pre}(w))$.

- Suppose $q, r \in Q_M$, $x, y \in \mathbf{Str}$, $(q, xy) \in X$ and $(q, x, r) \in T_M$. Assume the inductive hypothesis: $q \in \Delta_M(P, \mathbf{pre}(xy))$. Thus there is an $lp \in \mathbf{LP}$ such that $\mathbf{pre}(xy)$ is the label of lp, lp is valid for M, the start state of lp is in P, and q is the end state of lp. Let $lp' \in \mathbf{LP}$ be the result of adding the step $q, x \Rightarrow r$ at the end of lp. Thus $\mathbf{pre}(y)$ is the label of lp', lp' is valid for M, the start state of lp' is in P, and r is the end state of lp', showing that $r \in \Delta_M(P, \mathbf{pre}(y))$.

For the 'if" (right-to-left) direction, we have that there is a labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots \, q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

that is valid for M and where $\mathbf{pre}(y) = x_1x_2\cdots x_{n-1}$, $q_1 \in P$ and $q_n = q$. Since $q_1 \in P$ and $w = \mathbf{pre}(y)y = x_1x_2\cdots x_{n-1}y$, we have that $(q_1, x_1x_2\cdots x_{n-1}y) = (q_1, w) \in X$. But $(q_1, x_1, q_2) \in T_M$, and thus $(q_2, x_2\cdots x_{n-1}y) \in X$. Continuing on in this way, (we could do this by mathematical induction), we finally get that $(q, y) = (q_n, y) \in X$. $\square$

Lemma 3.5.2
For all $q \in Q_M$, $(q, \%) \in X$ iff $q \in \Delta_M(P, w)$.

Proof. Suppose $(q, \%) \in X$. Lemma 3.5.1 tells us that $q \in \Delta_M(P, \mathbf{pre}(\%))$. But $\mathbf{pre}(\%) = w$, and thus $q \in \Delta_M(P, w)$.

Suppose $q \in \Delta_M(P, w)$. Since $w = \mathbf{pre}(\%)$, we have that $q \in \Delta_M(P, \mathbf{pre}(\%))$. Lemma 3.5.1 tells us that $(q, \%) \in X$. $\square$

By Lemma 3.5.2, we have that

$$\Delta_M(P, w) = \{\, q \in Q_M \mid (q, \%) \in X \,\}.$$

Thus, we return the set of all states q that are paired with % in X.

Proposition 3.5.3
Suppose M is a finite automaton. Then

$$L(M) = \{\, w \in \mathbf{Str} \mid \Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset \,\}.$$

Proof. Suppose $w \in L(M)$. Then w is the label of a labeled path lp such that lp is valid in M, the start state of lp is s_M and the end state of lp is in A_M. Let q be the end state of lp. Thus $q \in \Delta_M(\{s_M\}, w)$ and $q \in A_M$, showing that $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$.

Suppose $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$, so that there is a q such that $q \in \Delta_M(\{s_M\}, w)$ and $q \in A_M$. Thus w is the label of a labeled path lp such that lp is valid in M, the start state of lp is s_M, and the end state of lp is $q \in A_M$. Thus $w \in L(M)$. $\square$

According to Proposition 3.5.3, to check if a string w is accepted by a finite automaton M, we simply use our algorithm to generate $\Delta_M(\{s_M\}, w)$, and then check if this set contains at least one accepting state.

Given a finite automaton M, subsets P, R of Q_M and a string w, how do we search for a labeled path that is labeled by w, valid in M, starts from an element of P, and ends with an element of R? What we need to do is associate with each pair

$$(q, y)$$

of the set X that we generate when computing $\Delta_M(P, w)$ a labeled path *lp* such that *lp* is labeled by $\mathbf{pre}(y)$, *lp* is valid in M, the start state of *lp* is an element of P, and the end state of *lp* is q. If we process the elements of X in a breadth-first (rather than depth-first) manner, this will ensure that these labeled paths are as short as possible. As we generate the elements of X, we look for a pair of the form $(q, \%)$, where $q \in R$. Our answer will then be the labeled path associated with this pair.

The Forlan module `FA` also contains the following functions for processing strings and checking string acceptance:

```
val processStr          : fa -> sym set * str -> sym set
val processStrBackwards : fa -> sym set * str -> sym set
val accepted            : fa -> str -> bool
```

The function `processStr` takes in a finite automaton M, and returns a function that takes in a pair (P, w) and returns $\Delta_M(P, w)$. The function `processStrBackwards` is similar, except that it works its way backwards through the w, i.e, acts as if the transitions of M were reversed. The function `accepted` takes in a finite automaton M, and returns a function that checks whether a string x is accepted by M.

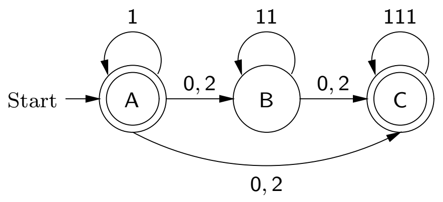
The Forlan module `FA` also contains the following functions for finding labeled paths:

```
val findLP          : fa -> sym set * str * sym set -> lp
val findAcceptingLP : fa -> str -> lp
```

The function `findLP` takes in a finite automaton M, and returns a function that takes in a triple (P, w, R) and tries to find a labeled path *lp* that is labeled by w, valid for M, starts out with an element of P, and ends up at an element of R. It issues an error message when there is no such labeled path. The function `findAcceptingLP` takes in a finite automaton M, and returns a function that looks for a labeled path *lp* that explains why a string

w is accepted by M. It issues an error message when there is no such labeled path. The labeled paths returned by these functions are always of minimal length.

Suppose `fa` is the finite automaton



We begin by applying our five functions to `fa`, and giving names to the resulting functions:

```
- val processStr = FA.processStr fa;
val processStr = fn : sym set * str -> sym set
- val processStrBackwards = FA.processStrBackwards fa;
val processStrBackwards = fn : sym set * str -> sym set
- val accepted = FA.accepted fa;
val accepted = fn : str -> bool
- val findLP = FA.findLP fa;
val findLP = fn : sym set * str * sym set -> lp
- val findAcceptingLP = FA.findAcceptingLP fa;
val findAcceptingLP = fn : str -> lp
```

Next, we'll define a set of states and a string to use later:

```
- val bs = SymSet.input "";
@ A, B, C
@ .
val bs = - : sym set
- val x = Str.input "";
@ 11
@ .
val x = [-,-] : str
```

Here are some example uses of our functions:

```
- SymSet.output("", processStr(bs, x));
A, B
val it = () : unit
- SymSet.output("", processStrBackwards(bs, x));
A, B
val it = () : unit
- accepted(Str.input "");
```

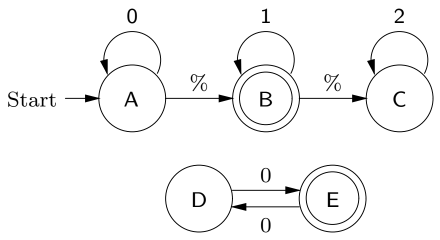
```
@ 12111111
@ .
val it = true : bool
- accepted(Str.input "");
@ 1211
@ .
val it = false : bool
- LP.output("", findLP(bs, x, bs));
B, 11 => B
val it = () : unit
- LP.output("", findAcceptingLP(Str.input ""));
@ 12111111
@ .
A, 1 => A, 2 => C, 111 => C, 111 => C
val it = () : unit
- LP.output("", findAcceptingLP(Str.input ""));
@ 222
@ .
no such labeled path exists

uncaught exception Error
```

3.6 Simplification of Finite Automata

In this section, we: say what it means for a finite automaton to be simplified; study an algorithm for simplifying finite automata; and see how finite automata can be simplified in Forlan.

Suppose M is the finite automaton



M is odd for two distinct reasons. First, there are no valid labeled paths from the start state to D and E, and so these states are redundant. Second, there are no valid labeled paths from C to an accepting state, and so it is also redundant. We will say that C is not "live" (C is "dead"), and that D and E are not "reachable".

Suppose M is a finite automaton. We say that a state $q \in Q_M$ is:

- *reachable* iff there is a labeled path lp such that lp is valid for M, the start state of lp is s_M, and the end state of lp is q;
- *live* iff there is a labeled path lp such that lp is valid for M, the start state of lp is q, and the end state of lp is in A_M;
- *dead* iff q is not live;
- *useful* iff q is both reachable and live.

Let M be our example finite automaton. The reachable states of M are: A, B and C. The live states of M are: A, B, D and E. And, the useful states of M are: A and B.

There is a simple algorithm for generating the set of reachable states of a finite automaton M. We generate the least subset X of Q_M such that:

- $s_M \in X$;
- for all $q, r \in Q_M$ and $x \in \mathbf{Str}$, if $q \in X$ and $(q, x, r) \in T_M$, then $r \in X$.

The start state of M is added to X, since s_M is always reachable, by the zero-length labeled path s_M. Then, if q is reachable, and (q, x, r) is a transition of M, then r is clearly reachable. Thus all of the elements of X are indeed reachable. And, it's not hard to show that every reachable state will be added to X.

Similarly, there is a simple algorithm for generating the set of live states of a finite automaton M. We generate the least subset Y of Q_M such that:

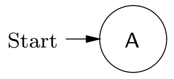
- $A_M \subseteq Y$;
- for all $q, r \in Q_M$ and $x \in \mathbf{Str}$, if $r \in Y$ and $(q, x, r) \in T_M$, then $q \in Y$.

This time it's the accepting states of M that initially added to our set, since each accepting state is trivially live. Then, if r is live, and (q, x, r) is a transition of M, then q is clearly live.

We say that a finite automaton M is *simplified* iff either

- every state of M is useful; or
- $|Q_M| = 1$ and $|A_M| = |T_M| = 0$.

Let N be the finite automaton



Then N is simplified, even though $s_N = \mathsf{A}$ is not live, and thus is not useful.

Proposition 3.6.1
Suppose M is a simplified finite automaton. Then $\textbf{alphabet}(M) = \textbf{alphabet}(L(M))$.

We always have that $\textbf{alphabet}(L(M)) \subseteq \textbf{alphabet}(M)$. But, when M is simplified, we also have that $\textbf{alphabet}(M) \subseteq \textbf{alphabet}(L(M))$, i.e., that every symbol appearing in a string of one of M's transitions also appears in one of the strings accepted by M.

Now we can give an algorithm for simplifying finite automata. We define a function $\textbf{simplify} \in \textbf{FA} \to \textbf{FA}$ by: $\textbf{simplify}(M)$ is the finite automaton N such that:

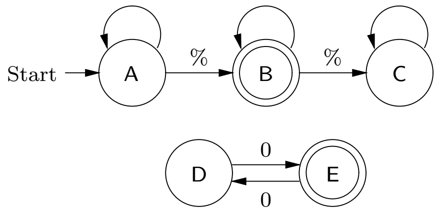
- if s_M is useful in M, then:
  - $Q_N = \{\, q \in Q_M \mid q \text{ is useful in } M \,\}$;
  - $s_N = s_M$;
  - $A_N = A_M \cap Q_N = \{\, q \in A_M \mid q \in Q_N \,\}$;
  - $T_N = \{\, (q, x, r) \in T_M \mid q, r \in Q_N \,\}$; and
- if s_M is not useful in M, then:
  - $Q_N = \{s_M\}$;
  - $s_N = s_M$;
  - $A_N = \emptyset$;
  - $T_N = \emptyset$.

Proposition 3.6.2
Suppose M is a finite automaton. Then:

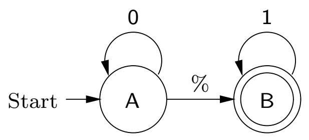
(1) $\textbf{simplify}(M)$ *is simplified;*

(2) $\textbf{simplify}(M) \approx M$.

Suppose M is the finite automaton



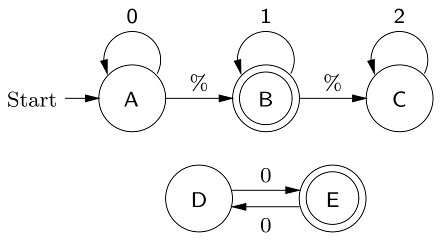
Then **simplify**(M) is the finite automaton



The Forlan module `FA` includes the following function for simplifying finite automata:

```
val simplify : fa -> fa
```

In the following, suppose `fa` is the finite automaton



Here are some example uses of `simplify`:

```
- val fa' = FA.simplify fa;
val fa' = - : fa
- FA.output("", fa');
{states}
A, B
{start state}
A
{accepting states}
B
{transitions}
A, % -> B; A, 0 -> A; B, 1 -> B
val it = () : unit
- val fa'' = FA.input "";
@ {states} A, B {start state} A {accepting states}
@ {transitions} A, 0 -> B; B, 0 -> A
@ .
val fa'' = - : fa
- FA.output("", FA.simplify fa'');
{states}
A
{start state}
```

```
A
{accepting states}

{transitions}

val it = () : unit
```

3.7 Proving the Correctness of Finite Automata

In this section, we consider techniques for proving the correctness of finite automata, i.e., for proving that finite automata accept the languages we want them to. We begin with some propositions concerning the Δ function.

Proposition 3.7.1
Suppose M is a finite automaton.

(1) For all $q \in Q_M$, $q \in \Delta_M(\{q\}, \%)$.

*(2) For all $q, r \in Q_M$ and $w \in$ **Str**, if $(q, w, r) \in T_M$, then $r \in \Delta_M(\{q\}, w)$.*

*(3) For all $p, q, r \in Q_M$ and $x, y \in$ **Str**, if $q \in \Delta_M(\{p\}, x)$ and $r \in \Delta_M(\{q\}, y)$, then $r \in \Delta_M(\{p\}, xy)$.*

Proposition 3.7.2
*Suppose M is a finite automaton. For all $p, r \in Q_M$ and $w \in$ **Str**, if $r \in \Delta_M(\{p\}, w)$, then either:*

- *$r = p$ and $w = \%$; or*
- *there are $q \in Q_M$ and $x, y \in$ **Str** such that $w = xy$, $(p, x, q) \in T_M$ and $r \in \Delta_M(\{q\}, y)$.*

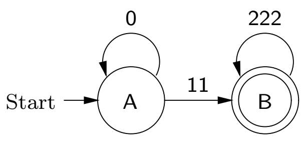
The preceding proposition identifies the first step in a labeled path explaining how one gets from p to r by doing w. In contrast, the following proposition focuses on the last step of such a labeled path.

Proposition 3.7.3
*Suppose M is a finite automaton. For all $p, r \in Q_M$ and $w \in$ **Str**, if $r \in \Delta_M(\{p\}, w)$, then either:*

- *$r = p$ and $w = \%$; or*

- *there are $q \in Q_M$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta_M(\{p\}, x)$ and $(q, y, r) \in T_M$.*

Now we consider a first, almost trivial, example of the correctness proof of an FA. Let M be the finite automaton



To prove that $L(M) = \{0\}^*\{11\}\{222\}^*$, it will suffice to show that $L(M) \subseteq \{0\}^*\{11\}\{222\}^*$ and $\{0\}^*\{11\}\{222\}^* \subseteq L(M)$.

First, we show that $\{0\}^*\{11\}\{222\}^* \subseteq L(M)$; then, we show that $L(M) \subseteq \{0\}^*\{11\}\{222\}^*$.

Lemma 3.7.4
For all $n \in \mathbb{N}$, $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^n)$.

Proof. We proceed by mathematical induction.

(Basis Step) By Proposition 3.7.1(1), we have that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, \%)$. But $0^0 = \%$, and thus $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^0)$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^n)$. We must show that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^{n+1})$. Since $(\mathsf{A}, 0, \mathsf{A}) \in T$, Proposition 3.7.1(2) tells us that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0)$. Since $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0)$ and $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^n)$, Proposition 3.7.1(3) tells us that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 00^n)$. Since $0^{n+1} = 00^n$, it follows that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^{n+1})$. $\square$

Lemma 3.7.5
For all $n \in \mathbb{N}$, $\mathsf{B} \in \Delta(\{\mathsf{B}\}, (222)^n)$.

Proof. Similar to the proof of Lemma 3.7.4. $\square$

Now, suppose $w \in \{0\}^*\{11\}\{222\}^*$. Then $w = 0^n 11(222)^m$ for some $n, m \in \mathbb{N}$. By Lemma 3.7.4, we have that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, 0^n)$. Since $(\mathsf{A}, 11, \mathsf{B}) \in T$, we have that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, 11)$, by Proposition 3.7.1(2). Thus, by Proposition 3.7.1(3), we have that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, 0^n 11)$. By Lemma 3.7.5, we have that $\mathsf{B} \in \Delta(\{\mathsf{B}\}, (222)^m)$. Thus, by Proposition 3.7.1(3), we have that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, 0^n 11(222)^m)$. But $w = 0^n 11(222)^m$, and thus $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. Since A is M's start state and B is an accepting state of M, it follows that $\Delta(\{s_M\}, w) \cap A_M \neq \emptyset$, so that (by Proposition 3.5.3) $w \in L(M)$.

Now we show that $L(M) \subseteq \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$. Since $\mathbf{alphabet}(M) = \{\mathsf{0},\mathsf{1},\mathsf{2}\}$, it will suffice to show that, for all $w \in \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$,

$$\text{if } \mathsf{B} \in \Delta(\{\mathsf{A}\}, w), \text{ then } w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*.$$

(To see that this is so, suppose $w \in L(M)$. Then $\Delta(\{\mathsf{A}\}, w) \cap \{\mathsf{B}\} \neq \emptyset$, so that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. By Proposition 3.3.1, we have that

$$\mathbf{alphabet}(w) \subseteq \mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}(M) = \{\mathsf{0},\mathsf{1},\mathsf{2}\},$$

so that $w \in \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$. Thus $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.)

Unfortunately, if we try to prove the above formula true, using strong string induction, we will get stuck, having a prefix of our string w that takes us from A to A, but not being able to conclude anything useful about this string. For our strong string induction to succeed, we will need to *strengthen* the property of w that we are proving. This leads us to a proof method in which we say, for each state q of the FA at hand, what we know about the strings that take us from the start state of the machine to q.

Lemma 3.7.6
For all $w \in \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$*:*

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$*, then* $w \in \{\mathsf{0}\}^*$*;*

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$*, then* $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.

Proof. We proceed by strong string induction. Suppose $w \in \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$, and assume the inductive hypothesis: for all $x \in \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$, if $|x| < |w|$, then

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$, then $x \in \{\mathsf{0}\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$, then $x \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.

We must show that

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.

(A) Suppose $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. We must show that $w \in \{\mathsf{0}\}^*$. By Proposition 3.7.3, there are two cases to consider.

- Suppose $\mathsf{A} = \mathsf{A}$ and $w = \%$. Then $w = \% \in \{\mathsf{0}\}^*$.

- Suppose there are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{A}) \in T$. Since $(q, y, \mathsf{A}) \in T$, we have that $q = \mathsf{A}$ and $y = \mathsf{0}$, so that $w = x\mathsf{0}$ and $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (A) of the inductive hypothesis tells us that $x \in \{\mathsf{0}\}^*$. Thus $w = x\mathsf{0} \in \{\mathsf{0}\}^*\{\mathsf{0}\} \subseteq \{\mathsf{0}\}^*$.

(B) Suppose $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. We must show that $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$. Since $\mathsf{B} \neq \mathsf{A}$, Proposition 3.7.3 tells us that there are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{B}) \in T$. Thus there are two cases to consider.

- Suppose $q = \mathsf{A}$ and $y = \mathsf{11}$. Thus $w = x\mathsf{11}$ and $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (A) of the inductive hypothesis tells us that $x \in \{\mathsf{0}\}^*$. Thus $w = x\mathsf{11}\% \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.
- Suppose $q = \mathsf{B}$ and $y = \mathsf{222}$. Thus $w = x\mathsf{222}$ and $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (B) of the inductive hypothesis tells us that $x \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$. Thus $w = x\mathsf{222} \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*\{\mathsf{222}\} \subseteq \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.

$\square$

We could also prove $\{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^* \subseteq L(M)$ by strong string induction. To prove that $L(M) \subseteq \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$, we proved that, for all $w \in \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^*$:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*$;

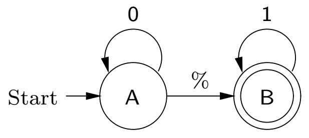
(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$.

To prove that $\{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^* \subseteq L(M)$, we could simply reverse the implications in (A) and (B) of this formula, proving that for all $w \in \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^*$:

(A) if $w \in \{\mathsf{0}\}^*$, then $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$;

(B) if $w \in \{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^*$, then $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$.

As a second example FA correctness proof, suppose N is the finite automaton



To prove that $L(N) = \{\mathsf{0}\}^*\{\mathsf{1}\}^*$, it will suffice to show that $L(N) \subseteq \{\mathsf{0}\}^*\{\mathsf{1}\}^*$ and $\{\mathsf{0}\}^*\{\mathsf{1}\}^* \subseteq L(N)$. The proof that $\{\mathsf{0}\}^*\{\mathsf{1}\}^* \subseteq L(N)$ is similar to our proof that $\{\mathsf{0}\}^*\{\mathsf{11}\}\{\mathsf{222}\}^* \subseteq L(M)$.

To show that $L(N) \subseteq \{\mathsf{0}\}^*\{\mathsf{1}\}^*$, it would suffice to show that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*\{\mathsf{1}\}^*$.

Unfortunately, we can't prove this using strong string induction: because of the transition $(\mathsf{A}, \%, \mathsf{B})$, the proof of Part (B) will fail. Here is how the failed proof begins.

> There are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{B}) \in T$. Since $(q, y, \mathsf{B}) \in T$, there are two cases to consider. Let's consider the case when $q = \mathsf{A}$ and $y = \%$. Then $w = x\%$ and $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$.

Unfortunately, $|x| = |w|$, and so we won't be able to use Part (A) of the inductive hypothesis to conclude that $x \in \{\mathsf{0}\}^*$.

Instead, we must do our proof using mathematical induction on the length of labeled paths. We use mathematical induction to prove that, for all $n \in \mathbb{N}$, for all $w \in \mathbf{Str}$:

(A) if there is an $lp \in \mathbf{LP}$ such that $|lp| = n$, $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{A}$, then $w \in \{\mathsf{0}\}^*$;

(B) if there is an $lp \in \mathbf{LP}$ such that $|lp| = n$, $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{B}$, then $w \in \{\mathsf{0}\}^*\{\mathsf{1}\}^*$.

We can use the above formula to prove $L(N) \subseteq \{\mathsf{0}\}^*\{\mathsf{1}\}^*$, as follows. Suppose $w \in L(N)$. Then, there is an $lp \in \mathbf{LP}$ such that $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{B}$. Let $n = |lp|$. Thus, Part (B) of the above formula holds. Hence $w \in \{\mathsf{0}\}^*\{\mathsf{1}\}^*$.

In the inductive step, we assume that $n \in \mathbb{N}$ and that the inductive hypothesis holds: for all $w \in \mathbf{Str}$, (A) and (B) hold. We must show that for all $w \in \mathbf{Str}$, (A) and (B) hold, where $n + 1$ has been substituted for n. So, we suppose that $w \in \mathbf{Str}$, and show that (A) and (B) hold, where $n + 1$ has been substituted for n:

(A) if there is an $lp \in \mathbf{LP}$ such that $|lp| = n+1$, $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{A}$, then $w \in \{\mathsf{0}\}^*$;

(B) if there is an $lp \in \mathbf{LP}$ such that $|lp| = n + 1$, $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{B}$, then $w \in \{\mathsf{0}\}^*\{\mathsf{1}\}^*$.

Let's consider the proof of Part (B), where $n+1$ has been substituted for n. We assume that there is an $lp \in \mathbf{LP}$ such that $|lp| = n+1$, $\mathbf{label}(lp) = w$, lp is valid for N, $\mathbf{startState}(lp) = \mathsf{A}$ and $\mathbf{endState}(lp) = \mathsf{B}$. Let $lp' \in \mathbf{LP}$ be the first n steps of lp, and let $x = \mathbf{label}(lp')$ and $q = \mathbf{endState}(lp')$. Let y be such that the last step of lp uses the transition $(q, y, \mathsf{B}) \in T$. Then $w = xy$, lp' is valid for N and $\mathbf{startState}(lp') = \mathsf{A}$. Let's consider the case when $q = \mathsf{A}$ and $y = \%$. Then $w = x\%$ and $\mathbf{endState}(lp') = \mathsf{A}$. By the inductive hypothesis, we know that (A) holds, where x has been substituted for w. Since $|lp'| = n$, $\mathbf{label}(lp') = x$, lp' is valid for N, $\mathbf{startState}(lp') = \mathsf{A}$ and $\mathbf{endState}(lp') = \mathsf{A}$, it follows that $x \in \{\mathsf{0}\}^*$. Thus $w = x\% \in \{\mathsf{0}\}^*\{\mathsf{1}\}^*$. The proof of the other case is easy.

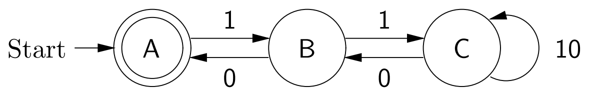
We conclude this section by considering one more example. Recall the definition of the "difference" function that was introduced in Section 2.2: given a string $w \in \{\mathsf{0}, \mathsf{1}\}^*$, we write $\mathbf{diff}(w)$ for

$$\text{the number of } \mathsf{1}\text{'s in } w - \text{the number of } \mathsf{0}\text{'s in } w.$$

Define $X = \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid \mathbf{diff}(w) = 0 \text{ and, for all prefixes } v \text{ of } w,\ 0 \leq \mathbf{diff}(v) \leq 3 \,\}$.

For example, $\mathsf{110100} \in X$, since $\mathbf{diff}(\mathsf{110100}) = 0$ and every prefix of 110100 has a diff between 0 and 3 (the diff's of %, 1, 11, 110, 1101, 11010 and 110100 are 0, 1, 2, 1, 2, 1 and 0, respectively). On the other hand, $\mathsf{1001} \notin X$, even though $\mathbf{diff}(\mathsf{1001}) = 0$, because 100 is a prefix of 1001, $\mathbf{diff}(\mathsf{100}) = -1$ and $-1 < 0$.

First, let's consider the problem of synthesizing an FA M such that $L(M) = X$. What we can do is think of each state of our machine as "keeping track" of the diff of the strings that take us to that state. Because every prefix of an element of X has a diff between 0 and 3, this would lead to our having four states: A, B, C and D, corresponding to diff's of 0, 1, 2 and 3, respectively. If we are in state A, there will be a transition labeled 1 that takes us to B. From B, there will be a transition labeled 0, that takes us back to A, as well as a transition labeled 1, that takes us to C, and so on. It turns out, however, that we can dispense with the state D, by having a transition labeled 10 from C back to itself. Thus, our machine is



Now we show that M is correct, i.e., that $L(M) = X$. Define Z_i, for $i \in \{0, 1, 2\}$, by: $Z_i = \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid \mathbf{diff}(w) = i$ and, for all prefixes v of w, $0 \leq \mathbf{diff}(v) \leq 3\,\}$. Since $Z_0 = X$, it will suffice to show that $L(M) = Z_0$.

Lemma 3.7.7
For all $w \in \{\mathsf{0}, \mathsf{1}\}^*$:

(0) $w \in Z_0$ iff $w = \%$ or $w = x\mathsf{0}$, for some $x \in Z_1$;

(1) $w \in Z_1$ iff $w = x\mathsf{1}$, for some $x \in Z_0$, or $w = x\mathsf{0}$, for some $x \in Z_2$;

(2) $w \in Z_2$ iff $w = x\mathsf{1}$, for some $x \in Z_1$, or $w = x\mathsf{10}$, for some $x \in Z_2$.

Proof. (0, "only if") Suppose $w \in Z_0$. If $w = \%$, then $w = \%$ or $w = x\mathsf{0}$, for some $x \in Z_1$. So, suppose $w \neq \%$. Thus $w = xa$, for some $x \in \{\mathsf{0}, \mathsf{1}\}^*$ and $a \in \{\mathsf{0}, \mathsf{1}\}$.

Suppose, toward a contradiction, that $a = \mathsf{1}$. Then $\mathbf{diff}(x) + 1 = \mathbf{diff}(x\mathsf{1}) = \mathbf{diff}(xa) = \mathbf{diff}(w) = 0$, so that $\mathbf{diff}(x) = -1$. But this is impossible, since x is a prefix of w, and $w \in Z_0$. Thus $a = \mathsf{0}$, so that $w = x\mathsf{0}$.

Since $\mathbf{diff}(x) - 1 = \mathbf{diff}(x\mathsf{0}) = \mathbf{diff}(w) = 0$, we have that $\mathbf{diff}(x) = 1$. To complete the proof that $x \in Z_1$, suppose v is a prefix of x. Thus v is a prefix of w. But $w \in Z_0$, and thus $0 \leq \mathbf{diff}(v) \leq 3$. Since $w = x\mathsf{0}$ and $x \in Z_1$, we have that $w = \%$ or $w = x\mathsf{0}$, for some $x \in Z_1$.

(0, "if") Suppose $w = \%$ or $w = x\mathsf{0}$, for some $x \in Z_1$. There are two cases to consider.

- Suppose $w = \%$. Then $\mathbf{diff}(w) = \mathbf{diff}(\%) = 0$. To complete the proof that $w \in Z_0$, suppose v is a prefix of w. Then $v = \%$, so that $\mathbf{diff}(v) = 0$, and thus $0 \leq \mathbf{diff}(v) \leq 3$.

- Suppose $w = x\mathsf{0}$, for some $x \in Z_1$. Then $\mathbf{diff}(w) = \mathbf{diff}(x\mathsf{0}) = \mathbf{diff}(x) - 1 = 1 - 1 = 0$. To complete the proof that $w \in Z_0$, suppose v is a prefix of w. If v is a prefix of x, then $0 \leq \mathbf{diff}(v) \leq 3$, since $x \in Z_1$. And, if $v = x\mathsf{0} = w$, then $\mathbf{diff}(v) = \mathbf{diff}(w) = 0$, so that $0 \leq \mathbf{diff}(v) \leq 3$.

(1, "only if") Suppose $w \in Z_1$. Then $\mathbf{diff}(w) = 1$, so that $w \neq \%$. There are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in \{\mathsf{0}, \mathsf{1}\}^*$. Since $\mathbf{diff}(x)+1 = \mathbf{diff}(w) = 1$, we have that $\mathbf{diff}(x) = 0$. Since x is a prefix of w and $w \in Z_1$, it follows that $x \in Z_0$. Since $w = x\mathsf{1}$ and $x \in Z_0$, we have that $w = x\mathsf{1}$, for some $x \in Z_0$, or $w = x\mathsf{0}$, for some $x \in Z_2$.

- Suppose $w = x\mathsf{0}$, for some $x \in \{\mathsf{0}, \mathsf{1}\}^*$. Since $\mathbf{diff}(x) - 1 = \mathbf{diff}(w) = 1$, we have that $\mathbf{diff}(x) = 2$. Since x is a prefix of w and $w \in Z_1$, it follows that $x \in Z_2$. Since $w = x\mathsf{0}$ and $x \in Z_2$, we have that $w = x\mathsf{1}$, for some $x \in Z_0$, or $w = x\mathsf{0}$, for some $x \in Z_2$.

(1, "if") Suppose $w = x\mathsf{1}$, for some $x \in Z_0$, or $w = x\mathsf{0}$, for some $x \in Z_2$. There are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in Z_0$. Then $\mathbf{diff}(w) = \mathbf{diff}(x) + 1 = 0 + 1 = 1$. To complete the proof that $w \in Z_1$, suppose v is a prefix of w. If v is a prefix of x, then $0 \leq \mathbf{diff}(v) \leq 3$, since $x \in Z_0$. And, if $v = x\mathsf{1} = w$, then $\mathbf{diff}(v) = \mathbf{diff}(w) = 1$, so that $0 \leq \mathbf{diff}(v) \leq 3$.
- Suppose $w = x\mathsf{0}$, for some $x \in Z_2$. Then $\mathbf{diff}(w) = \mathbf{diff}(x) - 1 = 2 - 1 = 1$. To complete the proof that $w \in Z_1$, suppose v is a prefix of w. If v is a prefix of x, then $0 \leq \mathbf{diff}(v) \leq 3$, since $x \in Z_2$. And, if $v = x\mathsf{0} = w$, then $\mathbf{diff}(v) = \mathbf{diff}(w) = 1$, so that $0 \leq \mathbf{diff}(v) \leq 3$.

(2, "only if") Suppose $w \in Z_2$. Then $\mathbf{diff}(w) = 2$, so that $w \neq \%$. There are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in \{\mathsf{0}, \mathsf{1}\}^*$. Since $\mathbf{diff}(x) + 1 = \mathbf{diff}(w) = 2$, we have that $\mathbf{diff}(x) = 1$. Since x is a prefix of w and $w \in Z_2$, it follows that $x \in Z_1$. Since $w = x\mathsf{1}$ and $x \in Z_1$, we have that $w = x\mathsf{1}$, for some $x \in Z_1$, or $w = x\mathsf{10}$, for some $x \in Z_2$.
- Suppose $w = y\mathsf{0}$, for some $y \in \{\mathsf{0}, \mathsf{1}\}^*$. Since $\mathbf{diff}(y) - 1 = \mathbf{diff}(w) = 2$, we have that $\mathbf{diff}(y) = 3$. Thus $y \neq \%$, so that $y = xa$, for some $x \in \{\mathsf{0}, \mathsf{1}\}^*$ and $a \in \{\mathsf{0}, \mathsf{1}\}$. Hence $w = y\mathsf{0} = xa\mathsf{0}$. If $a = \mathsf{0}$, then $\mathbf{diff}(x) - 1 = \mathbf{diff}(x\mathsf{0}) = \mathbf{diff}(y) = 3$, so that $\mathbf{diff}(x) = 4$. But x is a prefix of w and $w \in Z_2$, and thus this is impossible. Thus $a = \mathsf{1}$, so that $w = x\mathsf{10}$. Since $\mathbf{diff}(x) = \mathbf{diff}(x) + 1 + -1 = \mathbf{diff}(w) = 2$, x is a prefix of w, and $w \in Z_2$, we have that $x \in Z_2$. Since $w = x\mathsf{10}$ and $x \in Z_2$, we have that $w = x\mathsf{1}$, for some $x \in Z_1$, or $w = x\mathsf{10}$, for some $x \in Z_2$.

(2, "if") Suppose $w = x\mathsf{1}$, for some $x \in Z_1$, or $w = x\mathsf{10}$, for some $x \in Z_2$. There are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in Z_1$. Then $\mathbf{diff}(w) = \mathbf{diff}(x) + 1 = 1 + 1 = 2$. To complete the proof that $w \in Z_2$, suppose v is a prefix of w. If v is a prefix of x, then $0 \leq \mathbf{diff}(v) \leq 3$, since $x \in Z_1$. And, if $v = x\mathsf{1} = w$, then $\mathbf{diff}(v) = \mathbf{diff}(w) = 2$, so that $0 \leq \mathbf{diff}(v) \leq 3$.

- Suppose $w = x\mathsf{10}$, for some $x \in Z_2$. Then $\mathbf{diff}(w) = \mathbf{diff}(x) + 1 + -1 = 2 + 1 + -1 = 2$. To complete the proof that $w \in Z_2$, suppose v is a prefix of w. If v is a prefix of x, then $0 \leq \mathbf{diff}(v) \leq 3$, since $x \in Z_2$. And, if $v = x\mathsf{1}$, then $\mathbf{diff}(v) = \mathbf{diff}(x) + 1 = 2 + 1 = 3$, so that $0 \leq \mathbf{diff}(v) \leq 3$. Finally, if $v = x\mathsf{10} = w$, then $\mathbf{diff}(v) = \mathbf{diff}(w) = 2$, so that $0 \leq \mathbf{diff}(v) \leq 3$.

$\square$

Now we prove a lemma that will allow us to establish that $L(M) \subseteq Z_0$.

Lemma 3.7.8
For all $w \in \{\mathsf{0}, \mathsf{1}\}^*$:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_0$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_1$;

(C) if $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_2$.

Proof. 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 $|x| < |w|$, then:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$, then $x \in Z_0$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$, then $x \in Z_1$;

(C) if $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$, then $x \in Z_2$.

We must show that:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_0$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_1$;

(C) if $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in Z_2$.

(A) Suppose $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. We must show that $w \in Z_0$. Since $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, there are two cases to consider.

- Suppose $\mathsf{A} = \mathsf{A}$ and $w = \%$. By Lemma 3.7.7(0), $w \in Z_0$.

- Suppose there are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{A}) \in T$. Since $(q, y, \mathsf{A}) \in T$, we have that $q = \mathsf{B}$ and $y = \mathsf{0}$. Thus $w = x\mathsf{0}$ and $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (B) of the inductive hypothesis tells us that $x \in Z_1$. Hence, by Lemma 3.7.7(0), we have that $w \in Z_0$.

(B) Suppose $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. We must show that $w \in Z_1$. Since $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$ and $\mathsf{A} \neq \mathsf{B}$, we have that there are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{B}) \in T$. Since $(q, y, \mathsf{B}) \in T$, there are two cases to consider.

- Suppose $q = \mathsf{A}$ and $y = \mathsf{1}$. Thus $w = x\mathsf{1}$ and $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (A) of the inductive hypothesis tells us that $x \in Z_0$. Hence, by Lemma 3.7.7(1), we have that $w \in Z_1$.
- Suppose $q = \mathsf{C}$ and $y = \mathsf{0}$. Thus $w = x\mathsf{0}$ and $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (C) of the inductive hypothesis tells us that $x \in Z_2$. Hence, by Lemma 3.7.7(1), we have that $w \in Z_1$.

(C) Suppose $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$. We must show that $w \in Z_2$. Since $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$ and $\mathsf{A} \neq \mathsf{C}$, we have that there are $q \in Q$ and $x, y \in \mathbf{Str}$ such that $w = xy$, $q \in \Delta(\{\mathsf{A}\}, x)$ and $(q, y, \mathsf{C}) \in T$. Since $(q, y, \mathsf{C}) \in T$, there are two cases to consider.

- Suppose $q = \mathsf{B}$ and $y = \mathsf{1}$. Thus $w = x\mathsf{1}$ and $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (B) of the inductive hypothesis tells us that $x \in Z_1$. Hence, by Lemma 3.7.7(2), we have that $w \in Z_2$.
- Suppose $q = \mathsf{C}$ and $y = \mathsf{10}$. Thus $w = x\mathsf{10}$ and $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$. Since $|x| < |w|$, Part (C) of the inductive hypothesis tells us that $x \in Z_2$. Hence, by Lemma 3.7.7(2), we have that $w \in Z_2$.

□

Now, we use the preceding lemma to show that $L(M) \subseteq Z_0$. Suppose $w \in L(M)$. Hence $\Delta(\{\mathsf{A}\}, w) \cap \{\mathsf{A}\} \neq \emptyset$, so that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. Since $\mathbf{alphabet}(M) = \{\mathsf{0}, \mathsf{1}\}$, we have that $w \in \{\mathsf{0}, \mathsf{1}\}^*$. Thus, we have that Parts (A)–(C) of Lemma 3.7.8 hold. By Part (A), it follows that $w \in Z_0$.

To show that $Z_0 \subseteq L(M)$, we show a lemma that is what could be called the converse of Lemma 3.7.8: we simply reverse each of the implications of the lemma's statement.

Lemma 3.7.9
First, we show that, for all $w \in \{\mathsf{0}, \mathsf{1}\}^*$:

(A) if $w \in Z_0$, then $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$;

(B) if $w \in Z_1$, then $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$;

(C) if $w \in Z_2$, then $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$.

Proof. 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 $|x| < |w|$, then:

(A) if $x \in Z_0$, then $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$;

(B) if $x \in Z_1$, then $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$;

(C) if $x \in Z_2$, then $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$.

We must show that:

(A) if $w \in Z_0$, then $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$;

(B) if $w \in Z_1$, then $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$;

(C) if $w \in Z_2$, then $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$.

(A) Suppose $w \in Z_0$. We must show that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. By Lemma 3.7.7(0), there are two cases to consider.

- Suppose $w = \%$. We have that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, \%)$, and thus that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$.
- Suppose $w = x\mathsf{0}$, for some $x \in Z_1$. Since $|x| < |w|$, Part (B) of the inductive hypothesis tells us that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$. Since $(\mathsf{B}, \mathsf{0}, \mathsf{A}) \in T$, we have that $\mathsf{A} \in \Delta(\{\mathsf{B}\}, \mathsf{0})$. Thus $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x\mathsf{0})$, i.e., $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$.

(B) Suppose $w \in Z_1$. We must show that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. By Lemma 3.7.7(1), there are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in Z_0$. Since $|x| < |w|$, Part (A) of the inductive hypothesis tells us that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, x)$. Since $(\mathsf{A}, \mathsf{1}, \mathsf{B}) \in T$, we have that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, \mathsf{1})$. Thus $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x\mathsf{1})$, i.e., $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$.
- Suppose $w = x\mathsf{0}$, for some $x \in Z_2$. Since $|x| < |w|$, Part (C) of the inductive hypothesis tells us that $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$. Since $(\mathsf{C}, \mathsf{0}, \mathsf{B}) \in T$, we have that $\mathsf{B} \in \Delta(\{\mathsf{C}\}, \mathsf{0})$. Thus $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x\mathsf{0})$, i.e., $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$.

(C) Suppose $w \in Z_2$. We must show that $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$. By Lemma 3.7.7(2), there are two cases to consider.

- Suppose $w = x\mathsf{1}$, for some $x \in Z_1$. Since $|x| < |w|$, Part (B) of the inductive hypothesis tells us that $\mathsf{B} \in \Delta(\{\mathsf{A}\}, x)$. Since $(\mathsf{B}, \mathsf{1}, \mathsf{C}) \in T$, we have that $\mathsf{C} \in \Delta(\{\mathsf{B}\}, \mathsf{1})$. Thus $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x\mathsf{1})$, i.e., $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$.
- Suppose $w = x\mathsf{10}$, for some $x \in Z_2$. Since $|x| < |w|$, Part (C) of the inductive hypothesis tells us that $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x)$. Since $(\mathsf{C}, \mathsf{10}, \mathsf{C}) \in T$, we have that $\mathsf{C} \in \Delta(\{\mathsf{C}\}, \mathsf{10})$. Thus $\mathsf{C} \in \Delta(\{\mathsf{A}\}, x\mathsf{10})$, i.e., $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$.

$\square$

Now, we use the preceding lemma to show that $Z_0 \subseteq L(M)$. Suppose $w \in Z_0$. Since $Z_0 \subseteq \{\mathsf{0}, \mathsf{1}\}^*$, we have that Parts (A)–(C) of Lemma 3.7.9 hold. By Part (A), we have that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. Thus $\Delta(\{\mathsf{A}\}, w) \cap \{\mathsf{A}\} \neq \emptyset$, showing that $w \in L(M)$.

Putting Lemmas 3.7.8 and 3.7.9 together, we have that, for all $w \in \{\mathsf{0}, \mathsf{1}\}^*$:

(A) $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$ iff $w \in Z_0$;

(B) $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$ iff $w \in Z_1$;

(C) $\mathsf{C} \in \Delta(\{\mathsf{A}\}, w)$ iff $w \in Z_2$.

Thus, the strings that take us to state A are exactly the ones that are in Z_0, etc.

3.8 Empty-string Finite Automata

In this and the following two sections, we will study three progressively more restricted kinds of finite automata:

- Empty-string finite automata (EFAs);
- Nondeterministic finite automata (NFAs);
- Deterministic finite automata (DFAs).

Every DFA will be an NFA; every NFA will be an EFA; and every EFA will be an FA. Thus, $L(M)$ will be well-defined, if M is a DFA, NFA or EFA. The more restricted kinds of automata will be easier to process on the computer than the more general kinds; they will also have nicer reasoning principles than the more general kinds. We will give algorithms for converting the more general kinds of automata into the more restricted kinds. Thus even the deterministic finite automata will accept the same set of languages as the finite automata. On the other hand, it will sometimes be easier to find one of the more general kinds of automata that accepts a given language rather than one of the more restricted kinds accepting the language. And, there are languages where the smallest DFA accepting the language is much bigger than the smallest FA accepting the language.

In this section, we will focus on EFAs. An *empty-string finite automaton* (EFA) M is a finite automaton such that

$$T_M \subseteq \{\, (q, x, r) \mid q, r \in \textbf{Sym} \text{ and } x \in \textbf{Str} \text{ and } |x| \leq 1 \,\}.$$

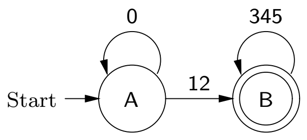
In other words, an FA is an EFA iff every string of every transition of the FA is either % or has a single symbol. For example, $(\mathsf{A}, \%, \mathsf{B})$ and $(\mathsf{A}, \mathsf{1}, \mathsf{B})$ are legal EFA transitions, but $(\mathsf{A}, \mathsf{11}, \mathsf{B})$ is not legal. We write **EFA** for the set of all empty-string finite automata. Thus $\textbf{EFA} \subsetneq \textbf{FA}$.

Now, we consider a proposition that holds for EFAs but not for all FAs.

Proposition 3.8.1
Suppose M is an EFA. For all $p, r \in Q_M$ and $x, y \in \textbf{Str}$, if $r \in \Delta_M(\{p\}, xy)$, then there is a $q \in Q_M$ such that $q \in \Delta_M(\{p\}, x)$ and $r \in \Delta_M(\{q\}, y)$.

Proof. Suppose $p, r \in Q_M$, $x, y \in \textbf{Str}$ and $r \in \Delta_M(\{p\}, xy)$. Thus there is an $lp \in \textbf{LP}$ such that xy is the label of lp, lp is valid for M, the start state of lp is p, and r is the end state of lp. Because M is an EFA, the label of each step of lp is either % or consists of a single symbol. Thus we can divide lp into two labeled paths that explain why $q \in \Delta_M(\{p\}, x)$ and $r \in \Delta_M(\{q\}, y)$, for some $q \in Q_M$. □

To see that this proposition doesn't hold for arbitrary FAs, let M be the FA



Let $x = \mathsf{1}$ and $y = \mathsf{2345}$. Then $xy = (\mathsf{12})(\mathsf{345})$ and so $\mathsf{B} \in \Delta(\{\mathsf{A}\}, xy)$. But there is no $q \in Q$ such that $q \in \Delta_M(\{\mathsf{A}\}, x)$ and $\mathsf{B} \in \Delta_M(\{q\}, y)$, since there is no valid labeled path for M that starts at A and has label $\mathsf{1}$.

The following proposition obviously holds.

Proposition 3.8.2
Suppose M is an EFA.

- *For all $N \in$* **FA**, *if M* **iso** *N, then N is an EFA.*
- *For all bijections f from Q_M to some set of symbols,* **renameStates**(M, f) *is an EFA.*
- **renameStatesCanonically**(M) *is an EFA.*
- **simplify**(M) *is an EFA.*

If we want to convert an FA into an equivalent EFA, we can proceed as follows. Every state of the FA will be a state of the EFA, the start and accepting states are unchanged, and every transition of the FA that is a legal EFA transition will be a transition of the EFA. If our FA has a transition

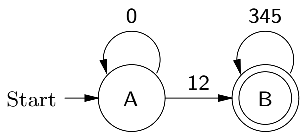
$$(p, b_1b_2 \cdots b_n, r),$$

where $n \geq 2$ and the b_i are symbols, then we replace this transition with the transitions

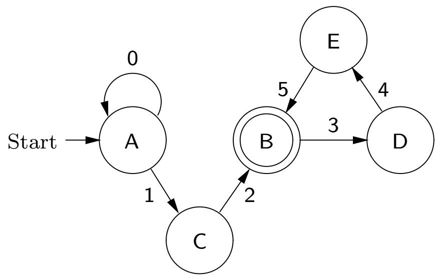
$$(p, b_1, q_1), (q_1, b_2, q_2), \ldots, (q_{n-1}, b_n, r),$$

where $q_1, \ldots, q_{n-1}$ are new, non-accepting, states.

For example, we can convert the FA



into the EFA



In order to turn our informal conversion procedure into an algorithm, we must say how we go about choosing our new states. The symbols we choose can't be states of the original machine, and we can't choose the same symbol twice.

It turns out to be convenient to rename each old state q to $\langle 1, q\rangle$. Then we can replace a transition

$$(p, b_1 b_2 \cdots b_n, r),$$

where $n \geq 2$ and the b_i are symbols, with the transitions

$$\begin{gathered}(\langle 1, p\rangle, b_1, \langle 2, \langle p, b_1, b_2 \cdots b_n, r\rangle\rangle),\\ (\langle 2, \langle p, b_1, b_2 \cdots b_n, r\rangle\rangle, b_2, \langle 2, \langle p, b_1 b_2, b_3 \cdots b_n, r\rangle\rangle),\\ \ldots,\\ (\langle 2, \langle p, b_1 b_2 \cdots b_{n-1}, b_n, r\rangle\rangle, b_n, \langle 1, r\rangle).\end{gathered}$$

We define a function **faToEFA** $\in$ **FA** $\to$ **EFA** that converts FAs into EFAs by saying that **faToEFA**(M) is the result of running the above algorithm on input M.

Theorem 3.8.3
*For all $M \in$ **FA**:*

- **faToEFA**$(M) \approx M$*; and*
- **alphabet**(**faToEFA**(M)) $=$ **alphabet**(M).

Proof. Suppose $M \in$ **FA**, and let $N =$ **faToEFA**(M). Because M and N differ only in that each M-transition of the form

$$(p, b_1 b_2 \cdots b_n, r),$$

where $n \geq 2$ and the b_i are symbols, was replaced by N-transitions of the form

$$(p, b_1, q_1), (q_1, b_2, q_2), \ldots, (q_{n-1}, b_n, r),$$

where $q_1, \ldots, q_{n-1}$ are new, non-accepting, states, it is clear that $L(N) = L(M)$ and **alphabet**$(N) =$ **alphabet**(M). (If the q_i's were preexisting or accepting states, then $L(M) \subseteq L(N)$ would still hold, but there might be elements of $L(M)$ that were not in $L(N)$.) $\square$

The Forlan module `EFA` defines an abstract type `efa` (in the top-level environment) of empty-string finite automata, along with various functions for processing EFAs. Values of type `efa` are implemented as values of type `fa`, and the module EFA provides functions

```
val injToFA    : efa -> fa
val projFromFA : fa -> efa
```

for making a value of type `efa` have type `fa`, i.e., "injecting" an `efa` into type `fa`, and for making a value of type `fa` that is an EFA have type `efa`, i.e., "projecting" an `fa` that is an EFA to type `efa`. If one tries to project an `fa` that is not an EFA to type `efa`, an error is signaled. The functions `injToFA` and `projFromFA` are available in the top-level environment as `injEFAToFA` and `projFAToEFA`, respectively.

The module `EFA` also defines the functions:

```
val input  : string -> efa
val fromFA : fa -> efa
```

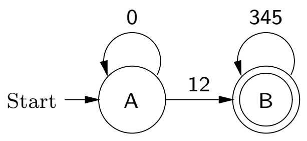
The function `input` is used to input an EFA, i.e., to input a value of type `fa` using `FA.input`, and then attempt to project it to type `efa`. The function `fromFA` corresponds to our conversion function **faToEFA**, and is available in the top-level environment with that name:

```
val faToEFA : fa -> efa
```

Finally, most of the functions for processing FAs that were introduced in previous sections are inherited by `EFA`:

```
val output                  : string * efa -> unit
val numStates               : efa -> int
val numTransitions          : efa -> int
val alphabet                : efa -> sym set
val equal                   : efa * efa -> bool
val isomorphism             : efa * efa * sym_rel -> bool
val findIsomorphism         : efa * efa -> sym_rel
val isomorphic              : efa * efa -> bool
val renameStates            : efa * sym_rel -> efa
val renameStatesCanonically : efa -> efa
val processStr              : efa -> sym set * str -> sym set
val processStrBackwards     : efa -> sym set * str -> sym set
val accepted                : efa -> str -> bool
val checkLP                 : efa -> lp -> unit
val validLP                 : efa -> lp -> bool
val findLP                  : efa -> sym set * str * sym set -> lp
val findAcceptingLP         : efa -> str -> lp
val simplify                : efa -> efa
```

Suppose that fa is the finite automaton



Here are some example uses of a few of the above functions:

```
- projFAToEFA fa;
invalid label in transition : "12"

uncaught exception Error
- val efa = faToEFA fa;
val efa = - : efa
- EFA.output("", efa);
{states}
<1,A>, <1,B>, <2,<A,1,2,B>>, <2,<B,3,45,B>>, <2,<B,34,5,B>>
{start state}
<1,A>
{accepting states}
<1,B>
{transitions}
<1,A>, 0 -> <1,A>; <1,A>, 1 -> <2,<A,1,2,B>>;
<1,B>, 3 -> <2,<B,3,45,B>>; <2,<A,1,2,B>>, 2 -> <1,B>;
<2,<B,3,45,B>>, 4 -> <2,<B,34,5,B>>;
<2,<B,34,5,B>>, 5 -> <1,B>
val it = () : unit
- val efa' = EFA.renameStatesCanonically efa;
val efa' = - : efa
- EFA.output("", efa');
{states}
A, B, C, D, E
{start state}
A
{accepting states}
B
{transitions}
A, 0 -> A; A, 1 -> C; B, 3 -> D; C, 2 -> B; D, 4 -> E;
E, 5 -> B
val it = () : unit
- val rel = EFA.findIsomorphism(efa, efa');
val rel = - : sym_rel
- SymRel.output("", rel);
(<1,A>, A), (<1,B>, B), (<2,<A,1,2,B>>, C),
(<2,<B,3,45,B>>, D), (<2,<B,34,5,B>>, E)
val it = () : unit
- LP.output("", FA.findAcceptingLP fa (Str.input ""));
@ 012345
@ .
A, 0 => A, 12 => B, 345 => B
val it = () : unit
- LP.output("", EFA.findAcceptingLP efa' (Str.input ""));
@ 012345
@ .
A, 0 => A, 1 => C, 2 => B, 3 => D, 4 => E, 5 => B
val it = () : unit
```

3.9 Nondeterministic Finite Automata

In this section, we study the second of our more restricted kinds of finite automata: nondeterministic finite automata. A *nondeterministic finite automaton* (NFA) M is a finite automaton such that

$$T_M \subseteq \{\, (q, x, r) \mid q, r \in \textbf{Sym} \text{ and } x \in \textbf{Str} \text{ and } |x| = 1 \,\}.$$

In other words, an FA is an NFA iff every string of every transition of the FA has a single symbol. For example, $(\mathsf{A}, 1, \mathsf{B})$ is a legal NFA transition, but $(\mathsf{A}, \%, \mathsf{B})$ and $(\mathsf{A}, 11, \mathsf{B})$ are not legal. We write **NFA** for the set of all nondeterministic finite automata. Thus $\textbf{NFA} \subsetneq \textbf{EFA} \subsetneq \textbf{FA}$.

Now we consider several propositions that don't hold for arbitrary EFAs.

Proposition 3.9.1
Suppose M is an NFA. For all $p, q \in Q_M$, if $q \in \Delta(\{p\}, \%)$, then $q = p$.

Proposition 3.9.2
*Suppose M is an NFA. For all $p, r \in Q_M$, $a \in$ **Sym** and $x \in$ **Str**, if $r \in \Delta_M(\{p\}, ax)$, then there is a $q \in Q_M$ such that $(p, a, q) \in T_M$ and $r \in \Delta_M(\{q\}, x)$.*

Proposition 3.9.3
*Suppose M is an NFA. For all $p, r \in Q_M$, $a \in$ **Sym** and $x \in$ **Str**, if $r \in \Delta_M(\{p\}, xa)$, then there is a $q \in Q_M$ such that $q \in \Delta_M(\{p\}, x)$ and $(q, a, r) \in T_M$.*

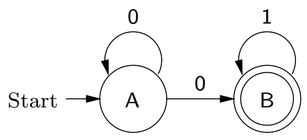
The following proposition obviously holds.

Proposition 3.9.4
Suppose M is an NFA.

- *For all $N \in$ **FA**, if M **iso** N, then N is an NFA.*
- *For all bijections f from Q_M to some set of symbols,* **renameStates**(M, f) *is an NFA.*
- **renameStatesCanonically**(M) *is an NFA.*
- **simplify**(M) *is an NFA.*

Since none of the strings of the transitions of an NFA are %, when proving $L(M) \subseteq X$, for an NFA M and a language X, we can always use strong string induction, instead of having to resort to using induction on the length of labeled paths. In fact, since every string of every transition consists of a single symbol, we can use left string induction rather than strong string induction.

Next, we give an example of an NFA-correctness proof using left string induction. Let M be the NFA



To show that $L(M) = \{0\}^*\{0\}\{1\}^*$, it will suffice to show that $L(M) \subseteq \{0\}^*\{0\}\{1\}^*$ and $\{0\}^*\{0\}\{1\}^* \subseteq L(M)$. We will show the proof of $L(M) \subseteq \{0\}^*\{0\}\{1\}^*$.

Lemma 3.9.5
$L(M) \subseteq \{0\}^*\{0\}\{1\}^*$.

Proof. Since **alphabet**$(M) = \{0, 1\}$, it will suffice to show that, for all $w \in \{0, 1\}^*$:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{0\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{0\}^*\{0\}\{1\}^*$.

We proceed by left string induction.

(Basis Step) We must show that:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, \%)$, then $\% \in \{0\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, \%)$, then $\% \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.

(A) Suppose $\mathsf{A} \in \Delta(\{\mathsf{A}\}, \%)$. Then $\% \in \{\mathsf{0}\}^*$.

(B) Suppose $\mathsf{B} \in \Delta(\{\mathsf{A}\}, \%)$. By Proposition 3.9.1, we have that $\mathsf{B} = \mathsf{A}$—contradiction. Thus $\% \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.

(Inductive Step) Suppose $a \in \{\mathsf{0}, \mathsf{1}\}$ and $w \in \{\mathsf{0}, \mathsf{1}\}^*$. Assume the inductive hypothesis:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$, then $w \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.

We must show that:

(A) if $\mathsf{A} \in \Delta(\{\mathsf{A}\}, wa)$, then $wa \in \{\mathsf{0}\}^*$;

(B) if $\mathsf{B} \in \Delta(\{\mathsf{A}\}, wa)$, then $wa \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.

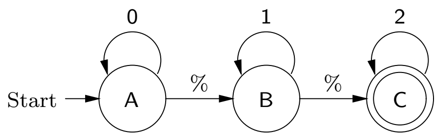
(A) Suppose $\mathsf{A} \in \Delta(\{\mathsf{A}\}, wa)$. We must show that $wa \in \{\mathsf{0}\}^*$. By Proposition 3.9.3, there is a $q \in Q$ such that $q \in \Delta(\{\mathsf{A}\}, w)$ and $(q, a, \mathsf{A}) \in T$. Thus $q = \mathsf{A}$ and $a = \mathsf{0}$, so that $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. By part (A) of the inductive hypothesis, we have that $w \in \{\mathsf{0}\}^*$. Thus $wa = w\mathsf{0} \in \{\mathsf{0}\}^*\{\mathsf{0}\} \subseteq \{\mathsf{0}\}^*$.

(B) Suppose $\mathsf{B} \in \Delta(\{\mathsf{A}\}, wa)$. We must show that $wa \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$. By Proposition 3.9.3, there is a $q \in Q$ such that $q \in \Delta(\{\mathsf{A}\}, w)$ and $(q, a, \mathsf{B}) \in T$. There are two subcases to consider.

- Suppose $q = \mathsf{A}$ and $a = \mathsf{0}$. Then $\mathsf{A} \in \Delta(\{\mathsf{A}\}, w)$. Part (A) of the inductive hypothesis tell us that $w \in \{\mathsf{0}\}^*$. Thus $wa = w\mathsf{0}\% \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.
- Suppose $q = \mathsf{B}$ and $a = \mathsf{1}$. Then $\mathsf{B} \in \Delta(\{\mathsf{A}\}, w)$. Part (B) of the inductive hypothesis tell us that $w \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$. Thus $wa = w\mathsf{1} \in \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*\{\mathsf{1}\} \subseteq \{\mathsf{0}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^*$.

□

Next, we consider the problem of converting EFAs to NFAs. Suppose M is the EFA



To convert M into an equivalent NFA, we will have to:

- replace the transitions $(\mathsf{A}, \%, \mathsf{B})$ and $(\mathsf{B}, \%, \mathsf{C})$ with legal transitions (for example, because of the valid labeled path

$$\mathsf{A} \overset{\%}{\Rightarrow} \mathsf{B} \overset{\mathsf{1}}{\Rightarrow} \mathsf{B} \overset{\%}{\Rightarrow} \mathsf{C},$$

  we will add the transition $(\mathsf{A}, \mathsf{1}, \mathsf{C})$);

- make (at least) A be an accepting state (so that % is accepted by the NFA).

Before defining our general procedure for converting EFAs to NFAs, we first say what we mean by the empty-closure of a set of states. Suppose M is a finite automaton and $P \subseteq Q_M$. The *empty-closure* of P ($\mathbf{emptyClose}_M(P)$) is the least subset X of Q_M such that

- $P \subseteq X$;

- for all $q, r \in Q_M$, if $q \in X$ and $(q, \%, r) \in T_M$, then $r \in X$.

We sometimes abbreviate $\mathbf{emptyClose}_M(P)$ to $\mathbf{emptyClose}(P)$.

For example, if M is our example EFA and $P = \{\mathsf{A}\}$, then:

- $\mathsf{A} \in X$;

- $\mathsf{B} \in X$, since $\mathsf{A} \in X$ and $(\mathsf{A}, \%, \mathsf{B}) \in T_M$;

- $\mathsf{C} \in X$, since $\mathsf{B} \in X$ and $(\mathsf{B}, \%, \mathsf{C}) \in T_M$.

Thus $\mathbf{emptyClose}(P) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

Proposition 3.9.6
Suppose M is a finite automaton. For all $P \subseteq Q_M$, $\mathbf{emptyClose}_M(P) = \Delta_M(P, \%)$.

In other words, $\mathbf{emptyClose}_M(P)$ is all of the states that can be reached from elements of P by sequences of empty moves.

Next, we consider backwards empty-closure. Suppose M is a finite automaton and $P \subseteq Q_M$. The *backwards empty-closure* of P ($\mathbf{emptyCloseBackwards}_M(P)$) is the least subset X of Q_M such that

- $P \subseteq X$;

- for all $q, r \in Q_M$, if $r \in X$ and $(q, \%, r) \in T_M$, then $q \in X$.

We sometimes abbreviate $\mathbf{emptyCloseBackwards}_M(P)$ to $\mathbf{emptyCloseBackwards}(P)$.

For example, if M is our example EFA and $P = \{\mathsf{C}\}$, then:

- $\mathsf{C} \in X$;
- $\mathsf{B} \in X$, since $\mathsf{C} \in X$ and $(\mathsf{B}, \%, \mathsf{C}) \in T_M$;
- $\mathsf{A} \in X$, since $\mathsf{B} \in X$ and $(\mathsf{A}, \%, \mathsf{B}) \in T_M$.

Thus $\mathbf{emptyCloseBackwards}(P) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

Proposition 3.9.7
Suppose M is a finite automaton. For all $P \subseteq Q_M$,
$\mathbf{emptyCloseBackwards}_M(P) = \{\, q \in Q_M \mid \Delta_M(\{q\}, \%) \cap P \neq \emptyset \,\}$.

In other words, $\mathbf{emptyCloseBackwards}_M(P)$ is all of the states from which it is possible to reach elements of P by sequences of empty moves.

Now we use our auxiliary functions in order to define our algorithm for converting EFAs to NFAs. We define a function $\mathbf{efaToNFA} \in \mathbf{EFA} \rightarrow \mathbf{NFA}$ that converts EFAs into NFAs by saying that $\mathbf{efaToNFA}(M)$ is the NFA N such that:

- $Q_N = Q_M$;
- $s_N = s_M$;
- $A_N = \mathbf{emptyCloseBackwards}_M(A_M)$;
- T_N is the set of all triples (q', a, r') such that $q', r' \in Q_M$, $a \in \mathbf{Sym}$, and there are $q, r \in Q_M$ such that:
  - $(q, a, r) \in T_M$;
  - $q' \in \mathbf{emptyCloseBackwards}_M(\{q\})$; and
  - $r' \in \mathbf{emptyClose}_M(\{r\})$.

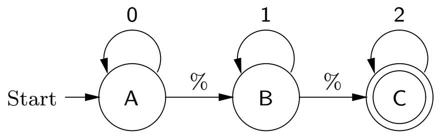
If, in the definition of T_N, we had required that $r' = r$, then N would still have been equivalent to M. Our definition has the advantage of being symmetric.

To compute the set T_N, we process each transition (q, x, r) of M as follows. If $x = \%$, then we generate no transitions. Otherwise, our transition is (q, a, r) for some symbol a. We then compute the backwards empty-closure of $\{q\}$, and call the result X, and compute the (forwards) empty-closure of $\{r\}$, and call the result Y. We then add all of the elements of

$$\{\,(q', a, r') \mid q' \in X \text{ and } r' \in Y\,\}$$

to T_N.

Let M be our example EFA



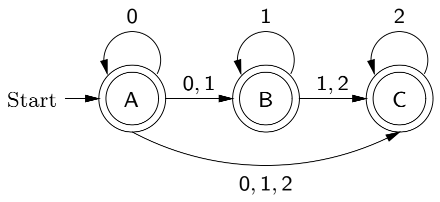
and let $N = \textbf{efaToNFA}(M)$. Then

- $Q_N = Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$;
- $s_N = s_M = \mathsf{A}$;
- $A_N = \textbf{emptyCloseBackwards}_M(A_M) =$ $\textbf{emptyCloseBackwards}_M(\{\mathsf{C}\}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

Now, let's work out what T_N is, by processing each of M's transitions.

- From the transitions $(\mathsf{A}, \%, \mathsf{B})$ and $(\mathsf{B}, \%, \mathsf{C})$, we get no elements of T_N.
- Consider the transition $(\mathsf{A}, 0, \mathsf{A})$. Since $\textbf{emptyCloseBackwards}_M(\{\mathsf{A}\}) = \{\mathsf{A}\}$ and $\textbf{emptyClose}_M(\{\mathsf{A}\}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, we add $(\mathsf{A}, 0, \mathsf{A})$, $(\mathsf{A}, 0, \mathsf{B})$ and $(\mathsf{A}, 0, \mathsf{C})$ to T_N.
- Consider the transition $(\mathsf{B}, 1, \mathsf{B})$. Since $\textbf{emptyCloseBackwards}_M(\{\mathsf{B}\}) = \{\mathsf{A}, \mathsf{B}\}$ and $\textbf{emptyClose}_M(\{\mathsf{B}\}) = \{\mathsf{B}, \mathsf{C}\}$, we add $(\mathsf{A}, 1, \mathsf{B})$, $(\mathsf{A}, 1, \mathsf{C})$, $(\mathsf{B}, 1, \mathsf{B})$ and $(\mathsf{B}, 1, \mathsf{C})$ to T_N.
- Consider the transition $(\mathsf{C}, 2, \mathsf{C})$. Since $\textbf{emptyCloseBackwards}_M(\{\mathsf{C}\}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and $\textbf{emptyClose}_M(\{\mathsf{C}\}) = \{\mathsf{C}\}$, we add $(\mathsf{A}, 2, \mathsf{C})$, $(\mathsf{B}, 2, \mathsf{C})$ and $(\mathsf{C}, 2, \mathsf{C})$ to T_N.

Thus our NFA N is



Theorem 3.9.8
*For all $M \in$ **EFA**:*

- **efaToNFA**$(M) \approx M$*; and*
- **alphabet**(**efaToNFA**(M)) $=$ **alphabet**(M).

Proof. Suppose $M \in$ **EFA** and let $N =$ **efaToNFA**(M). Because each transition (q, a, r) of M is turned into one or more N-transitions with string a, it's clear that **alphabet**$(N) =$ **alphabet**(M). It remains to show that $L(N) = L(M)$.

Suppose $w \in L(N)$, so that there is an *lp* $\in$ **LP** such that the label of *lp* is w, *lp* is valid for N, the start state of *lp* is the start state of N (which is also the start state of M), and the end state of *lp* is an accepting state of N. To show that $w \in L(M)$, we explain how *lp* may be turned into a valid labeled path of M with the same label and start state, and with an end state that is one of M's accepting states. If we are at the end, q, of our labeled path *lp*, then $q \in A_N =$ **emptyCloseBackwards**$_M(A_M)$, so that we can tack on to the labeled path we are constructing enough %-transitions to take us from q to an accepting state of M. Otherwise, we have a step of *lp* corresponding to an N-transition (q', a, r'). Then there are $q, r \in Q_M$ such that:

- $(q, a, r) \in T_M$;
- $q' \in$ **emptyCloseBackwards**$_M(\{q\})$; and
- $r' \in$ **emptyClose**$_M(\{r\})$.

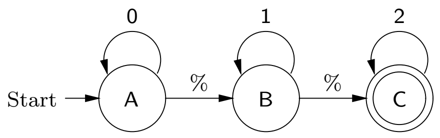
Thus the step corresponding to (q', a, r') may be expanded into the following sequence of steps corresponding to M-transitions. We start with the %-transitions that take us from q' to q. Then we use (q, a, r). Then we use the %-transitions taking us from r to r'. (We could turn all of this into an induction on the length of *lp*.)

Suppose $w \in L(M)$, so that there is an *lp* $\in$ **LP** such that the label of *lp* is w, *lp* is valid for M, the start state of *lp* is the start state of M (which is also the start state of N), and the end state of *lp* is an accepting state of M. To show that $w \in L(N)$, we explain how *lp* may be turned into a valid labeled path of N with the same label and start state, and with an end state that's one of N's accepting states. At some stage of this conversion process, suppose q' is the next state of *lp* to be dealt with. First, we pull off as many %-transitions as possible, taking us to a state q. Thus, we have that $q' \in \mathbf{emptyCloseBackwards}_M(\{q\})$. If this exhausts our labeled path *lp*, then $q \in A_M$, so that $q' \in \mathbf{emptyCloseBackwards}_M(A_M) = A_N$; thus, we add nothing to the labeled path we are constructing, and we are done. Otherwise, the next step in *lp* uses an M-transition (q, a, r). Since $r \in \mathbf{emptyClose}_M(\{r\})$, we have that $(q', a, r) \in T_N$. Thus the next step of the labeled path we're constructing uses this transition. (We could turn all of this into an induction of the length of *lp*.) $\square$

The Forlan module `FA` defines the following functions for computing forwards and backwards empty-closures:

```
val emptyClose          : fa -> sym set -> sym set
val emptyCloseBackwards : fa -> sym set -> sym set
```

It turns out that, `emptyClose` is implemented using `processStr`, and `emptyCloseBackwards` is implemented using `processStrBackwards`. For example, if `fa` is bound to the finite automaton



then we can compute the empty-closure of $\{\mathsf{A}\}$ as follows:

```
- SymSet.output("", FA.emptyClose fa (SymSet.input ""));
@ A
@ .
A, B, C
val it = () : unit
```

The Forlan module `NFA` defines an abstract type `nfa` (in the top-level environment) of nondeterministic finite automata, along with various functions for processing NFAs. Values of type `nfa` are implemented as values of type `fa`, and the module NFA provides the following injection and projection functions:

```
val injToFA     : nfa -> fa
val injToEFA    : nfa -> efa
val projFromFA  : fa -> nfa
val projFromEFA : efa -> nfa
```

The functions `injToFA`, `injToEFA`, `projFromFA` and `projFromEFA` are available in the top-level environment as `injNFAToFA`, `injNFAToEFA`, `projFAToNFA` and `projEFAToNFA`, respectively.

The module `NFA` also defines the functions:

```
val input   : string -> nfa
val fromEFA : efa -> nfa
```

The function `input` is used to input an NFA, and the function `fromEFA` corresponds to our conversion function **efaToNFA**, and is available in the top-level environment with that name:

```
val efaToNFA : efa -> nfa
```

Most of the functions for processing FAs that were introduced in previous sections are inherited by `NFA`:

```
val output                   : string * nfa -> unit
val numStates                : nfa -> int
val numTransitions           : nfa -> int
val alphabet                 : nfa -> sym set
val equal                    : nfa * nfa -> bool
val isomorphism              : nfa * nfa * sym_rel -> bool
val findIsomorphism          : nfa * nfa -> sym_rel
val isomorphic               : nfa * nfa -> bool
val renameStates             : nfa * sym_rel -> nfa
val renameStatesCanonically : nfa -> nfa
val processStr               : nfa -> sym set * str -> sym set
val processStrBackwards      : nfa -> sym set * str -> sym set
val accepted                 : nfa -> str -> bool
val checkLP                  : nfa -> lp -> unit
val validLP                  : nfa -> lp -> bool
val findLP                   : nfa -> sym set * str * sym set -> lp
val findAcceptingLP          : nfa -> str -> lp
val simplify                 : nfa -> nfa
```

Finally, the functions for computing forwards and backwards empty-closures are inherited by the EFA module

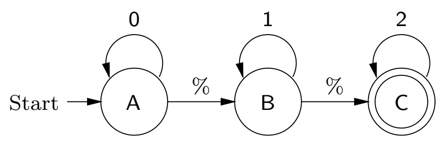
```
val emptyClose          : efa -> sym set -> sym set
val emptyCloseBackwards : efa -> sym set -> sym set
```

and by the NFA module

```
val emptyClose          : nfa -> sym set -> sym set
val emptyCloseBackwards : nfa -> sym set -> sym set
```

(of course, the NFA versions of these functions don't do anything interesting).

Suppose that `efa` is the `efa`



Here are some example uses of a few of the above functions:

```
- projEFAToNFA efa;
invalid label in transition : "%"

uncaught exception Error
- val nfa = efaToNFA efa;
val nfa = - : nfa
- NFA.output("", nfa);
{states}
A, B, C
{start state}
A
{accepting states}
A, B, C
{transitions}
A, 0 -> A | B | C; A, 1 -> B | C; A, 2 -> C;
B, 1 -> B | C; B, 2 -> C; C, 2 -> C
val it = () : unit
- LP.output("", EFA.findAcceptingLP efa (Str.input ""));
@ 012
@ .
A, 0 => A, % => B, 1 => B, % => C, 2 => C
val it = () : unit
- LP.output("", NFA.findAcceptingLP nfa (Str.input ""));
@ 012
@ .
A, 0 => A, 1 => B, 2 => C
val it = () : unit
```

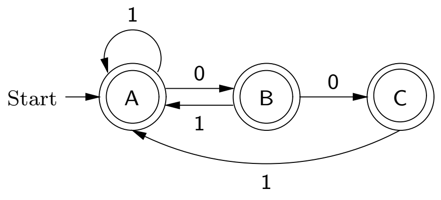
3.10 Deterministic Finite Automata

In this section, we study the third of our more restricted kinds of finite automata: deterministic finite automata. A *deterministic finite automaton* (DFA) M is a finite automaton such that:

- $T_M \subseteq \{\, (q, x, r) \mid q, r \in \textbf{Sym} \text{ and } x \in \textbf{Str} \text{ and } |x| = 1 \,\}$;
- for all $q \in Q_M$ and $a \in \textbf{alphabet}(M)$, there is a unique $r \in Q_M$ such that $(q, a, r) \in T_M$.

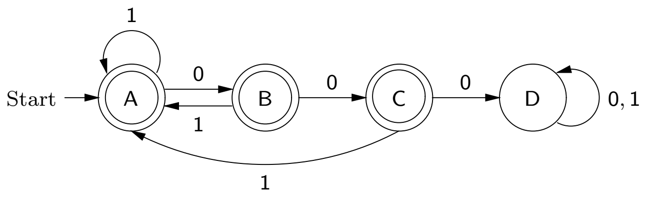
In other words, an FA is a DFA iff it is an NFA and, for every state q of the automaton and every symbol a of the automaton's alphabet, there is exactly one state that can be entered from state q by reading a from the automaton's input. We write **DFA** for the set of all deterministic finite automata. Thus $\mathbf{DFA} \subsetneq \mathbf{NFA} \subsetneq \mathbf{EFA} \subsetneq \mathbf{FA}$.

Let M be the finite automaton



It turns out, as we will later prove, that $L(M) = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathsf{000}$ is not a substring of $w \,\}$. M is almost a DFA; there is never more than one way of processing a symbol from one of M's states. On the other hand, there is no transition of the form $(\mathsf{C}, \mathsf{0}, r)$, and so M is not a DFA, since $\mathsf{0} \in \mathbf{alphabet}(M)$.

We can make M into a DFA by adding a dead state D:



We will never need more than one dead state in a DFA.

The following proposition obviously holds.

Proposition 3.10.1
Suppose M is a DFA.

- *For all $N \in \mathbf{FA}$, if M $\mathbf{iso}$ N, then N is a DFA.*
- *For all bijections f from Q_M to some set of symbols, $\mathbf{renameStates}(M, f)$ is a DFA.*
- *$\mathbf{renameStatesCanonically}(M)$ is a DFA.*

Now we prove a proposition that doesn't hold for arbitrary NFAs.

Proposition 3.10.2
Suppose M is a DFA. For all $q \in Q_M$ and $w \in \mathbf{alphabet}(M)^*$, $|\Delta_M(\{q\}, w)| = 1$.

Proof. An easy left string induction on w. $\square$

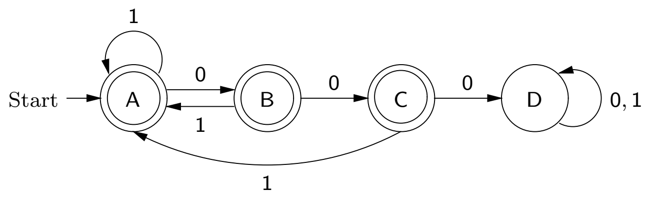
Suppose M is a DFA. Because of Proposition 3.10.2, we can define a function $\delta_M \in Q_M \times \textbf{alphabet}(M)^* \to Q_M$ by:

$$\delta_M(q, w) = \text{the unique } r \in Q_M \text{ such that } r \in \Delta_M(\{q\}, w).$$

In other words, $\delta_M(q, w)$ is the unique state r of M that is the end of a valid labeled path for M that starts at q and is labeled by w. Thus, for all $q, r \in Q_M$ and $w \in \textbf{alphabet}(M)^*$,

$$\delta_M(q, w) = r \qquad \text{iff} \qquad r \in \Delta_M(\{q\}, w).$$

We sometimes abbreviate $\delta_M(q, w)$ to $\delta(q, w)$. For example, if M is the DFA



then

- $\delta(\mathsf{A}, \%) = \mathsf{A}$;
- $\delta(\mathsf{A}, \mathsf{0100}) = \mathsf{C}$;
- $\delta(\mathsf{B}, \mathsf{000100}) = \mathsf{D}$.

Having defined the δ function, we can study its properties.

Proposition 3.10.3
Suppose M is a DFA.

(1) *For all $q \in Q_M$, $\delta_M(q, \%) = q$.*

(2) *For all $q \in Q_M$ and $a \in \textbf{alphabet}(M)$, $\delta_M(q, a) =$ the unique $r \in Q_M$ such that $(q, a, r) \in T_M$.*

(3) *For all $q \in Q_M$ and $x, y \in \textbf{alphabet}(M)^*$, $\delta_M(q, xy) = \delta_M(\delta_M(q, x), y)$.*

Suppose M is a DFA. By Part (2) of the preceding proposition, we have that, for all $q, r \in Q_M$ and $a \in \mathbf{alphabet}(M)$,

$$\delta_M(q, a) = r \qquad \text{iff} \qquad (q, a, r) \in T_M.$$

Now we can use the δ function to explain when a string is accepted by an FA.

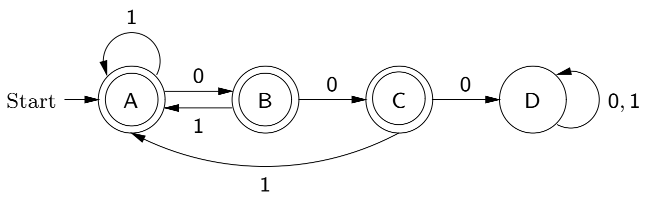
Proposition 3.10.4
Suppose M is a DFA. $L(M) = \{\, w \in \mathbf{alphabet}(M)^* \mid \delta_M(s_M, w) \in A_M \,\}$.

Proof. Let $X = \{\, w \in \mathbf{alphabet}(M)^* \mid \delta_M(s_M, w) \in A_M \,\}$. We must show that $L(M) \subseteq X \subseteq L(M)$.

($L(M) \subseteq X$) Suppose $w \in L(M)$. Then $w \in \mathbf{alphabet}(M)^*$ and there is a $q \in A_M$ such that $q \in \Delta_M(\{s_M\}, w)$. Thus $\delta_M(s_M, w) = q \in A_M$, so that $w \in X$.

($X \subseteq L(M)$) Suppose $w \in X$, so that $w \in \mathbf{alphabet}(M)^*$ and $\delta_M(s_M, w) \in A_M$. Then $\delta_M(s_M, w) \in \Delta_M(\{s_M\}, w)$, and thus $w \in L(M)$. □

The preceding propositions give us an efficient algorithm for checking whether a string is accepted by a DFA. For example, suppose M is the DFA



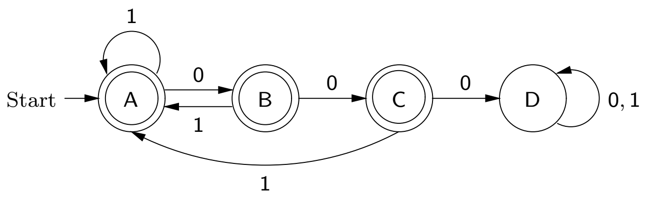
To check whether 0100 is accepted by M, we need to determine whether $\delta(\mathsf{A}, \mathsf{0100}) \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. We have that:

$$\begin{aligned}
\delta(\mathsf{A}, \mathsf{0100}) &= \delta(\delta(\mathsf{A}, \mathsf{0}), \mathsf{100}) \\
&= \delta(\mathsf{B}, \mathsf{100}) \\
&= \delta(\delta(\mathsf{B}, \mathsf{1}), \mathsf{00}) \\
&= \delta(\mathsf{A}, \mathsf{00}) \\
&= \delta(\delta(\mathsf{A}, \mathsf{0}), \mathsf{0}) \\
&= \delta(\mathsf{B}, \mathsf{0}) \\
&= \mathsf{C} \\
&\in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.
\end{aligned}$$

Thus $\mathsf{0100}$ is accepted by M.

Since every DFA is an NFA, we could prove the correctness of DFAs using the techniques that we have already studied. We can often avoid a lot of work, however, by exploiting the fact that we are working with DFAs. It will also be convenient to express things using the δ_M function rather than the Δ_M function.

Next, we do an example DFA correctness proof. Suppose M is the DFA



and let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathsf{000}$ is not a substring of $w\,\}$. We will show that $L(M) = X$.

Lemma 3.10.5
For all $w \in \{\mathsf{0},\mathsf{1}\}^*$:

(A) if $\delta(\mathsf{A}, w) = \mathsf{A}$, then $w \in X$ and $\mathsf{0}$ is not a suffix of w;

(B) if $\delta(\mathsf{A}, w) = \mathsf{B}$, then $w \in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of w;

(C) if $\delta(\mathsf{A}, w) = \mathsf{C}$, then $w \in X$ and $\mathsf{00}$ is a suffix of w;

(D) if $\delta(\mathsf{A}, w) = \mathsf{D}$, then $w \notin X$.

Because A, B and C are accepting states, it's important that the right-sides of (A)–(C) imply that $w \in X$. And, since D is not an accepting state, it's important that the right-side of (D) implies that $w \notin X$. The rest of the right-sides of (A)–(C) have been chosen so that it's possible to prove the lemma by left string induction.

Proof. We proceed by left string induction.

(Basis Step) We must show that

(A) if $\delta(\mathsf{A}, \%) = \mathsf{A}$, then $\% \in X$ and $\mathsf{0}$ is not a suffix of $\%$;

(B) if $\delta(\mathsf{A}, \%) = \mathsf{B}$, then $\% \in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of $\%$;

(C) if $\delta(\mathsf{A}, \%) = \mathsf{C}$, then $\% \in X$ and $\mathsf{00}$ is a suffix of $\%$;

(D) if $\delta(\mathsf{A}, \%) = \mathsf{D}$, then $\% \notin X$.

Part (A) holds since % has no $\mathsf{0}$'s. Parts (B)–(D) hold "vacuously", since $\delta(\mathsf{A},\%)=\mathsf{A}$, by Proposition 3.10.3(1).

(Inductive Step) Suppose $a\in\{\mathsf{0},\mathsf{1}\}$ and $w\in\{\mathsf{0},\mathsf{1}\}^*$. Assume the inductive hypothesis:

(A) if $\delta(\mathsf{A},w)=\mathsf{A}$, then $w\in X$ and $\mathsf{0}$ is not a suffix of w;

(B) if $\delta(\mathsf{A},w)=\mathsf{B}$, then $w\in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of w;

(C) if $\delta(\mathsf{A},w)=\mathsf{C}$, then $w\in X$ and $\mathsf{00}$ is a suffix of w;

(D) if $\delta(\mathsf{A},w)=\mathsf{D}$, then $w\not\in X$.

We must show that:

(A) if $\delta(\mathsf{A},wa)=\mathsf{A}$, then $wa\in X$ and $\mathsf{0}$ is not a suffix of wa;

(B) if $\delta(\mathsf{A},wa)=\mathsf{B}$, then $wa\in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of wa;

(C) if $\delta(\mathsf{A},wa)=\mathsf{C}$, then $wa\in X$ and $\mathsf{00}$ is a suffix of wa;

(D) if $\delta(\mathsf{A},wa)=\mathsf{D}$, then $wa\not\in X$.

(A) Suppose $\delta(\mathsf{A},wa)=\mathsf{A}$. We must show that $wa\in X$ and $\mathsf{0}$ is not a suffix of wa. By Proposition 3.10.3(3), we have that $\delta(\delta(\mathsf{A},w),a)=\delta(\mathsf{A},wa)=\mathsf{A}$. Thus $(\delta(\mathsf{A},w),a,\mathsf{A})\in T$, by Proposition 3.10.3(2). Thus there are three cases to consider.

- Suppose $\delta(\mathsf{A},w)=\mathsf{A}$ and $a=\mathsf{1}$. By Part (A) of the inductive hypothesis, we have that $w\in X$. Thus $wa=w\mathsf{1}\in X$ and $\mathsf{0}$ is not a suffix of $w\mathsf{1}=wa$.

- Suppose $\delta(\mathsf{A},w)=\mathsf{B}$ and $a=\mathsf{1}$. By Part (B) of the inductive hypothesis, we have that $w\in X$. Thus $wa=w\mathsf{1}\in X$ and $\mathsf{0}$ is not a suffix of $w\mathsf{1}=wa$.

- Suppose $\delta(\mathsf{A},w)=\mathsf{C}$ and $a=\mathsf{1}$. By Part (C) of the inductive hypothesis, we have that $w\in X$. Thus $wa=w\mathsf{1}\in X$ and $\mathsf{0}$ is not a suffix of $w\mathsf{1}=wa$.

(B) Suppose $\delta(\mathsf{A},wa)=\mathsf{B}$. We must show that $wa\in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of wa. Since $\delta(\delta(\mathsf{A},w),a)=\delta(\mathsf{A},wa)=\mathsf{B}$, we have that $(\delta(\mathsf{A},w),a,\mathsf{B})\in T$. Thus $\delta(\mathsf{A},w)=\mathsf{A}$ and $a=\mathsf{0}$. By Part (A) of the inductive hypothesis, we have that $w \in X$ and $\mathsf{0}$ is not a suffix of w. Thus $wa = w\mathsf{0} \in X$, $\mathsf{0}$ is a suffix of $w\mathsf{0} = wa$, and $\mathsf{00}$ is not a suffix of $w\mathsf{0} = wa$.

(C) Suppose $\delta(\mathsf{A}, wa) = \mathsf{C}$. We must show that $wa \in X$ and $\mathsf{00}$ is a suffix of wa. Since $\delta(\delta(\mathsf{A}, w), a) = \delta(\mathsf{A}, wa) = \mathsf{C}$, we have that $(\delta(\mathsf{A}, w), a, \mathsf{C}) \in T$. Thus $\delta(\mathsf{A}, w) = \mathsf{B}$ and $a = \mathsf{0}$. By Part (B) of the inductive hypothesis, we have that $w \in X$ and $\mathsf{0}$, but not $\mathsf{00}$, is a suffix of w. Since $w \in X$ and $\mathsf{00}$ is not a suffix of w, we have that $wa = w\mathsf{0} \in X$. And, since $\mathsf{0}$ is a suffix of w, we have that $\mathsf{00}$ is a suffix of $w\mathsf{0} = wa$.

(D) Suppose $\delta(\mathsf{A}, wa) = \mathsf{D}$. We must show that $wa \not\in X$. Since $\delta(\delta(\mathsf{A}, w), a) = \delta(\mathsf{A}, wa) = \mathsf{D}$, we have that $(\delta(\mathsf{A}, w), a, \mathsf{D}) \in T$. Thus there are three cases to consider.

- Suppose $\delta(\mathsf{A}, w) = \mathsf{C}$ and $a = \mathsf{0}$. By Part (C) of the inductive hypothesis, we have that $\mathsf{00}$ is a suffix of w. Thus $wa = w\mathsf{0} \not\in X$.
- Suppose $\delta(\mathsf{A}, w) = \mathsf{D}$ and $a = \mathsf{0}$. By Part (D) of the inductive hypothesis, we have that $w \not\in X$. Thus $wa \not\in X$.
- Suppose $\delta(\mathsf{A}, w) = \mathsf{D}$ and $a = \mathsf{1}$. By Part (D) of the inductive hypothesis, we have that $w \not\in X$. Thus $wa \not\in X$.

□

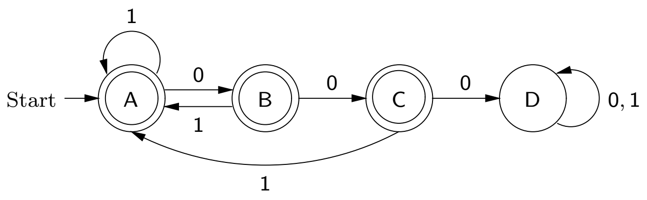
Now, we use the result of the preceding lemma to show that $L(M) = X$. Note how we are able to show that $X \subseteq L(M)$ using proof-by-contradiction.

Lemma 3.10.6
$L(M) = X$.

Proof. $(L(M) \subseteq X)$ Suppose $w \in L(M)$. Then $w \in \textbf{alphabet}(M)^* = \{\mathsf{0}, \mathsf{1}\}^*$ and $\delta(\mathsf{A}, w) \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. By Parts (A)–(C) of Lemma 3.10.5, we have that $w \in X$.

$(X \subseteq L(M))$ Suppose $w \in X$. Since $X \subseteq \{\mathsf{0}, \mathsf{1}\}^*$, we have that $w \in \{\mathsf{0}, \mathsf{1}\}^*$. Suppose, toward a contradiction, that $w \not\in L(M)$. Thus $\delta(\mathsf{A}, w) \not\in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, so that $\delta(\mathsf{A}, w) = \mathsf{D}$. But then Part (D) of Lemma 3.10.5 tells us that $w \not\in X$—contradiction. Thus $w \in L(M)$. □

Next, we consider the simplification of DFAs. Let M be our example DFA



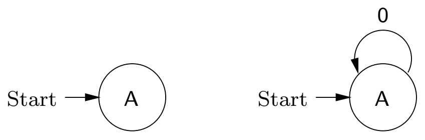
Then M is not simplified, since the state D is dead. But if we get rid of D, then we won't have a DFA anymore. Thus, we will need:

- a notion of when a DFA is simplified that is more liberal than our standard notion;
- a corresponding simplification procedure for DFAs.

We say that a DFA M is *deterministically simplified* iff

- every element of Q_M is reachable; and
- at most one element of Q_M is dead.

For example, consider the following DFAs, which both accept $\emptyset$:



(M_1) (M_2)

M_1 is simplified (recall that any FA with a single state and no transitions is simplified), but M_2 is not simplified. On the other hand, both of these DFAs are deterministically simplified, since A is reachable in both machines, and is the only dead state of each machine.

We define a simplification algorithm for DFAs that takes in

- a DFA M and
- an alphabet Σ

and returns a DFA N such that

- N is deterministically simplified,
- N is equivalent to M,
- $\mathbf{alphabet}(N) = \mathbf{alphabet}(L(M)) \cup \Sigma$.

Thus, the alphabet of N will consist of all symbols that either appear in strings that are accepted by M or are in Σ.

We begin by letting the FA M' be $\mathbf{simplify}(M)$, i.e., the result of running our simplification algorithm for FAs on M. M' will have the following properties.

- M' is simplified.
- $M' \approx M$.
- $\mathbf{alphabet}(M') = \mathbf{alphabet}(L(M')) = \mathbf{alphabet}(L(M))$.
- For all $q \in Q_{M'}$ and $a \in \mathbf{alphabet}(M')$, there is at most one $r \in Q_{M'}$ such that $(q, a, r) \in T_{M'}$. This property holds since M is a DFA and M' was formed by removing states and transitions from M.

Let $\Sigma' = \mathbf{alphabet}(M') \cup \Sigma = \mathbf{alphabet}(L(M)) \cup \Sigma$. If M' is a DFA and $\mathbf{alphabet}(M') = \Sigma'$, then we return M' as our DFA, N. Otherwise, we must turn M' into a DFA whose alphabet is Σ'. We have that

- $\mathbf{alphabet}(M') \subseteq \Sigma'$; and
- for all $q \in Q_{M'}$ and $a \in \Sigma'$, there is at most one $r \in Q_{M'}$ such that $(q, a, r) \in T_{M'}$.

Since M' is simplified, there are two cases to consider.

If M' has no accepting states, then $s_{M'}$ is the only state of M' and M' has no transitions. Thus we can define our DFA N by:

- $Q_N = Q_{M'} = \{s_{M'}\}$;
- $s_N = s_{M'}$;
- $A_N = A_{M'} = \emptyset$;
- $T_N = \{\, (s_{M'}, a, s_{M'}) \mid a \in \Sigma' \,\}$.

Alternatively, M' has at least one accepting state. Thus, M' has no dead states. We define our DFA N by:

- $Q_N = Q_{M'} \cup \{\langle \mathsf{dead} \rangle\}$ (actually, we put enough brackets around $\langle \mathsf{dead} \rangle$ so that it's not in $Q_{M'}$);
- $s_N = s_{M'}$;
- $A_N = A_{M'}$;

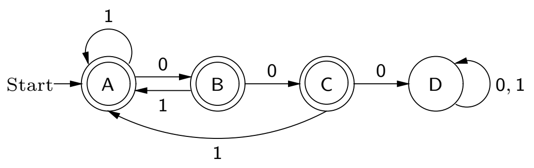
- $T_N = T_{M'} \cup T'$, where T' is the set of all triples $(q, a, \langle\mathsf{dead}\rangle)$ such that either
  - $q \in Q_{M'}$ and $a \in \Sigma'$, but there is no $r \in Q_{M'}$ such that $(q, a, r) \in T_{M'}$; or
  - $q = \langle\mathsf{dead}\rangle$ and $a \in \Sigma'$.

We define a function **determSimplify** $\in$ **DFA** $\times$ **Alp** $\rightarrow$ **DFA** by: **determSimplify**(M, Σ) is the result of running the above algorithm on M and Σ.

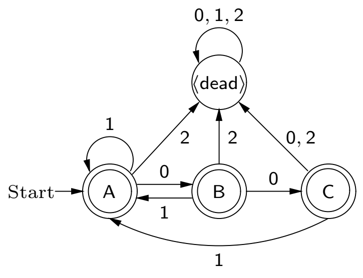
Theorem 3.10.7
*For all $M \in$ **DFA** and $\Sigma \in$ **Alp**:*

- **determSimplify**(M, Σ) *is deterministically simplified;*
- **determSimplify**(M, Σ) *is equivalent to M; and*
- **alphabet**(**determSimplify**(M, Σ)) $=$ **alphabet**$(L(M)) \cup \Sigma$.

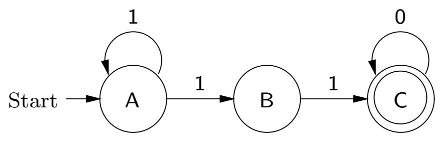
For example, suppose M is the DFA



Then **determSimplify**$(M, \{2\})$ is the DFA



Now we consider the problem of converting NFAs to DFAs. Suppose M is the NFA



How can we convert M into a DFA? Our approach will be to convert M into a DFA N whose states represent the elements of the set

$$\{\, \Delta_M(\{\mathsf{A}\}, w) \mid w \in \{\mathsf{0}, \mathsf{1}\}^* \,\}.$$

For example, one the states of N will be $\langle \mathsf{A}, \mathsf{B} \rangle$, which represents $\{\mathsf{A}, \mathsf{B}\} = \Delta_M(\{\mathsf{A}\}, \mathsf{1})$. This is the state that our DFA will be in after processing $\mathsf{1}$ from the start state.

Before describing our conversion algorithm, we first consider a proposition concerning the Δ function for NFAs and say how we will represent finite sets of symbols as symbols.

Proposition 3.10.8
Suppose M is an NFA.

(1) For all $P \subseteq Q_M$, $\Delta_M(P, \%) = P$.

*(2) For all $P \subseteq Q_M$ and $a \in$ **alphabet**(M), $\Delta_M(P, a) = \{\, r \in Q_M \mid (p, a, r) \in T_M, \text{ for some } p \in P \,\}$.*

*(3) For all $P \subseteq Q_M$ and $x, y \in$ **alphabet**$(M)^*$, $\Delta_M(P, xy) = \Delta_M(\Delta_M(P, x), y)$.*

Given a finite set of symbols P, we write $\overline{P}$ for the symbol

$$\langle a_1, \ldots, a_n \rangle,$$

where $a_1, \ldots, a_n$ are all of the elements of P, in order according to our ordering on **Sym**, and without repetition. For example, $\overline{\{\mathsf{B}, \mathsf{A}\}} = \langle \mathsf{A}, \mathsf{B} \rangle$ and $\overline{\emptyset} = \langle\rangle$. It is easy to see that, if P and R are finite sets of symbols, then $\overline{P} = \overline{R}$ iff $P = R$.

We convert an NFA M into a DFA N as follows. First, we generate the least subset X of $\mathcal{P}(Q_M)$ such that:

- $\{s_M\} \in X$;
- for all $P \in X$ and $a \in$ **alphabet**(M), $\Delta_M(P, a) \in X$.

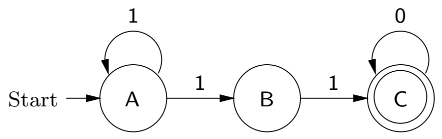
Then we define the DFA N as follows:

- $Q_N = \{\, \overline{P} \mid P \in X \,\}$;
- $s_N = \overline{\{s_M\}} = \langle s_M \rangle$;
- $A_N = \{\, \overline{P} \mid P \in X \text{ and } P \cap A_M \neq \emptyset \,\}$;

- $T_N = \{\, (\overline{P}, a, \overline{\Delta_M(P,a)}) \mid P \in X$ and $a \in \mathbf{alphabet}(M)\,\}$.

Then N is a DFA with alphabet $\mathbf{alphabet}(M)$ and, for all $P \in X$ and $a \in \mathbf{alphabet}(M)$, $\delta_N(\overline{P}, a) = \overline{\Delta_M(P,a)}$.

Now, we show how our example NFA can be converted into a DFA. Suppose M is the NFA



Let's work out what the DFA N is.

- To begin with, $\{\mathsf{A}\} \in X$, so that $\langle\mathsf{A}\rangle \in Q_N$. And $\langle\mathsf{A}\rangle$ is the start state of N. It is not an accepting state, since $\mathsf{A} \not\in A_M$.

- Since $\{\mathsf{A}\} \in X$, and $\Delta(\{\mathsf{A}\},\mathsf{0}) = \emptyset$, we add $\emptyset$ to X, $\langle\rangle$ to Q_N and $(\langle\mathsf{A}\rangle, \mathsf{0}, \langle\rangle)$ to T_N.

  Since $\{\mathsf{A}\} \in X$, and $\Delta(\{\mathsf{A}\},\mathsf{1}) = \{\mathsf{A},\mathsf{B}\}$, we add $\{\mathsf{A},\mathsf{B}\}$ to X, $\langle\mathsf{A},\mathsf{B}\rangle$ to Q_N and $(\langle\mathsf{A}\rangle, \mathsf{1}, \langle\mathsf{A},\mathsf{B}\rangle)$ to T_N.

- Since $\emptyset \in X$, $\Delta(\emptyset,\mathsf{0}) = \emptyset$ and $\emptyset \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\rangle, \mathsf{0}, \langle\rangle)$ to T_N.

  Since $\emptyset \in X$, $\Delta(\emptyset,\mathsf{1}) = \emptyset$ and $\emptyset \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\rangle, \mathsf{1}, \langle\rangle)$ to T_N.

- Since $\{\mathsf{A},\mathsf{B}\} \in X$, $\Delta(\{\mathsf{A},\mathsf{B}\},\mathsf{0}) = \emptyset$ and $\emptyset \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\mathsf{A},\mathsf{B}\rangle, \mathsf{0}, \langle\rangle)$ to T_N.

  Since $\{\mathsf{A},\mathsf{B}\} \in X$, $\Delta(\{\mathsf{A},\mathsf{B}\},\mathsf{1}) = \{\mathsf{A},\mathsf{B}\} \cup \{\mathsf{C}\} = \{\mathsf{A},\mathsf{B},\mathsf{C}\}$, we add $\{\mathsf{A},\mathsf{B},\mathsf{C}\}$ to X, $\langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle$ to Q_N, and $(\langle\mathsf{A},\mathsf{B}\rangle, \mathsf{1}, \langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle)$ to T_N. Since $\{\mathsf{A},\mathsf{B},\mathsf{C}\}$ contains (the only) one of M's accepting states, we add $\langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle$ to A_N.

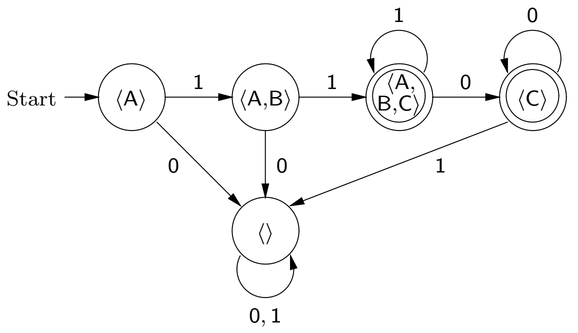
- Since $\{\mathsf{A},\mathsf{B},\mathsf{C}\} \in X$ and $\Delta(\{\mathsf{A},\mathsf{B},\mathsf{C}\},\mathsf{0}) = \emptyset \cup \emptyset \cup \{\mathsf{C}\} = \{\mathsf{C}\}$, we add $\{\mathsf{C}\}$ to X, $\langle\mathsf{C}\rangle$ to Q_N and $(\langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle, \mathsf{0}, \langle\mathsf{C}\rangle)$ to T_N. Since $\{\mathsf{C}\}$ contains one of M's accepting states, we add $\langle\mathsf{C}\rangle$ to A_N.

  Since $\{\mathsf{A},\mathsf{B},\mathsf{C}\} \in X$, $\Delta(\{\mathsf{A},\mathsf{B},\mathsf{C}\},\mathsf{1}) = \{\mathsf{A},\mathsf{B}\} \cup \{\mathsf{C}\} \cup \emptyset = \{\mathsf{A},\mathsf{B},\mathsf{C}\}$ and $\{\mathsf{A},\mathsf{B},\mathsf{C}\} \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle, \mathsf{1}, \langle\mathsf{A},\mathsf{B},\mathsf{C}\rangle)$ to T_N.

- Since $\{\mathsf{C}\} \in X$, $\Delta(\{\mathsf{C}\}, \mathsf{0}) = \{\mathsf{C}\}$ and $\{\mathsf{C}\} \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\mathsf{C}\rangle, \mathsf{0}, \langle\mathsf{C}\rangle)$ to T_N.

  Since $\{\mathsf{C}\} \in X$, $\Delta(\{\mathsf{C}\}, \mathsf{1}) = \emptyset$ and $\emptyset \in X$, we don't have to add anything to X or Q_N, but we add $(\langle\mathsf{C}\rangle, \mathsf{1}, \langle\rangle)$ to T_N.

Since there are no more elements to add to X, we are done. Thus, the DFA N is



The following two lemmas show why our conversion process is correct.

Lemma 3.10.9
For all $w \in \textbf{alphabet}(M)^*$*:*

- $\Delta_M(\{s_M\}, w) \in X$*; and*
- $\delta_N(s_N, w) = \overline{\Delta_M(\{s_M\}, w)}$.

Proof. By left string induction.

(Basis Step) We have that $\Delta_M(\{s_M\}, \%) = \{s_M\} \in X$ and $\delta_N(s_N, \%) = s_N = \overline{\{s_M\}} = \overline{\Delta_M(\{s_M\}, \%)}$.

(Inductive Step) Suppose $a \in \textbf{alphabet}(M)$ and $w \in \textbf{alphabet}(M)^*$. Assume the inductive hypothesis: $\Delta_M(\{s_M\}, w) \in X$ and $\delta_N(s_N, w) = \overline{\Delta_M(\{s_M\}, w)}$.

Since $\Delta_M(\{s_M\}, w) \in X$ and $a \in \textbf{alphabet}(M)$, we have that $\Delta_M(\{s_M\}, wa) = \Delta_M(\Delta_M(\{s_M\}, w), a) \in X$. Thus

$$\begin{aligned}
\delta_N(s_N, wa) &= \delta_N(\delta_N(s_N, w), a) \\
&= \delta_N(\overline{\Delta_M(\{s_M\}, w)}, a) \qquad \text{(inductive hypothesis)} \\
&= \overline{\Delta_M(\Delta_M(\{s_M\}, w), a)} \\
&= \overline{\Delta_M(\{s_M\}, wa)}.
\end{aligned}$$

□

Lemma 3.10.10
$L(N) = L(M)$.

Proof. ($L(M) \subseteq L(N)$) Suppose $w \in L(M)$, so that $w \in \mathbf{alphabet}(M)^* = \mathbf{alphabet}(N)^*$ and $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$. By Lemma 3.10.9, we have that $\Delta_M(\{s_M\}, w) \in X$ and $\delta_N(s_N, w) = \overline{\Delta_M(\{s_M\}, w)}$. Since $\Delta_M(\{s_M\}, w) \in X$ and $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$, it follows that $\delta_N(s_N, w) = \overline{\Delta_M(\{s_M\}, w)} \in A_N$. Thus $w \in L(N)$.

($L(N) \subseteq L(M)$) Suppose $w \in L(N)$, so that $w \in \mathbf{alphabet}(N)^* = \mathbf{alphabet}(M)^*$ and $\delta_N(s_N, w) \in A_N$. By Lemma 3.10.9, we have that $\delta_N(s_N, w) = \overline{\Delta_M(\{s_M\}, w)}$. Thus $\overline{\Delta_M(\{s_M\}, w)} \in A_N$, so that $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$. Thus $w \in L(M)$. $\square$

We define a function **nfaToDFA** $\in$ **NFA** $\to$ **DFA** by: **nfaToDFA**(M) is the result of running the preceding algorithm with input M.

Theorem 3.10.11
For all $M \in$ **NFA**:

- **nfaToDFA**$(M) \approx M$; *and*
- **alphabet**(**nfaToDFA**(M)) $=$ **alphabet**(M).

The Forlan module `DFA` defines an abstract type `dfa` (in the top-level environment) of deterministic finite automata, along with various functions for processing DFAs. Values of type `dfa` are implemented as values of type `fa`, and the module DFA provides the following injection and projection functions:

```
val injToFA     : dfa -> fa
val injToNFA    : dfa -> nfa
val injToEFA    : dfa -> efa
val projFromFA  : fa -> dfa
val projFromNFA : nfa -> dfa
val projFromEFA : efa -> dfa
```

These functions are available in the top-level environment with the names `injDFAToFA`, `injDFAToNFA`, `injDFAToEFA`, `projFAToDFA`, `projNFAToDFA` and `projEFAToDFA`.

The module `DFA` also defines the functions:

```
val input          : string -> dfa
val determProcStr  : dfa -> sym * str -> sym
val determAccepted : dfa -> str -> bool
val determSimplify : dfa * sym set -> dfa
val fromNFA        : nfa -> dfa
```

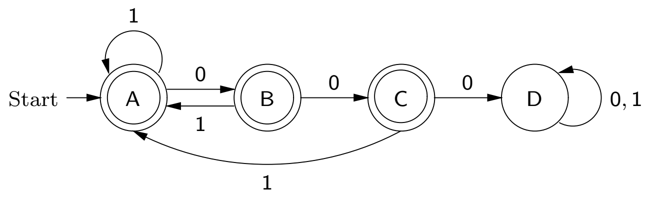
The function `input` is used to input a DFA. The function `determProcStr` is used to compute $\delta_M(q, w)$ for a DFA M, using the definition of δ_M. The function `determAccepted` uses `determProcStr` to check whether a string is accepted by a DFA. The function `determSimplify` corresponds to **determSimplify**. The function `fromNFA` corresponds to our conversion function **nfaToDFA**, and is available in the top-level environment with that name:

```
val nfaToDFA : nfa -> dfa
```

Most of the functions for processing FAs that were introduced in previous sections are inherited by `DFA`:

```
val output                  : string * dfa -> unit
val numStates               : dfa -> int
val numTransitions          : dfa -> int
val alphabet                : dfa -> sym set
val equal                   : dfa * dfa -> bool
val isomorphism             : dfa * dfa * sym_rel -> bool
val findIsomorphism         : dfa * dfa -> sym_rel
val isomorphic              : dfa * dfa -> bool
val renameStates            : dfa * sym_rel -> dfa
val renameStatesCanonically : dfa -> dfa
val processStr              : dfa -> sym set * str -> sym set
val processStrBackwards     : dfa -> sym set * str -> sym set
val accepted                : dfa -> str -> bool
val emptyClose              : dfa -> sym set -> sym set
val emptyCloseBackwards     : dfa -> sym set -> sym set
val checkLP                 : dfa -> lp -> unit
val validLP                 : dfa -> lp -> bool
val findLP                  : dfa -> sym set * str * sym set -> lp
val findAcceptingLP         : dfa -> str -> lp
```

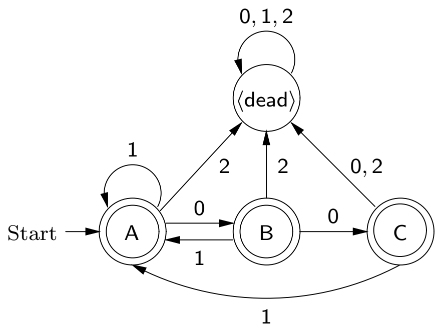
Suppose `dfa` is the `dfa`



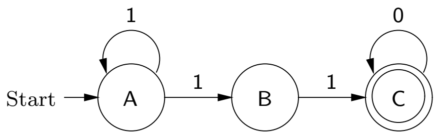
We can turn `dfa` into an equivalent deterministically simplified DFA whose alphabet is the union of the alphabet of the language of `dfa` and $\{2\}$, i.e., whose alphabet is $\{0, 1, 2\}$, as follows:

```
- val dfa' = DFA.determSimplify(dfa, SymSet.input "");
@ 2
@ .
val dfa' = - : dfa
- DFA.output("", dfa');
{states}
A, B, C, <dead>
{start state}
A
{accepting states}
A, B, C
{transitions}
A, 0 -> B; A, 1 -> A; A, 2 -> <dead>; B, 0 -> C;
B, 1 -> A; B, 2 -> <dead>; C, 0 -> <dead>; C, 1 -> A;
C, 2 -> <dead>; <dead>, 0 -> <dead>; <dead>, 1 -> <dead>;
<dead>, 2 -> <dead>
val it = () : unit
```

Thus `dfa'` is



Suppose that `nfa` is the `nfa`

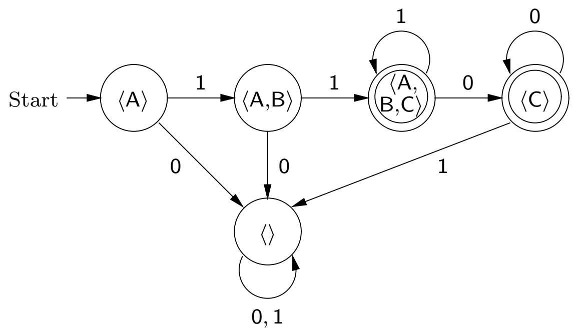


We can convert `nfa` to a DFA as follows:

```
- val dfa = nfaToDFA nfa;
val dfa = - : dfa
- DFA.output("", dfa);
{states}
<>, <A>, <C>, <A,B>, <A,B,C>
{start state}
```

```
<A>
{accepting states}
<C>, <A,B,C>
{transitions}
<>, 0 -> <>; <>, 1 -> <>; <A>, 0 -> <>; <A>, 1 -> <A,B>;
<C>, 0 -> <C>; <C>, 1 -> <>; <A,B>, 0 -> <>;
<A,B>, 1 -> <A,B,C>; <A,B,C>, 0 -> <C>;
<A,B,C>, 1 -> <A,B,C>
val it = () : unit
```

Thus `dfa` is



3.11 Closure Properties of Regular Languages

In this section, we show how to convert regular expressions to finite automata, as well as how to convert finite automata to regular expressions. As a result, we will be able to conclude that the following statements about a language L are equivalent:

- L is regular;
- L is generated by a regular expression;
- L is accepted by a finite automaton;
- L is accepted by an EFA;
- L is accepted by an NFA;
- L is accepted by a DFA.

Also, we will introduce:

- operations on FAs corresponding to union, concatenation and closure;

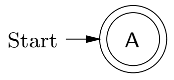
- an operation on EFAs corresponding to intersection;
- an operation on DFAs corresponding to set difference.

As a result, we will have that the set **RegLan** of regular languages is closed under union, concatenation, closure, intersection and set difference. I.e., we will have that, if $L, L_1, L_2 \in \textbf{RegLan}$, then $L_1 \cup L_2$, L_1L_2, L^*, $L_1 \cap L_2$ and $L_1 - L_2$ are in **RegLan**.

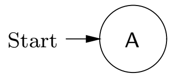
We will also show several additional closure properties of regular languages, in addition to giving the corresponding operations on regular expressions and automata.

In order to give an algorithm for converting regular expressions to finite automata, we must first define several constants and operations on FAs.

We write **emptyStr** for the DFA

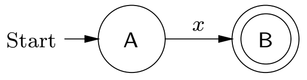


and **emptySet** for the DFA



Thus, we have that $L(\textbf{emptyStr}) = \{\%\}$ and $L(\textbf{emptySet}) = \emptyset$. Of course **emptyStr** and **emptySet** are also NFAs, EFAs and FAs.

Next, we define a function $\textbf{fromStr} \in \textbf{Str} \to \textbf{FA}$ by: $\textbf{fromStr}(x)$ is the FA



Thus, for all $x \in \textbf{Str}$, $L(\textbf{fromStr}(x)) = \{x\}$. It is also convenient to define a function $\textbf{fromSym} \in \textbf{Sym} \to \textbf{NFA}$ by: $\textbf{fromSym}(a) = \textbf{fromStr}(a)$. Of course, **fromSym** is also an element of $\textbf{Sym} \to \textbf{EFA}$ and $\textbf{Sym} \to \textbf{FA}$. Furthermore, for all $a \in \textbf{Sym}$, $L(\textbf{fromSym}(a)) = \{a\}$.

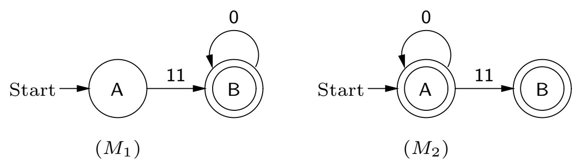
Next, we define a function $\textbf{union} \in \textbf{FA} \times \textbf{FA} \to \textbf{FA}$ such that $L(\textbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$. In other words, a string will be accepted by $\textbf{union}(M_1, M_2)$ iff it is accepted by M_1 or M_2. If $M_1, M_2 \in \textbf{FA}$, then $\textbf{union}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \mathsf{A}$;

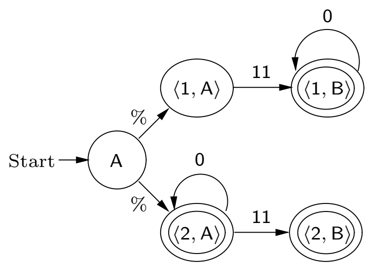
- $A_N = \{\, \langle 1, q\rangle \mid q \in A_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in A_{M_2} \,\}$;
- $T_N =$

$$\begin{aligned} &\{(\mathsf{A}, \%, \langle 1, s_{M_1}\rangle), (\mathsf{A}, \%, \langle 2, s_{M_2}\rangle)\} \\ \cup\, &\{\, (\langle 1, q\rangle, a, \langle 1, r\rangle) \mid (q, a, r) \in T_{M_1} \,\} \\ \cup\, &\{\, (\langle 2, q\rangle, a, \langle 2, r\rangle) \mid (q, a, r) \in T_{M_2} \,\}. \end{aligned}$$

For example, if M_1 and M_2 are the FAs



then **union**(M_1, M_2) is the FA



Proposition 3.11.1
For all $M_1, M_2 \in$ **FA***:*

- $L(\textbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$;
- $\textbf{alphabet}(\textbf{union}(M_1, M_2)) = \textbf{alphabet}(M_1) \cup \textbf{alphabet}(M_2)$.

Proposition 3.11.2
For all $M_1, M_2 \in$ **EFA***,* **union**$(M_1, M_2) \in$ **EFA**.

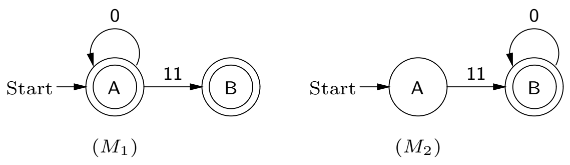
Next, we define a function **concat** $\in$ **FA** $\times$ **FA** $\to$ **FA** such that $L(\textbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in$ **FA**. In other words, a string will be accepted by **concat**(M_1, M_2) iff it can be divided into two parts in such a way that the first part is accepted by M_1 and the second part is accepted by M_2. If $M_1, M_2 \in$ **FA**, then **concat**(M_1, M_2) is the FA N such that:

- $Q_N = \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;

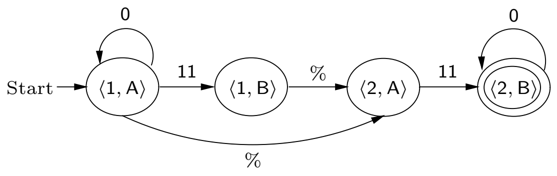
- $s_N = \langle 1, s_{M_1}\rangle$;
- $A_N = \{\, \langle 2, q\rangle \mid q \in A_{M_2}\,\}$;
- $T_N =$

$$\begin{aligned}&\{\, (\langle 1, q\rangle, \%, \langle 2, s_{M_2}\rangle) \mid q \in A_{M_1}\,\}\\ \cup\, &\{\, (\langle 1, q\rangle, a, \langle 1, r\rangle) \mid (q, a, r) \in T_{M_1}\,\}\\ \cup\, &\{\, (\langle 2, q\rangle, a, \langle 2, r\rangle) \mid (q, a, r) \in T_{M_2}\,\}.\end{aligned}$$

For example, if M_1 and M_2 are the FAs



then **concat**(M_1, M_2) is the FA



Proposition 3.11.3
For all $M_1, M_2 \in$ **FA***:*

- $L($**concat**$(M_1, M_2)) = L(M_1)L(M_2)$;
- **alphabet**(**concat**$(M_1, M_2)) =$ **alphabet**$(M_1) \cup$ **alphabet**(M_2).

Proposition 3.11.4
For all $M_1, M_2 \in$ **EFA**, **concat**$(M_1, M_2) \in$ **EFA**.

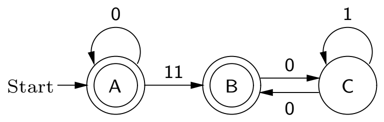
As the last of our operations on FAs, we define a function **closure** $\in$ **FA** $\rightarrow$ **FA** such that $L($**closure**$(M)) = L(M)^*$, for all $M \in$ **FA**. In other words, a string will be accepted by **closure**(M) iff it can be formed by concatenating together some number of strings that are accepted by M. If $M \in$ **FA**, then **closure**(M) is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q\rangle \mid q \in Q_M\,\}$;
- $s_N = \mathsf{A}$;

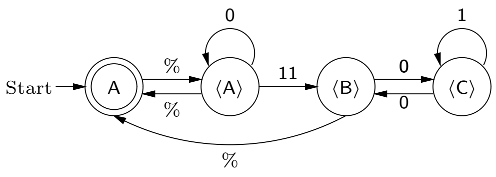
- $A_N = \{\mathsf{A}\}$;
- $T_N =$

$$
\begin{aligned}
&\{(\mathsf{A}, \%, \langle s_M \rangle)\} \\
\cup\, &\{\, (\langle q \rangle, \%, \mathsf{A}) \mid q \in A_M \,\} \\
\cup\, &\{\, (\langle q \rangle, a, \langle r \rangle) \mid (q, a, r) \in T_M \,\}.
\end{aligned}
$$

For example, if M is the FA



then **closure**(M) is the FA



Proposition 3.11.5
*For all $M \in$ **FA**,*

- $L(\textbf{closure}(M)) = L(M)^*$;
- $\textbf{alphabet}(\textbf{closure}(M)) = \textbf{alphabet}(M)$.

Proposition 3.11.6
*For all $M \in$ **EFA**, **closure**(M) $\in$ **EFA**.*

Now, we use our FA constants and operations on FAs in order to give an algorithm for converting regular expressions to FAs. We define a function $\textbf{regToFA} \in \textbf{Reg} \to \textbf{FA}$ by recursion, as follows. The goal is for $L(\textbf{regToFA}(\alpha))$ to be equal to $L(\alpha)$, for all regular expressions α.

(1) $\textbf{regToFA}(\%) = \textbf{emptyStr}$;

(2) $\textbf{regToFA}(\$) = \textbf{emptySet}$;

(3) for all $\alpha \in \textbf{Reg}$, $\textbf{regToFA}(\alpha^*) = \textbf{closure}(\textbf{regToFA}(\alpha))$;

(4) for all $\alpha, \beta \in \textbf{Reg}$, $\textbf{regToFA}(\alpha + \beta) =$
$\textbf{union}(\textbf{regToFA}(\alpha), \textbf{regToFA}(\beta))$;

(5) for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \mathbf{Sym}$, $\mathbf{regToFA}(a_1 \cdots a_n) = \mathbf{fromStr}(a_1 \cdots a_n)$;

(6) for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in \mathbf{Sym}$ and $\alpha \in \mathbf{Reg}$, if α doesn't consist of a single symbol, and α doesn't have the form $b\,\beta$ for some $b \in \mathbf{Sym}$ and $\beta \in \mathbf{Reg}$, then $\mathbf{regToFA}(a_1 \cdots a_n\, \alpha) = \mathbf{concat}(\mathbf{fromStr}(a_1 \cdots a_n), \mathbf{regToFA}(\alpha))$;

(7) for all $\alpha, \beta \in \mathbf{Reg}$, if α doesn't consist of a single symbol, then $\mathbf{regToFA}(\alpha\beta) = \mathbf{concat}(\mathbf{regToFA}(\alpha), \mathbf{regToFA}(\beta))$.

For example, by Rule (6), we have that $\mathbf{regToFA}(0101^*) = \mathbf{concat}(\mathbf{fromStr}(010), \mathbf{regToFA}(1^*))$. Rule (5), when $n = 1$, handles regular expressions consisting of single symbols. Rule (5), when $n \geq 2$, plus Rules (6)–(7) handle all concatenations. Of course, it would be possible to replace Rule (5) by a rule handling single symbols only, delete Rule (6), and remove the pre-condition on Rule (7). But this would have been at the cost of never introducing transitions with labels consists of multiple symbols.

Theorem 3.11.7
For all $\alpha \in \mathbf{Reg}$:

- $L(\mathbf{regToFA}(\alpha)) = L(\alpha)$;
- $\mathbf{alphabet}(\mathbf{regToFA}(\alpha)) = \mathbf{alphabet}(\alpha)$.

Proof. Because of the form of recursion used, the proof uses induction on the height of α. $\square$

For example, $\mathbf{regToFA}(0^*11 + 001^*)$ is isomorphic to the FA in Figure 3.1.

The Forlan module `FA` includes these constants and functions for building finite automata and converting regular expressions to finite automata:

```
val emptyStr : fa
val emptySet : fa
val fromStr  : str -> fa
val fromSym  : sym -> fa
val union    : fa * fa -> fa
val concat   : fa * fa -> fa
val closure  : fa -> fa
val fromReg  : reg -> fa
```

The function `fromReg` corresponds to **regToFA** and is available in the top-level environment with that name:

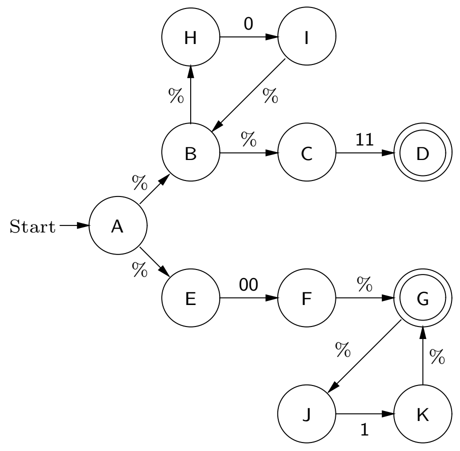


Figure 3.1: Regular Expression to FA Conversion Example

```
val regToFA : reg -> fa
```

The constants `emptyStr` and `emptySet` are inherited by the modules `DFA`, `NFA` and `EFA`. The function `fromSym` is inherited by the modules `NFA` and `EFA`. The functions `union`, `concat` and `closure` are inherited by the module `EFA`.

Here is how the regular expression $0^*11 + 001^*$ can be converted to an FA in Forlan:

```
- val reg = Reg.input "";
@ 0*11 + 001*
@ .
val reg = - : reg
- val fa = regToFA reg;
val fa = - : fa
- FA.output("", fa);
{states}
A, <1,<1,A>>, <1,<2,A>>, <1,<2,B>>, <2,<1,A>>, <2,<1,B>>,
<2,<2,A>>, <1,<1,<A>>>, <1,<1,<B>>>, <2,<2,<A>>>,
<2,<2,<B>>>
{start state}
A
{accepting states}
<1,<2,B>>, <2,<2,A>>
{transitions}
A, % -> <1,<1,A>> | <2,<1,A>>;
```

```
<1,<1,A>>, % -> <1,<2,A>> | <1,<1,<A>>>;
<1,<2,A>>, 11 -> <1,<2,B>>; <2,<1,A>>, 00 -> <2,<1,B>>;
<2,<1,B>>, % -> <2,<2,A>>; <2,<2,A>>, % -> <2,<2,<A>>>;
<1,<1,<A>>>, 0 -> <1,<1,<B>>>;
<1,<1,<B>>>, % -> <1,<1,A>>;
<2,<2,<A>>>, 1 -> <2,<2,<B>>>; <2,<2,<B>>>, % -> <2,<2,A>>
val it = () : unit
- val fa' = FA.renameStatesCanonically fa;
val fa' = - : fa
- FA.output("", fa');
{states}
A, B, C, D, E, F, G, H, I, J, K
{start state}
A
{accepting states}
D, G
{transitions}
A, % -> B | E; B, % -> C | H; C, 11 -> D; E, 00 -> F;
F, % -> G; G, % -> J; H, 0 -> I; I, % -> B; J, 1 -> K;
K, % -> G
val it = () : unit
```

Thus `fa'` is the finite automaton in Figure 3.1.

We will now work towards the description of an algorithm for converting FAs to Regular Expressions. To see how the algorithm will work, we need to study a set of languages that are derived from finite automata, which we refer to as "between languages".

We begin by defining two auxiliary functions. Suppose M is an FA. We define a function $\mathbf{ord}_M \in Q_M \to \{1, \ldots, |Q_M|\}$ by: $\mathbf{ord}_M(q) = |\{ r \in Q_M \mid r \leq q \}|$. Here $\leq$ is our standard ordering on symbols. We refer to $\mathbf{ord}_M(q)$ as the *ordinal number of q in M*. For example, if $Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, then $\mathbf{ord}_M(\mathsf{A}) = 1$, $\mathbf{ord}_M(\mathsf{B}) = 2$ and $\mathbf{ord}_M(\mathsf{C}) = 3$. Clearly, $\mathbf{ord}_M$ is a bijection from Q_M to $\{1, \ldots, |Q_M|\}$. Thus we can define a function $\mathbf{state}_M \in \{1, \ldots, |Q_M|\} \to Q_M$ by: $\mathbf{state}_M(n) =$ the unique q such that $\mathbf{ord}_M(q) = n$. For example, if $Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, then $\mathbf{state}_M(2) = \mathsf{B}$. We often abbreviate $\mathbf{ord}_M$ and $\mathbf{state}_M$ to $\mathbf{ord}$ and $\mathbf{state}$, respectively.

Suppose M is an FA. We define a function

$$\mathbf{Btw}_M \in \{1, \ldots, |Q_M|\} \times \{1, \ldots, |Q_M|\} \times \{0, \ldots, |Q_M|\} \to \mathbf{Lan}$$

by: $\mathbf{Btw}_M(i, j, k)$ is the set of all $w \in \mathbf{Str}$ such that there is a labeled path

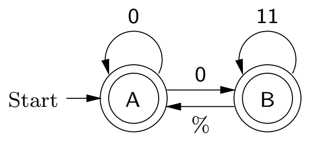
$$lp = q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_{n-1} \overset{x_{n-1}}{\Rightarrow} q_n,$$

such that

- lp is valid for M;
- $w = x_1 x_2 \cdots x_{n-1} = \mathbf{label}(lp)$;
- $\mathbf{ord}(q_1) = i$;
- $\mathbf{ord}(q_n) = j$;
- for all $1 < l < n$, $\mathbf{ord}(q_l) \leq k$.

In other words, $\mathbf{Btw}_M(i, j, k)$ consists of the labels of all of the valid labeled paths for M that take us *between* the i'th state and the j'th state without going through any *intermediate* states whose ordinal numbers are greater than k. Here, intermediate doesn't include the labeled path's start or end states. We think of the $\mathbf{Btw}_M(i, j, k)$ as "between languages". We often abbreviate $\mathbf{Btw}_M$ to $\mathbf{Btw}$.

Suppose M is the finite automaton



- $\mathsf{0} \in \mathbf{Btw}(1, 2, 0)$, because of the labeled path

  $$\mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B}$$

  (which has no intermediate states).

- $\% \in \mathbf{Btw}(2, 2, 0)$, because of the labeled path

  $$\mathsf{B}$$

  (which has no intermediate states).

- $\mathsf{00} \in \mathbf{Btw}(2, 2, 1)$ because of the labeled path

  $$\mathsf{B} \overset{\%}{\Rightarrow} \mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{A} \overset{\mathsf{0}}{\Rightarrow} \mathsf{B},$$

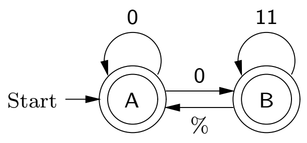
  (both of the intermediate states are A, and A's ordinal number is 1, which is less-than-or-equal-to 1).

- $\mathsf{1111} \not\in \mathbf{Btw}(2,2,1)$ because the only valid labeled path between B and B whose label is $\mathsf{1111}$ is

$$\mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B} \overset{\mathsf{11}}{\Rightarrow} \mathsf{B},$$

  and this labeled path has an intermediate state whose ordinal number, 2, is greater-than 1.

Consider our example FA M again:



What is another way of describing the language $\mathbf{Btw}_M(1,1,2) \cup \mathbf{Btw}_M(1,2,2)$? Since the first argument of each call to $\mathbf{Btw}$ is the ordinal number of M's start state, the second arguments consist of the ordinal numbers of M's accepting states, and the third argument of each call is $2 = |Q|$, the answer is $L(M)$. Thus, if we can figure out how to translate the between languages of a finite automaton to regular expressions, we will be able to translate the FA to a regular expression. The key to translating between languages to regular expressions is the following lemma, which shows how to express $\mathbf{Btw}(i,j,k)$ recursively.

Lemma 3.11.8
Suppose M is an FA.

- *For all $1 \leq i, j \leq |Q|$, $\mathbf{Btw}_M(i,j,0) =$*

$$\{\, w \in \mathbf{Str} \mid (\mathbf{state}(i), w, \mathbf{state}(j)) \in T_M \,\} \cup \{\, \% \mid i = j \,\}.$$

- *For all $1 \leq i, j \leq |Q|$ and $0 \leq k < |Q|$, $\mathbf{Btw}_M(i,j,k+1) =$*

$$\begin{aligned} &\mathbf{Btw}_M(i,j,k) \\ \cup\, &\mathbf{Btw}_M(i,k+1,k)\, \mathbf{Btw}_M(k+1,k+1,k)^*\, \mathbf{Btw}_M(k+1,j,k). \end{aligned}$$

Now we are ready to give our algorithm for converting FAs to regular expressions. We define a function

$$\mathbf{faToReg} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{FA} \to \mathbf{Reg}.$$

This function takes in a function *simp* (like the functions **weakSimplify** or **simplify**(**weakSubset**) defined in Section 3.2) for simplifying regular expressions, and returns a function that uses *simp* in order to turn an FA M into a regular expression.

Suppose $simp \in \mathbf{Reg} \to \mathbf{Reg}$ and M is an FA. We must say what the regular expression $\mathbf{faToReg}(simp)(M)$ is. First, we define a function

$$\mathbf{btw} \in \{1, \ldots, |Q_M|\} \times \{1, \ldots, |Q_M|\} \times \{0, \ldots, |Q_M|\} \to \mathbf{Reg}$$

by recursion on its third argument.

- For all $1 \leq i, j \leq |Q_M|$, $\mathbf{btw}(i, j, 0)$ is formed by turning each element of $\mathbf{Btw}(i, j, 0)$ into a regular expression in the obvious way, putting these regular expressions in order and summing them together (yielding \$ if there aren't any of them), and applying *simp* to this regular expression.

- For all $1 \leq i, j \leq |Q_M|$ and $0 \leq k < |Q_M|$, $\mathbf{btw}(i, j, k+1)$ is the result of applying *simp* to

$$\mathbf{btw}(i, j, k) + \mathbf{btw}(i, k+1, k)\, \mathbf{btw}(k+1, k+1, k)^*\, \mathbf{btw}(k+1, j, k)).$$

Actually, we use memoization to avoid computing the result of a given recursive call more than once.

Lemma 3.11.9
Suppose that $simp(\alpha) \approx \alpha$*, for all* $\alpha \in \mathbf{Reg}$*. For all* $1 \leq i, j \leq |Q_M|$ *and* $0 \leq k \leq |Q_M|$*,* $L(\mathbf{btw}(i, j, k)) = \mathbf{Btw}(i, j, k)$*.*

Proof. By mathematical induction on k. $\square$

Let $q_1, \ldots, q_n$ be the accepting states of M. Then $\mathbf{faToReg}(simp)(M)$ is the result of applying *simp* to

$$\mathbf{btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_1), |Q_M|) + \cdots + \mathbf{btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_n), |Q_M|).$$

(Actually, the $\mathbf{btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_i), |Q_M|)$'s are put in order before being summed and then simplified.)

Thus, assuming that $simp(\alpha) \approx \alpha$, for all $\alpha \in \mathbf{Reg}$, we will have that

$$\begin{aligned}
L(\mathbf{faToReg}(simp)(M)) &= L(\mathbf{btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_1), |Q_M|)) \cup \cdots \cup \\
&\; L(\mathbf{btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_n), |Q_M|)) \\
&= \mathbf{Btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_1), |Q_M|) \cup \cdots \cup \\
&\; \mathbf{Btw}(\mathbf{ord}(s_M), \mathbf{ord}(q_n), |Q_M|) \\
&= L(M).
\end{aligned}$$

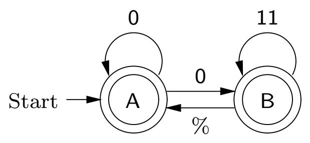
Theorem 3.11.10
For all $\mathit{simp} \in \mathbf{Reg} \to \mathbf{Reg}$ *such that,*

> *for all* $\alpha \in \mathbf{Reg}$, $\mathit{simp}(\alpha) \approx \alpha$ *and* $\mathbf{alphabet}(\mathit{simp}(\alpha)) \subseteq \mathbf{alphabet}(\alpha)$,

and $M \in \mathbf{FA}$*:*

- $L(\mathbf{faToReg}(\mathit{simp})(M)) = L(M)$*;*
- $\mathbf{faToReg}(\mathit{simp})(M) = \mathit{simp}(\alpha)$, *for some* $\alpha \in \mathbf{Reg}$*;*
- $\mathbf{alphabet}(\mathbf{faToReg}(\mathit{simp})(M)) \subseteq \mathbf{alphabet}(M)$*;*
- *if* M *is simplified, then* $\mathbf{alphabet}(\mathbf{faToReg}(\mathit{simp})(M)) = \mathbf{alphabet}(M)$.

Now, let's work through an FA-to-regular expression conversion example. Suppose $\mathit{simp} \in \mathbf{Reg} \to \mathbf{Reg}$ is $\mathbf{simplify}(\mathbf{weakSubset})$ and M is our example FA:



Since both A and B are accepting states, $\mathbf{faToReg}(\mathit{simp})(M)$ will be

$$\mathit{simp}(\mathbf{btw}(1,1,2) + \mathbf{btw}(1,2,2)).$$

(Actually, $\mathbf{btw}(1,1,2)$ and $\mathbf{btw}(1,2,2)$ should be put in order, before being summed and then simplified.) Thus we must work out the values of $\mathbf{btw}(1,1,2)$ and $\mathbf{btw}(1,2,2)$.

We begin by working top-down:

$$\begin{aligned}
\mathbf{btw}(1,1,2) &= \mathit{simp}(\mathbf{btw}(1,1,1) + \mathbf{btw}(1,2,1)\,\mathbf{btw}(2,2,1)^*\,\mathbf{btw}(2,1,1)),\\
\mathbf{btw}(1,2,2) &= \mathit{simp}(\mathbf{btw}(1,2,1) + \mathbf{btw}(1,2,1)\,\mathbf{btw}(2,2,1)^*\,\mathbf{btw}(2,2,1)),\\
\mathbf{btw}(1,1,1) &= \mathit{simp}(\mathbf{btw}(1,1,0) + \mathbf{btw}(1,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,1,0)),\\
\mathbf{btw}(1,2,1) &= \mathit{simp}(\mathbf{btw}(1,2,0) + \mathbf{btw}(1,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,2,0)),\\
\mathbf{btw}(2,1,1) &= \mathit{simp}(\mathbf{btw}(2,1,0) + \mathbf{btw}(2,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,1,0)),\\
\mathbf{btw}(2,2,1) &= \mathit{simp}(\mathbf{btw}(2,2,0) + \mathbf{btw}(2,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,2,0)).
\end{aligned}$$

Next, we need to work out the values of $\mathbf{btw}(1,1,0)$, $\mathbf{btw}(1,2,0)$, $\mathbf{btw}(2,1,0)$ and $\mathbf{btw}(2,2,0)$.

- Since $\mathbf{Btw}(1,1,0) = \{\%, \mathsf{0}\}$, we have that $\mathbf{btw}(1,1,0) = \% + \mathsf{0}$.
- Since $\mathbf{Btw}(1,2,0) = \{\mathsf{0}\}$, we have that $\mathbf{btw}(1,2,0) = \mathsf{0}$.
- Since $\mathbf{Btw}(2,1,0) = \{\%\}$, we have that $\mathbf{btw}(2,1,0) = \%$.
- Since $\mathbf{Btw}(2,2,0) = \{\%, \mathsf{11}\}$, we have that $\mathbf{btw}(2,2,0) = \% + \mathsf{11}$.

Thus, working bottom-up, and using Forlan to do the simplification, we have:

$$\begin{aligned}
\mathbf{btw}(1,1,1) &= \mathit{simp}(\mathbf{btw}(1,1,0) + \mathbf{btw}(1,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,1,0)) \\
&= \mathit{simp}((\% + \mathsf{0}) + (\% + \mathsf{0})(\% + \mathsf{0})^*(\% + \mathsf{0})) \\
&= \mathsf{0}^*, \\
\mathbf{btw}(1,2,1) &= \mathit{simp}(\mathbf{btw}(1,2,0) + \mathbf{btw}(1,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,2,0)) \\
&= \mathit{simp}(\mathsf{0} + (\% + \mathsf{0})(\% + \mathsf{0})^*\mathsf{0}) \\
&= \mathsf{00}^*, \\
\mathbf{btw}(2,1,1) &= \mathit{simp}(\mathbf{btw}(2,1,0) + \mathbf{btw}(2,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,1,0)) \\
&= \mathit{simp}(\% + \%(\% + \mathsf{0})^*(\% + \mathsf{0})) \\
&= \mathsf{0}^*, \\
\mathbf{btw}(2,2,1) &= \mathit{simp}(\mathbf{btw}(2,2,0) + \mathbf{btw}(2,1,0)\,\mathbf{btw}(1,1,0)^*\,\mathbf{btw}(1,2,0)) \\
&= \mathit{simp}((\% + \mathsf{11}) + \%(\% + \mathsf{0})^*\mathsf{0}) \\
&= \mathsf{0}^* + \mathsf{11}.
\end{aligned}$$

Continuing further, we have:

$$\begin{aligned}
\mathbf{btw}(1,1,2) &= \mathit{simp}(\mathbf{btw}(1,1,1) + \mathbf{btw}(1,2,1)\,\mathbf{btw}(2,2,1)^*\,\mathbf{btw}(2,1,1)) \\
&= \mathit{simp}(\mathsf{0}^* + (\mathsf{00}^*)(\mathsf{0}^* + \mathsf{11})^*\mathsf{0}^*) \\
&= \% + \mathsf{0}(\mathsf{0} + \mathsf{11})^*, \\
\mathbf{btw}(1,2,2) &= \mathit{simp}(\mathbf{btw}(1,2,1) + \mathbf{btw}(1,2,1)\,\mathbf{btw}(2,2,1)^*\,\mathbf{btw}(2,2,1)) \\
&= \mathit{simp}(\mathsf{00}^* + (\mathsf{00}^*)(\mathsf{0}^* + \mathsf{11})^*(\mathsf{0}^* + \mathsf{11})) \\
&= \mathsf{0}(\mathsf{0} + \mathsf{11})^*.
\end{aligned}$$

Finally, (since $\mathbf{btw}(1,1,2)$ is greater-than $\mathbf{btw}(1,2,2)$) we have that:

$$\begin{aligned}
\mathbf{faToReg}(\mathit{simp})(M) &= \mathit{simp}(\mathbf{btw}(1,2,2) + \mathbf{btw}(1,1,2)) \\
&= \mathit{simp}(\mathsf{0}(\mathsf{0} + \mathsf{11})^* + (\% + \mathsf{0}(\mathsf{0} + \mathsf{11})^*)) \\
&= \% + \mathsf{0}(\mathsf{0} + \mathsf{11})^*.
\end{aligned}$$

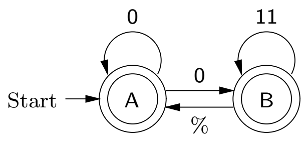
The Forlan module `FA` contains the function

```
val toReg : (reg -> reg) -> fa -> reg
```

which corresponds to our function **faToReg** and is available in the top-level environment with that name:

```
val faToReg : (reg -> reg) -> fa -> reg
```

The modules `DFA`, `NFA` and `EFA` inherit the `toReg` function from `FA`, and these functions are available in the top-level environment with the names `dfaToReg`, `nfaToReg` and `efaToReg`. Suppose `fa` is bound to our example FA



We can convert `fa` into a regular expression as follows:

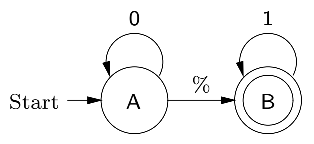
```
- val reg = faToReg (Reg.simplify Reg.weakSubset) fa;
val reg = - : reg
- Reg.output("", reg);
% + 0(0 + 11)*
val it = () : unit
```

Since we have algorithms for converting back and forth between regular expressions and finite automata, as well as algorithms for converting FAs to EFAs, EFAs to NFAs, and NFAs to DFAs, we have the following theorem:

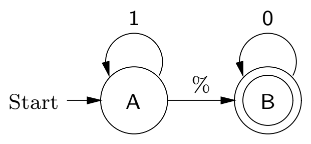
Theorem 3.11.11
Suppose L is a language. The following statements are equivalent:

- *L is regular;*
- *L is generated by a regular expression;*
- *L is accepted by a finite automaton;*
- *L is accepted by an EFA;*
- *L is accepted by an NFA;*
- *L is accepted by a DFA.*

Now we consider an intersection operation on EFAs. Consider the EFAs M_1 and M_2:



(M_1)



(M_2)

How can we construct an EFA N such that $L(N) = L(M_1) \cap L(M_2)$, i.e., so that a string is accepted by N iff it is accepted by both M_1 and M_2? The idea is to make the states of N represent certain pairs of the form (q, r), where $q \in Q_{M_1}$ and $r \in Q_{M_2}$. Since $L(M_1) = \{\mathsf{0}\}^*\{\mathsf{1}\}^*$ and $L(M_2) = \{\mathsf{1}\}^*\{\mathsf{0}\}^*$, we will have that $L(N) = \{\mathsf{0}\}^*\{\mathsf{1}\}^* \cap \{\mathsf{1}\}^*\{\mathsf{0}\}^* = \{\mathsf{0}\}^* \cup \{\mathsf{1}\}^*$.

In order to define our intersection operation on EFAs, we first need to define two auxiliary functions. Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextSym}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \times \mathbf{Sym} \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextSym}_{M_1,M_2}((q, r), a) =$

$$\{\, (q', r') \mid (q, a, q') \in T_{M_1} \text{ and } (r, a, r') \in T_{M_2} \,\}.$$

We often abbreviate $\mathbf{nextSym}_{M_1,M_2}$ to **nextSym**. If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextSym}((\mathsf{A}, \mathsf{A}), \mathsf{0}) = \emptyset$, since there is no $\mathsf{0}$-transition in M_2 from A;
- $\mathbf{nextSym}((\mathsf{A}, \mathsf{B}), \mathsf{0}) = \{(\mathsf{A}, \mathsf{B})\}$, since the only $\mathsf{0}$-transition from A in M_1 leads to A, and the only $\mathsf{0}$-transition from B in M_2 leads to B.

Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid (q, \%, q') \in T_{M_1} \,\} \cup \{\, (q, r') \mid (r, \%, r') \in T_{M_2} \,\}.$$

We often abbreviate $\mathbf{nextEmp}_{M_1,M_2}$ to **nextEmp**. If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$ (we either do a %-transition on the left, leaving the right-side unchanged, or leave the left-side unchanged, and do a %-transition on the right);

- $\textbf{nextEmp}(\mathsf{A},\mathsf{B}) = \{(\mathsf{B},\mathsf{B})\}$;
- $\textbf{nextEmp}(\mathsf{B},\mathsf{A}) = \{(\mathsf{B},\mathsf{B})\}$;
- $\textbf{nextEmp}(\mathsf{B},\mathsf{B}) = \emptyset$.

Now, we define a function $\textbf{inter} \in \textbf{EFA} \times \textbf{EFA} \to \textbf{EFA}$ such that $L(\textbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$, for all $M_1, M_2 \in \textbf{EFA}$. Given EFAs M_1 and M_2, $\textbf{inter}(M_1, M_2)$ is the EFA N that is constructed as follows.

First, we let $\Sigma = \textbf{alphabet}(M_1) \cap \textbf{alphabet}(M_2)$. Next, we generate the least subset X of $Q_{M_1} \times Q_{M_2}$ such that

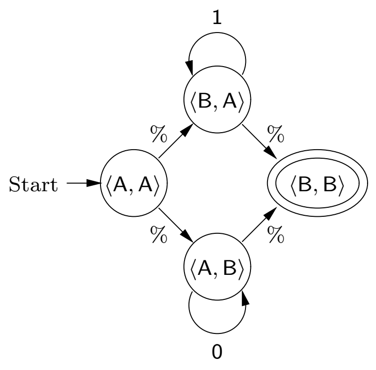
- $(s_{M_1}, s_{M_2}) \in X$;
- for all $q \in Q_{M_1}$, $r \in Q_{M_2}$ and $a \in \Sigma$, if $(q, r) \in X$, then $\textbf{nextSym}((q, r), a) \subseteq X$;
- for all $q \in Q_{M_1}$ and $r \in Q_{M_2}$, if $(q, r) \in X$, then $\textbf{nextEmp}(q, r) \subseteq X$.

Then, the EFA N is defined by:

- $Q_N = \{\, \langle q, r\rangle \mid (q, r) \in X \,\}$;
- $s_N = \langle s_{M_1}, s_{M_2}\rangle$;
- $A_N = \{\, \langle q, r\rangle \mid (q, r) \in X \text{ and } q \in A_{M_1} \text{ and } r \in A_{M_2} \,\}$;
- $T_N =$

$$\begin{aligned}
&\{\, (\langle q, r\rangle, a, \langle q', r'\rangle) \mid (q, r) \in X \text{ and } a \in \Sigma \text{ and} \\
&\qquad\qquad (q', r') \in \textbf{nextSym}((q, r), a) \,\} \\
\cup\, &\{\, (\langle q, r\rangle, \%, \langle q', r'\rangle) \mid (q, r) \in X \text{ and} \\
&\qquad\qquad (q', r') \in \textbf{nextEmp}(q, r) \,\}.
\end{aligned}$$

Suppose M_1 and M_2 are our example EFAs. Then $\textbf{inter}(M_1, M_2)$ is



Theorem 3.11.12
For all $M_1, M_2 \in \mathbf{EFA}$*:*

- $L(\mathbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$*; and*
- $\mathbf{alphabet}(\mathbf{inter}(M_1, M_2)) \subseteq \mathbf{alphabet}(M_1) \cap \mathbf{alphabet}(M_2)$.

Proposition 3.11.13
For all $M_1, M_2 \in \mathbf{NFA}$, $\mathbf{inter}(M_1, M_2) \in \mathbf{NFA}$.

Proposition 3.11.14
For all $M_1, M_2 \in \mathbf{DFA}$, $\mathbf{inter}(M_1, M_2) \in \mathbf{DFA}$.

Next, we consider a complementation operation on DFAs. We define a function **complement** $\in \mathbf{DFA} \times \mathbf{Alp} \to \mathbf{DFA}$ such that, for all $M \in \mathbf{DFA}$ and $\Sigma \in \mathbf{Alp}$,

$$L(\mathbf{complement}(M, \Sigma)) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M).$$

In other words, a string will be accepted by $\mathbf{complement}(M, \Sigma)$ iff its symbols all come from the alphabet $\mathbf{alphabet}(L(M)) \cup \Sigma$ and w is not accepted by M. In the common case when $L(M) \subseteq \Sigma^*$, we will have that $\mathbf{alphabet}(L(M)) \subseteq \Sigma$, and thus that $(\mathbf{alphabet}(L(M)) \cup \Sigma)^* = \Sigma^*$. Hence, it will be the case that

$$L(\mathbf{complement}(M, \Sigma)) = \Sigma^* - L(M).$$

Given a DFA M and an alphabet Σ, $\mathbf{complement}(M, \Sigma)$ is the DFA N that is produced as follows. First, we let the DFA $M' = \mathbf{determSimplify}(M, \Sigma)$. Thus:

- M' is equivalent to M;
- $\mathbf{alphabet}(M') = \mathbf{alphabet}(L(M)) \cup \Sigma$.

Then, we define N by:

- $Q_N = Q_{M'}$;
- $s_N = s_{M'}$;
- $A_N = Q_{M'} - A_{M'}$;
- $T_N = T_{M'}$.

In other words, N is equal to M', except that its accepting states are the non-accepting states of M'.

Then, for all $w \in \mathbf{alphabet}(M')^* = \mathbf{alphabet}(N)^* = (\mathbf{alphabet}(L(M)) \cup \Sigma)^*$,

$$\begin{aligned} w \in L(N) \quad &\text{iff} \quad \delta_N(s_N, w) \in A_N \\ &\text{iff} \quad \delta_N(s_N, w) \in Q_{M'} - A_{M'} \\ &\text{iff} \quad \delta_{M'}(s_{M'}, w) \not\in A_{M'} \\ &\text{iff} \quad w \not\in L(M') \\ &\text{iff} \quad w \not\in L(M). \end{aligned}$$

Now, we can check that $L(N) \subseteq (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M) \subseteq L(N)$, so that $L(N) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$.

Suppose $w \in L(N)$. Then $w \in \mathbf{alphabet}(N)^* = (\mathbf{alphabet}(L(M)) \cup \Sigma)^*$, so that, by the above fact, $w \not\in L(M)$. Thus $w \in (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$.

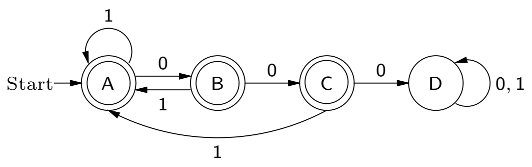
Suppose $w \in (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$. Thus $w \in (\mathbf{alphabet}(L(M)) \cup \Sigma)^*$ and $w \not\in L(M)$. Hence, by the above fact, we have that $w \in L(N)$.

Thus, we have that:

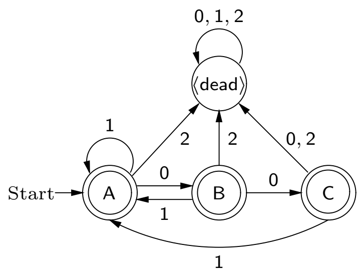
Theorem 3.11.15
For all $M \in \mathbf{DFA}$ and $\Sigma \in \mathbf{Alp}$:

- $L(\mathbf{complement}(M, \Sigma)) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$;
- $\mathbf{alphabet}(\mathbf{complement}(M, \Sigma)) = \mathbf{alphabet}(L(M)) \cup \Sigma$.

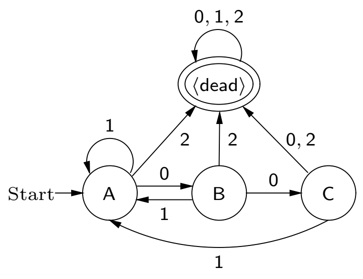
For example, suppose the DFA M is



Then $\mathbf{determSimplify}(M, \{2\})$ is the DFA



Let the DFA $N = \mathbf{complement}(M, \{2\})$. Thus N is



Let $X = \{\, w \in \{0, 1\}^* \mid 000 \text{ is not a substring of } w \,\}$. By Lemma 3.10.6, we have that $L(M) = X$. Thus, by Theorem 3.11.15,

$$\begin{aligned}
L(N) &= L(\mathbf{complement}(M, \{2\})) \\
&= (\mathbf{alphabet}(L(M)) \cup \{2\})^* - L(M) \\
&= (\{0, 1\} \cup \{2\})^* - X \\
&= \{\, w \in \{0, 1, 2\}^* \mid w \not\in X \,\} \\
&= \{\, w \in \{0, 1, 2\}^* \mid 2 \in \mathbf{alphabet}(w) \text{ or } 000 \text{ is a substring of } w \,\}.
\end{aligned}$$

Next, we consider a set difference operation on DFAs. We define a function $\mathbf{minus} \in \mathbf{DFA} \times \mathbf{DFA} \to \mathbf{DFA}$ by:

$$\mathbf{minus}(M_1, M_2) = \mathbf{inter}(M_1, \mathbf{complement}(M_2, \mathbf{alphabet}(M_1))).$$

Theorem 3.11.16
For all $M_1, M_2 \in \mathbf{DFA}$, $L(\mathbf{minus}(M_1, M_2)) = L(M_1) - L(M_2)$.

In other words, a string is accepted by $\mathbf{minus}(M_1, M_2)$ iff it is accepted by M_1 but is not accepted by M_2.

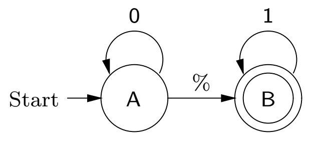
Proof. Suppose $w \in \mathbf{Str}$. Then

$$\begin{aligned}
&w \in L(\mathbf{minus}(M_1, M_2)) \\
\text{iff}\quad &w \in L(\mathbf{inter}(M_1, \mathbf{complement}(M_2, \mathbf{alphabet}(M_1)))) \\
\text{iff}\quad &w \in L(M_1) \text{ and } w \in L(\mathbf{complement}(M_2, \mathbf{alphabet}(M_1))) \\
\text{iff}\quad &w \in L(M_1) \text{ and } w \in (\mathbf{alphabet}(L(M_2)) \cup \mathbf{alphabet}(M_1))^* \text{ and} \\
&w \notin L(M_2) \\
\text{iff}\quad &w \in L(M_1) \text{ and } w \notin L(M_2) \\
\text{iff}\quad &w \in L(M_1) - L(M_2).
\end{aligned}$$

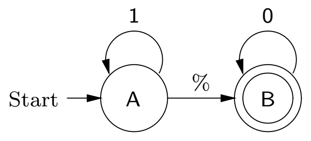
□

To see why the second argument to **complement** is $\mathbf{alphabet}(M_1)$, in the definition of $\mathbf{minus}(M_1, M_2)$, look at the "if" direction of the second-to-last step of the preceding proof: since $w \in L(M_1)$, we have that $w \in \mathbf{alphabet}(M_1)^*$, so that $w \in (\mathbf{alphabet}(L(M_2)) \cup \mathbf{alphabet}(M_1))^*$.

For example, let M_1 and M_2 be the EFAs



(M_1)



(M_2)

Since $L(M_1) = \{0\}^*\{1\}^*$ and $L(M_2) = \{1\}^*\{0\}^*$, we have that

$$L(M_1) - L(M_2) = \{0\}^*\{1\}^* - \{1\}^*\{0\}^* = \{0\}\{0\}^*\{1\}\{1\}^*.$$

Define the DFAs N_1 and N_2 by:

$$\begin{aligned}
N_1 &= \mathbf{nfaToDFA}(\mathbf{efaToNFA}(M_1)), \\
N_2 &= \mathbf{nfaToDFA}(\mathbf{efaToNFA}(M_2)).
\end{aligned}$$

Thus we have that

$$\begin{aligned}
L(N_1) &= L(\mathbf{nfaToDFA}(\mathbf{efaToNFA}(M_1))) \\
&= L(\mathbf{efaToNFA}(M_1)) && \text{(Theorem 3.10.11)} \\
&= L(M_1) && \text{(Theorem 3.9.8)} \\
L(N_2) &= L(\mathbf{nfaToDFA}(\mathbf{efaToNFA}(M_2))) \\
&= L(\mathbf{efaToNFA}(M_2)) && \text{(Theorem 3.10.11)} \\
&= L(M_2) && \text{(Theorem 3.9.8).}
\end{aligned}$$

Let the DFA $N = \mathbf{minus}(N_1, N_2)$. Then

$$\begin{aligned} L(N) &= L(\mathbf{minus}(N_1, N_2)) \\ &= L(N_1) - L(N_2) && \text{(Theorem 3.11.16)} \\ &= L(M_1) - L(M_2) \\ &= \{\mathsf{0}\}\{\mathsf{0}\}^*\{\mathsf{1}\}\{\mathsf{1}\}^*. \end{aligned}$$

Next, we consider the reversal of languages and regular expressions. The *reversal of* a language L (L^R) is $\{\, w \mid w^R \in L \,\} = \{\, w^R \mid w \in L \,\}$. I.e., L^R is formed by reversing all of the elements of L. For example, $\{\mathsf{011}, \mathsf{1011}\}^R = \{\mathsf{110}, \mathsf{1101}\}$.

We define the *reversal of* a regular expression α ($\mathbf{rev}(\alpha)$) by recursion:

$$\begin{aligned} \mathbf{rev}(\%) &= \%; \\ \mathbf{rev}(\$) &= \$; \\ \mathbf{rev}(a) &= a, \text{ for all } a \in \mathbf{Sym}; \\ \mathbf{rev}(\alpha^*) &= \mathbf{rev}(\alpha)^*, \text{ for all } \alpha \in \mathbf{Reg}; \\ \mathbf{rev}(\alpha\,\beta) &= \mathbf{rev}(\beta)\,\mathbf{rev}(\alpha), \text{ for all } \alpha, \beta \in \mathbf{Reg}; \\ \mathbf{rev}(\alpha + \beta) &= \mathbf{rev}(\alpha) + \mathbf{rev}(\beta), \text{ for all } \alpha, \beta \in \mathbf{Reg}. \end{aligned}$$

For example $\mathbf{rev}(\mathsf{01} + (\mathsf{10})^*) = \mathsf{10} + (\mathsf{01})^*$.

Theorem 3.11.17
For all $\alpha \in \mathbf{Reg}$:

- $L(\mathbf{rev}(\alpha)) = L(\alpha)^R$;
- $\mathbf{alphabet}(\mathbf{rev}(\alpha)) = \mathbf{alphabet}(\alpha)$.

Next, we consider the prefix-, suffix- and substring-closures of languages, as well as the associated operations on automata. Suppose L is a language. Then:

- The *prefix-closure* of L (L^P) is $\{\, x \mid xy \in L, \text{ for some } y \in \mathbf{Str} \,\}$. I.e., L^P is all of the prefixes of elements of L. E.g., $\{\mathsf{012}, \mathsf{3}\}^P = \{\%, \mathsf{0}, \mathsf{01}, \mathsf{012}, \mathsf{3}\}$.
- The *suffix-closure* of L (L^S) is $\{\, y \mid xy \in L, \text{ for some } x \in \mathbf{Str} \,\}$. I.e., L^S is all of the suffixes of elements of L. E.g., $\{\mathsf{012}, \mathsf{3}\}^S = \{\%, \mathsf{2}, \mathsf{12}, \mathsf{012}, \mathsf{3}\}$.

- The *substring-closure* of L (L^{SS}) is $\{\, y \mid xyz \in L, \text{ for some } x, z \in \mathbf{Str}\,\}$. I.e., L^{SS} is all of the substrings of elements of L. E.g., $\{012, 3\}^{SS} = \{\%, 0, 1, 2, 01, 12, 012, 3\}$.

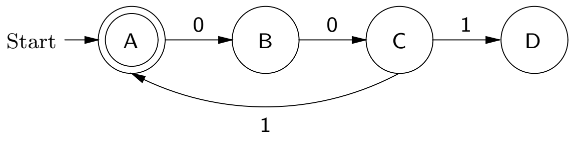
The following proposition shows that we can express suffix- and substring-closure in terms of prefix-closure and language reversal.

Proposition 3.11.18
For all languages L:

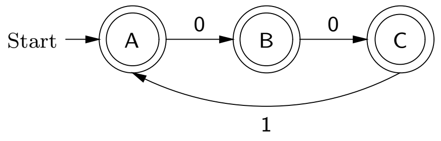
- $L^S = ((L^R)^P)^R$;
- $L^{SS} = (L^P)^S$.

Now, we define a function $\textbf{prefix} \in \textbf{NFA} \to \textbf{NFA}$ such that $L(\textbf{prefix}(M)) = L(M)^P$, for all $M \in \textbf{NFA}$. Given an NFA M, $\textbf{prefix}(M)$ is the NFA N that is constructed as follows. First, we simplify M, producing an NFA M' that is equivalent to M and either has no useless states, or consists of a single dead state and no-transitions. If M' has no useless states, then we let N be the same as M' except that $A_N = Q_N = Q_{M'}$, i.e., all states of N are accepting states. If M' consists of a single dead state and no transitions, then we let $N = M'$.

For example, suppose M is the NFA



so that $L(M) = \{001\}^*$. Then $\textbf{prefix}(M)$ is the NFA



which accepts $\{001\}^*\{\%, 0, 00\}$.

Theorem 3.11.19
For all $M \in \textbf{NFA}$:

- $L(\textbf{prefix}(M)) = L(M)^P$;
- $\textbf{alphabet}(\textbf{prefix}(M)) = \textbf{alphabet}(L(M))$.

Now we can define reversal, suffix-closure and substring-closure operations on NFAs as follows. The functions $\mathbf{rev}, \mathbf{suffix}, \mathbf{substring} \in \mathbf{NFA} \to \mathbf{NFA}$ are defined by:

$$\begin{aligned}\mathbf{rev}(M) &= \mathbf{efaToNFA}(\mathbf{faToEFA}(\mathbf{regToFA}(\mathbf{rev}(\mathbf{faToReg}(M))))),\\ \mathbf{suffix}(M) &= \mathbf{rev}(\mathbf{prefix}(\mathbf{rev}(M))),\\ \mathbf{substring}(M) &= \mathbf{suffix}(\mathbf{prefix}(M)).\end{aligned}$$

Theorem 3.11.20
For all $M \in \mathbf{NFA}$:

- $L(\mathbf{rev}(M)) = L(M)^R$;
- $L(\mathbf{suffix}(M)) = L(M)^S$;
- $L(\mathbf{substring}(M)) = L(M)^{SS}$.

Suppose L is a language, and f is a bijection from a set of symbols that is a superset of $\mathbf{alphabet}(L)$ (maybe $\mathbf{alphabet}(L)$ itself) to some set of symbols. Then the *renaming of L using f* (L^f) is formed by applying f to every symbol of every string of L. For example, if $L = \{012, 12\}$ and $f = \{(0,1),(1,2),(2,3)\}$, then $L^f = \{123, 23\}$.

Let $X = \{\, (\alpha, f) \mid \alpha \in \mathbf{Reg}$ and f is a bijection from a set of symbols that is a superset of $\mathbf{alphabet}(\alpha)$ to some set of symbols$\,\}$. The function $\mathbf{renameAlphabet} \in X \to \mathbf{Reg}$ takes in a pair (α, f) and returns the regular expression produced from α by renaming each sub-tree of the form a, for $a \in \mathbf{Sym}$, to $f(a)$. For example, $\mathbf{renameAlphabet}(012 + 12, \{(0,1),(1,2),(2,3)\}) = 123 + 23$.

Theorem 3.11.21
For all $\alpha \in \mathbf{Reg}$ and bijections f from sets of symbols that are supersets of $\mathbf{alphabet}(\alpha)$ to sets of symbols:

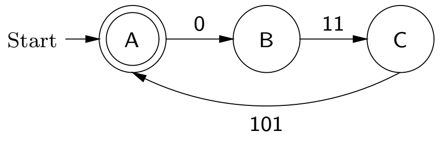
- $L(\mathbf{renameAlphabet}(\alpha, f)) = L(\alpha)^f$;
- $\mathbf{alphabet}(\mathbf{renameAlphabet}(\alpha, f)) = \{\, f(a) \mid a \in \mathbf{alphabet}(\alpha)\,\}$.

For example, if $f = \{(0,1),(1,2),(2,3)\}$, then

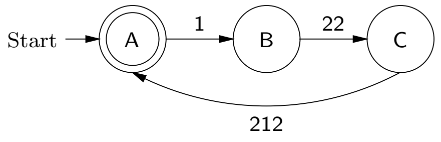
$$\begin{aligned}L(\mathbf{renameAlphabet}(012 + 12, f)) &= L(012 + 12)^f = \{012, 12\}^f\\ &= \{123, 23\}.\end{aligned}$$

Let $X = \{\, (M, f) \mid M \in \mathbf{FA}$ and f is a bijection from a set of symbols that is a superset of $\mathbf{alphabet}(M)$ to some set of symbols$\,\}$. The function

renameAlphabet $\in X \to \mathbf{FA}$ takes in a pair (M, f) and returns the FA produced from M by renaming each symbol of each label of each transition using f. For example, if M is the FA



and $f = \{(0, 1), (1, 2)\}$, then **renameAlphabet**(M, f) is the FA



Theorem 3.11.22
For all $M \in \mathbf{FA}$ *and bijections* f *from sets of symbols that are supersets of* **alphabet**(M) *to sets of symbols:*

- $L(\mathbf{renameAlphabet}(M, f)) = L(M)^f$*;*
- $\mathbf{alphabet}(\mathbf{renameAlphabet}(M, f)) = \{\, f(a) \mid a \in \mathbf{alphabet}(M)\,\}$*;*
- *if* M *is an EFA, then* **renameAlphabet**(M, f) *is an EFA;*
- *if* M *is an NFA, then* **renameAlphabet**(M, f) *is an NFA;*
- *if* M *is a DFA, then* **renameAlphabet**(M, f) *is a DFA.*

The preceding operations on regular expressions and finite automata give us the following theorems.

Theorem 3.11.23
Suppose $L, L_1, L_2 \in \mathbf{RegLan}$*. Then:*

(1) $L_1 \cup L_2 \in \mathbf{RegLan}$*;*

(2) $L_1 L_2 \in \mathbf{RegLan}$*;*

(3) $L^* \in \mathbf{RegLan}$*;*

(4) $L_1 \cap L_2 \in \mathbf{RegLan}$*;*

(5) $L_1 - L_2 \in \mathbf{RegLan}$*.*

Proof. Parts (1)–(5) hold because of the operations **union**, **concat** and **closure** on FAs, the operation **inter** on EFAs, the operation **minus** on DFAs, and Theorem 3.11.11. □

Theorem 3.11.24
Suppose $L \in \textbf{RegLan}$. *Then:*

(1) $L^R \in \textbf{RegLan}$*;*

(2) $L^P \in \textbf{RegLan}$*;*

(3) $L^S \in \textbf{RegLan}$*;*

(4) $L^{SS} \in \textbf{RegLan}$*;*

(5) $L^f \in \textbf{RegLan}$, *where* f *is a bijection from a set of symbols that is a superset of* $\textbf{alphabet}(L)$ *to some set of symbols.*

Proof. Parts (1)–(5) hold because of the operation **rev** on regular expressions, the operations **prefix**, **suffix** and **substring** on NFAs, the operation **renameAlphabet** on regular expressions, and Theorem 3.11.11. □

The Forlan module `EFA` defines the function

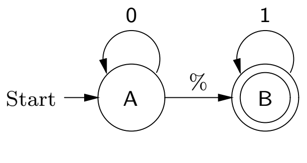
```
val inter : efa * efa -> efa
```

which corresponds to **inter**. It is also inherited by the modules `DFA` and `NFA`. The Forlan module DFA defines the functions

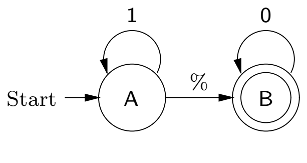
```
val complement : dfa * sym set -> dfa
val minus      : dfa * dfa -> dfa
```

which correspond to **complement** and **minus**.

Suppose the identifiers `efa1` and `efa2` of type `efa` are bound to our example EFAs M_1 and M_2:



(M_1)



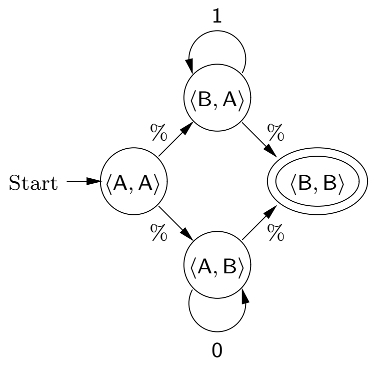
(M_2)

Then, we can construct $\textbf{inter}(M_1, M_2)$ as follows:

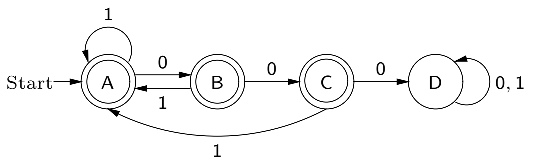
```
- val efa = EFA.inter(efa1, efa2);
```

```
val efa = - : efa
- EFA.output("", efa);
{states}
<A,A>, <A,B>, <B,A>, <B,B>
{start state}
<A,A>
{accepting states}
<B,B>
{transitions}
<A,A>, % -> <A,B> | <B,A>; <A,B>, % -> <B,B>;
<A,B>, 0 -> <A,B>; <B,A>, % -> <B,B>; <B,A>, 1 -> <B,A>
val it = () : unit
```

Thus `efa` is bound to the EFA



Suppose `dfa` is bound to our example DFA M

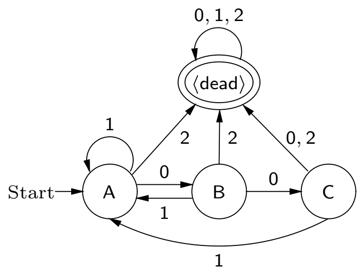


Then we can construct the DFA **complement**$(M, \{2\})$ as follows:

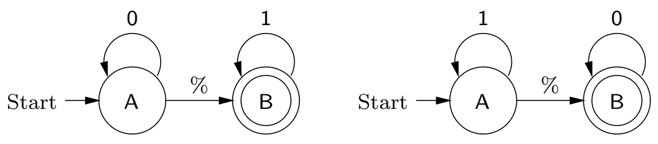
```
- val dfa' = DFA.complement(dfa, SymSet.input "");
@ 2
@ .
val dfa' = - : dfa
- DFA.output("", dfa');
{states}
A, B, C, <dead>
{start state}
A
```

```
{accepting states}
<dead>
{transitions}
A, 0 -> B; A, 1 -> A; A, 2 -> <dead>; B, 0 -> C;
B, 1 -> A; B, 2 -> <dead>; C, 0 -> <dead>; C, 1 -> A;
C, 2 -> <dead>; <dead>, 0 -> <dead>; <dead>, 1 -> <dead>;
<dead>, 2 -> <dead>
val it = () : unit
```

Thus `dfa'` is bound to the DFA



Suppose the identifiers `efa1` and `efa2` of type `efa` are bound to our example EFAs M_1 and M_2:



(M_1) (M_2)

We can construct an EFA that accepts $L(M_1) - L(M_2)$ as follows:

```
- val dfa1 = nfaToDFA(efaToNFA efa1);
val dfa1 = - : dfa
- val dfa2 = nfaToDFA(efaToNFA efa2);
val dfa2 = - : dfa
- val dfa = DFA.minus(dfa1, dfa2);
val dfa = - : dfa
- val efa = injDFAToEFA dfa;
val efa = - : efa
- EFA.accepted efa (Str.input "");
@ 01
@ .
val it = true : bool
- EFA.accepted efa (Str.input "");
@ 0
@ .
val it = false : bool
```

Note that we had to first convert `efa1` and `efa2` to DFAs, because the module `EFA` doesn't include an intersection operation.

Next, we see how we can carry out the reversal and alphabet-renaming of regular expressions in Forlan. The Forlan module `Reg` defines the functions

```
val rev            : reg -> reg
val renameAlphabet : reg * sym_rel -> reg
```

which correspond to **rev** and **renameAlphabet** (`renameAlphabet` issues an error message and raises an exception if its second argument isn't legal). Here is an example of how these functions can be used:

```
- val reg = Reg.fromString "(012)*(21)";
val reg = - : reg
- val rel = SymRel.fromString "(0, 1), (1, 2), (2, 3)";
val rel = - : sym_rel
- Reg.output("", Reg.rev reg);
(12)((21)0)*
val it = () : unit
- Reg.output("", Reg.renameAlphabet(reg, rel));
(123)*32
val it = () : unit
```

Next, we see how we can carry out the prefix-closure of NFAs in Forlan. The Forlan module `NFA` defines the function

```
val prefix : nfa -> nfa
```

which corresponds to **prefix**. Here is an example of how this function can be used:

```
- val nfa = NFA.input "";
@ {states}
@ A, B, C, D
@ {start state}
@ A
@ {accepting states}
@ A
@ {transitions}
@ A, 0 -> B; B, 0 -> C; C, 1 -> A; C, 1 -> D
@ .
val nfa = - : nfa
- val nfa' = NFA.prefix nfa;
```

```
val nfa' = - : nfa
- NFA.output("", nfa');
{states}
A, B, C
{start state}
A
{accepting states}
A, B, C
{transitions}
A, 0 -> B; B, 0 -> C; C, 1 -> A
val it = () : unit
```

Finally, we see how we can carry out alphabet-renaming of finite automata using Forlan. The Forlan module `FA` defines the function

```
val renameAlphabet : FA * sym_rel -> FA
```

which corresponds **renameAlphabet** (it issues an error message and raises an exception if its second argument isn't legal). This function is also inherited by the modules `DFA`, `NFA` and `EFA`. Here is an example of how one of these functions can be used:

```
- val dfa = DFA.input "";
@ {states}
@ A, B
@ {start state}
@ A
@ {accepting states}
@ A
@ {transitions}
@ A, 0 -> B; B, 0 -> A;
@ A, 1 -> A; B, 1 -> B
@ .
val dfa = - : dfa
- val rel = SymRel.fromString "(0, a), (1, b)";
val rel = - : sym_rel
- val dfa' = DFA.renameAlphabet(dfa, rel);
val dfa' = - : dfa
- DFA.output("", dfa');
{states}
A, B
{start state}
A
{accepting states}
A
```

```
{transitions}
A, a -> B; A, b -> A; B, a -> A; B, b -> B
val it = () : unit
```

3.12 Equivalence-testing and Minimization of Deterministic Finite Automata

In this section, we give algorithms for: testing whether two DFAs are equivalent; and minimizing the alphabet size and number of states of a DFA. We also show how to use the Forlan implementations of these algorithms. In addition, we consider an alternative way of synthesizing DFAs, using DFA minimization plus the operations on automata and regular expressions of the previous section.

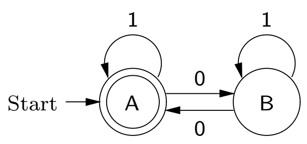
Suppose M and N are DFAs. To check whether they are equivalent, we can proceed as follows.

First, we need to convert M and N into DFAs with identical alphabets. Let $\Sigma = \textbf{alphabet}(M) \cup \textbf{alphabet}(N)$, and define the DFAs M' and N' by:

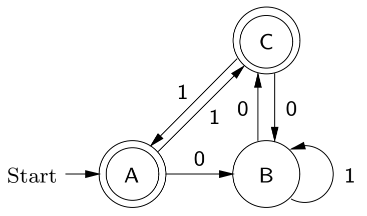
$$M' = \textbf{determSimplify}(M, \Sigma),$$
$$N' = \textbf{determSimplify}(N, \Sigma).$$

Since $\textbf{alphabet}(L(M)) \subseteq \textbf{alphabet}(M) \subseteq \Sigma$, we have that $\textbf{alphabet}(M') = \textbf{alphabet}(L(M)) \cup \Sigma = \Sigma$. Similarly, $\textbf{alphabet}(N') = \Sigma$. Furthermore, $M' \approx M$ and $N' \approx N$, so that it will suffice to determine whether M' and N' are equivalent.

For example, if M and N are the DFAs



(M)



(N)

then $\Sigma = \{0, 1\}$, $M' = M$ and $N' = N$.

Next, we generate the least subset X of $Q_{M'} \times Q_{N'}$ such that

- $(s_{M'}, s_{N'}) \in X$;

- for all $q \in Q_{M'}$, $r \in Q_{N'}$ and $a \in \Sigma$, if $(q, r) \in X$, then $(\delta_{M'}(q, a), \delta_{N'}(r, a)) \in X$.

With our example DFAs M' and N', we have that

- $(\mathsf{A}, \mathsf{A}) \in X$;
- since $(\mathsf{A}, \mathsf{A}) \in X$, we have that $(\mathsf{B}, \mathsf{B}) \in X$ and $(\mathsf{A}, \mathsf{C}) \in X$;
- since $(\mathsf{B}, \mathsf{B}) \in X$, we have that (again) $(\mathsf{A}, \mathsf{C}) \in X$ and (again) $(\mathsf{B}, \mathsf{B}) \in X$;
- since $(\mathsf{A}, \mathsf{C}) \in X$, we have that (again) $(\mathsf{B}, \mathsf{B}) \in X$ and (again) $(\mathsf{A}, \mathsf{A}) \in X$.

Back in the general case, we have the following lemmas.

Lemma 3.12.1
For all $w \in \Sigma^*$, $(\delta_{M'}(s_{M'}, w), \delta_{N'}(s_{N'}, w)) \in X$.

Proof. By left string induction on w. $\square$

Lemma 3.12.2
For all $q \in Q_{M'}$ and $r \in Q_{N'}$, if $(q, r) \in X$, then there is a $w \in \Sigma^*$ such that $q = \delta_{M'}(s_{M'}, w)$ and $r = \delta_{N'}(s_{N'}, w)$.

Proof. By induction on X. $\square$

Finally, we check that, for all $(q, r) \in X$,

$$q \in A_{M'} \quad \text{iff} \quad r \in A_{N'}.$$

If this is true, we say that the machines are equivalent; otherwise we say they are not equivalent.

Suppose every pair $(q, r) \in X$ consists of two accepting states or of two non-accepting states. Suppose, toward a contradiction, that $L(M') \neq L(N')$. Then there is a string w that is accepted by one of the machines but is not accepted by the other. Since both machines have alphabet Σ, Lemma 3.12.1 tells us that $(\delta_{M'}(s_{M'}, w), \delta_{N'}(s_{N'}, w)) \in X$. But one side of this pair is an accepting state and the other is a non-accepting one—contradiction. Thus $L(M') = L(N')$.

Suppose we find a pair $(q, r) \in X$ such that one of q and r is an accepting state but the other is not. By Lemma 3.12.2, it will follow that there is a string w that is accepted by one of the machines but not accepted by the other one, i.e., that $L(M') \neq L(N')$.

In the case of our example, we have that $X = \{(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{B}),(\mathsf{A},\mathsf{C})\}$. Since (A,A) and (A,C) are pairs of accepting states, and (B,B) is a pair of non-accepting states, it follows that $L(M') = L(N')$. Hence $L(M) = L(N)$.

We can easily modify our algorithm so that, when two machines are not equivalent, it explains why:

- giving a string that is accepted by the first machine but not by the second; and/or

- giving a string that is accepted by the second machine but not by the first.

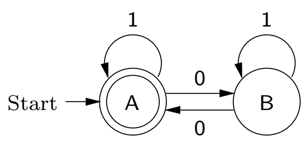
We can even arrange for these strings to be of minimum length. The Forlan implementation of our algorithm always produces minimum-length counterexamples.

The Forlan module `DFA` defines the functions:

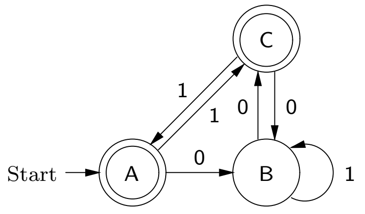
```
val relationship : dfa * dfa -> unit
val subset       : dfa * dfa -> bool
val equivalent   : dfa * dfa -> bool
```

The function `relationship` figures out the relationship between the languages accepted by two DFAs (are they equal, is one a proper subset of the other, do they have an empty intersection), and supplies minimum-length counterexamples to justify negative answers. The function `subset` tests whether its first argument's language is a subset of its second argument's language. The function `equivalent` tests whether two DFAs are equivalent. Note that `subset` (when turned into a function of type `reg * reg -> bool`—see below) can be used in conjunction with `Reg.simplify` (see Section 3.2).

For example, suppose `dfa1` and `dfa2` of type `dfa` are bound to our example DFAs M and N, respectively:



(M)

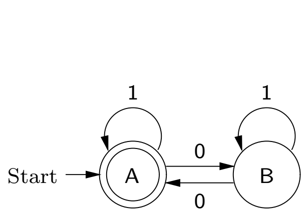


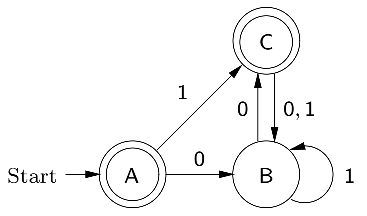
(N)

We can verify that these machines are equivalent as follows:

```
- DFA.relationship(dfa1, dfa2);
languages are equal
val it = () : unit
```

On the other hand, suppose that `dfa3` and `dfa4` of type `dfa` are bound to the DFAs:





We can find out why these machines are not equivalent as follows:

```
- DFA.relationship(dfa3, dfa4);
neither language is a subset of the other language : "11"
is in first language but is not in second language; "110"
is in second language but is not in first language
val it = () : unit
```

We can find the relationship between the languages denoted by regular expressions `reg1` and `reg2` by:

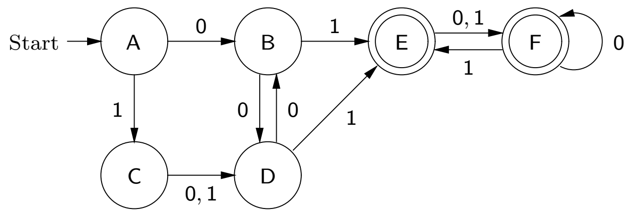
- converting `reg1` and `reg2` to DFAs `dfa1` and `dfa2`, and then
- running `DFA.relationship(dfa1, dfa2)` to find the relationship between those DFAs.

Of course, we can define an ML/Forlan function that carries out these actions:

```
- val regToDFA = nfaToDFA o efaToNFA o faToEFA o regToFA;
val regToDFA = fn : reg -> dfa
- fun relationshipReg(reg1, reg2) =
=       DFA.relationship(regToDFA reg1, regToDFA reg2);
val relationshipReg = fn : reg * reg -> unit
```

Now, we consider an algorithm for minimizing the sizes of the alphabet and set of states of a DFA M. First, we minimize the size of M's alphabet, and make the automaton be deterministically simplified, by letting $M' = \mathbf{determSimplify}(M, \emptyset)$. Thus $M' \approx M$ and $\mathbf{alphabet}(M') = \mathbf{alphabet}(L(M))$.

For example, if M is the DFA



then $M' = M$.

Next, we let X be the least subset of $Q_{M'} \times Q_{M'}$ such that:

1. $A_{M'} \times (Q_{M'} - A_{M'}) \subseteq X$;
2. $(Q_{M'} - A_{M'}) \times A_{M'} \subseteq X$;
3. for all $q, q', r, r' \in Q_{M'}$ and $a \in \textbf{alphabet}(M')$, if $(q, r) \in X$, $(q', a, q) \in T_{M'}$ and $(r', a, r) \in T_{M'}$, then $(q', r') \in X$.

We read "$(q, r) \in X$" as "q and r cannot be merged". The idea of (1) and (2) is that an accepting state can never be merged with a non-accepting state. And (3) says that if q and r can't be merged, and we can get from q' to q by processing an a, and from r' to r by processing an a, then q' and r' also can't be merged—since if we merged q' and r', there would have to be an a-transition from the merged state to the merging of q and r.

In the case of our example M', (1) tells us to add the pairs (E, A), (E, B), (E, C), (E, D), (F, A), (F, B), (F, C) and (F, D) to X. And, (2) tells us to add the pairs (A, E), (B, E), (C, E), (D, E), (A, F), (B, F), (C, F) and (D, F) to X.

Now we use rule (3) to compute the rest of X's elements. To begin with, we must handle each pair that has already been added to X.

- Since there are no transitions leading into A, no pairs can be added using (E, A), (A, E), (F, A) and (A, F).
- Since there are no $\mathsf{0}$-transitions leading into E, and there are no $\mathsf{1}$-transitions leading into B, no pairs can be added using (E, B) and (B, E).
- Since $(\mathsf{E}, \mathsf{C}), (\mathsf{C}, \mathsf{E}) \in X$ and $(\mathsf{B}, \mathsf{1}, \mathsf{E})$, $(\mathsf{D}, \mathsf{1}, \mathsf{E})$, $(\mathsf{F}, \mathsf{1}, \mathsf{E})$ and $(\mathsf{A}, \mathsf{1}, \mathsf{C})$ are the $\mathsf{1}$-transitions leading into E and C, we add (B, A) and (A, B), and (D, A) and (A, D) to X; we would also have added (F, A) and (A, F) to X if they hadn't been previously added. Since there are no $\mathsf{0}$-transitions into E, nothing can be added to X using (E, C) and (C, E) and $\mathsf{0}$-transitions.

- Since $(\mathsf{E},\mathsf{D}),(\mathsf{D},\mathsf{E}) \in X$ and $(\mathsf{B},1,\mathsf{E})$, $(\mathsf{D},1,\mathsf{E})$, $(\mathsf{F},1,\mathsf{E})$ and $(\mathsf{C},1,\mathsf{D})$ are the 1-transitions leading into E and D, we add (B,C) and (C,B), and (D,C) and (C,D) to X; we would also have added (F,C) and (C,F) to X if they hadn't been previously added. Since there are no 0-transitions into E, nothing can be added to X using (E,D) and (D,E) and 0-transitions.

- Since $(\mathsf{F},\mathsf{B}),(\mathsf{B},\mathsf{F}) \in X$ and $(\mathsf{E},0,\mathsf{F})$, $(\mathsf{F},0,\mathsf{F})$, $(\mathsf{A},0,\mathsf{B})$, and $(\mathsf{D},0,\mathsf{B})$ are the 0-transitions leading into F and B, we would have to add the following pairs to X, if they were not already present: (E,A), (A,E), (E,D), (D,E), (F,A), (A,F), (F,D), (D,F). Since there are no 1-transitions leading into B, no pairs can be added using (F,B) and (B,F) and 1-transitions.

- Since $(\mathsf{F},\mathsf{C}),(\mathsf{C},\mathsf{F}) \in X$ and $(\mathsf{E},1,\mathsf{F})$ and $(\mathsf{A},1,\mathsf{C})$ are the 1-transitions leading into F and C, we would have to add (E,A) and (A,E) to X if these pairs weren't already present. Since there are no 0-transitions leading into C, no pairs can be added using (F,C) and (C,F) and 0-transitions.

- Since $(\mathsf{F},\mathsf{D}),(\mathsf{D},\mathsf{F}) \in X$ and $(\mathsf{E},0,\mathsf{F})$, $(\mathsf{F},0,\mathsf{F})$, $(\mathsf{B},0,\mathsf{D})$ and $(\mathsf{C},0,\mathsf{D})$ are the 0-transitions leading into F and D, we would add (E,B), (B,E), (E,C), (C,E), (F,B), (B,F), (F,C), and (C,F) to X, if these pairs weren't already present. Since $(\mathsf{F},\mathsf{D}),(\mathsf{D},\mathsf{F}) \in X$ and $(\mathsf{E},1,\mathsf{F})$ and $(\mathsf{C},1,\mathsf{D})$ are the 1-transitions leading into F and D, we would add (E,C) and (C,E) to X, if these pairs weren't already in X.

We've now handled all of the elements of X that were added using rules (1) and (2). We must now handle the pairs that were subsequently added: (A,B), (B,A), (A,D), (D,A), (B,C), (C,B), (C,D), (D,C).

- Since there are no transitions leading into A, no pairs can be added using (A,B), (B,A), (A,D) and (D,A).

- Since there are no 1-transitions leading into B, and there are no 0-transitions leading into C, no pairs can be added using (B,C) and (C,B).

- Since $(\mathsf{C},\mathsf{D}),(\mathsf{D},\mathsf{C}) \in X$ and $(\mathsf{A},1,\mathsf{C})$ and $(\mathsf{C},1,\mathsf{D})$ are the 1-transitions leading into C and D, we add the pairs (A,C) and (C,A) to X. Since there are no 0-transitions leading into C, no pairs can be added to X using (C,D) and (D,C) and 0-transitions.

Now, we must handle the pairs that were added in the last phase: (A,C) and (C,A).

- Since there are no transitions leading into A, no pairs can be added using (A,C) and (C,A).

Since we have handled all the pairs we added to X, we are now done. Here are the 26 elements of X: (A,B), (A,C), (A,D), (A,E), (A,F), (B,A), (B,C), (B,E), (B,F), (C,A), (C,B), (C,D), (C,E), (C,F), (D,A), (D,C), (D,E), (D,F), (E,A), (E,B), (E,C), (E,D), (F,A), (F,B), (F,C), (F,D).

Going back to the general case, we now let the relation $Y = (Q_{M'} \times Q_{M'}) - X$. It turns out that Y is reflexive on $Q_{M'}$, symmetric and transitive, i.e., it is what is called an equivalence relation on $Q_{M'}$. We read "$(q, r) \in Y$" as "q and r can be merged".

Back with our example, we have that Y is

$$\begin{gathered}\{(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{B}),(\mathsf{C},\mathsf{C}),(\mathsf{D},\mathsf{D}),(\mathsf{E},\mathsf{E}),(\mathsf{F},\mathsf{F})\}\\ \cup\\ \{(\mathsf{B},\mathsf{D}),(\mathsf{D},\mathsf{B}),(\mathsf{F},\mathsf{E}),(\mathsf{E},\mathsf{F})\}.\end{gathered}$$

In order to define the DFA N that is the result of our minimization algorithm, we need a bit more notation. As in Section 3.10, we write $\overline{P}$ for the result of coding a finite set of symbols P as a symbol. E.g., $\overline{\{\mathsf{B},\mathsf{A}\}} = \langle\mathsf{A},\mathsf{B}\rangle$. If $q \in Q_{M'}$, we write $[q]$ for $\{\, r \in Q_{M'} \mid (r, q) \in Y \,\}$, which is called the *equivalence class* of q. In other words, $[q]$ consists of all of the states that are mergable with q (including itself). If P is a nonempty, finite set of symbols, then we write $\min(P)$ for the least element of P, according to our standard ordering on symbols.

Let $Z = \{\, [q] \mid q \in Q_{M'} \,\}$. In the case of our example, Z is

$$\{\{\mathsf{A}\},\{\mathsf{B},\mathsf{D}\},\{\mathsf{C}\},\{\mathsf{E},\mathsf{F}\}\}.$$

We define our DFA N as follows:

- $Q_N = \{\, \overline{P} \mid P \in Z \,\}$;
- $s_N = \overline{[s_{M'}]}$;
- $A_N = \{\, \overline{P} \mid P \in Z \text{ and } \min(P) \in A_{M'} \,\}$;
- $T_N = \{\, (\overline{P}, a, \overline{[\delta_{M'}(\min(P), a)]}) \mid P \in Z \text{ and } a \in \textbf{alphabet}(M') \,\}$.

(In the definitions of A_N and T_N any element of P could be substituted for $\min(P)$.)

In the case of our example, we have that

- $Q_N = \{\langle \mathsf{A}\rangle, \langle \mathsf{B}, \mathsf{D}\rangle, \langle \mathsf{C}\rangle, \langle \mathsf{E}, \mathsf{F}\rangle\}$;
- $s_N = \langle A\rangle$;
- $A_N = \{\langle \mathsf{E}, \mathsf{F}\rangle\}$.

We compute the elements of T_N as follows.

- Since $\{\mathsf{A}\} \in Z$ and $[\delta_{M'}(\mathsf{A}, \mathsf{0})] = [\mathsf{B}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{A}\rangle, \mathsf{0}, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{A}\} \in Z$ and $[\delta_{M'}(\mathsf{A}, \mathsf{1})] = [\mathsf{C}] = \{\mathsf{C}\}$, we have that $(\langle \mathsf{A}\rangle, \mathsf{1}, \langle \mathsf{C}\rangle) \in T_N$.

- Since $\{\mathsf{C}\} \in Z$ and $[\delta_{M'}(\mathsf{C}, \mathsf{0})] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{C}\rangle, \mathsf{0}, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{C}\} \in Z$ and $[\delta_{M'}(\mathsf{C}, \mathsf{1})] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{C}\rangle, \mathsf{1}, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

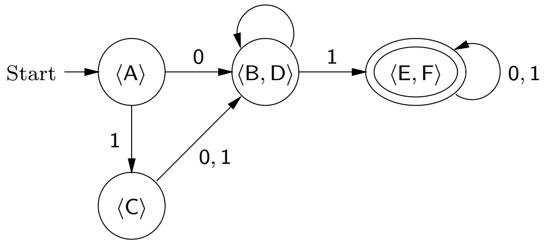
- Since $\{\mathsf{B}, \mathsf{D}\} \in Z$ and $[\delta_{M'}(\mathsf{B}, \mathsf{0})] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{B}, \mathsf{D}\rangle, \mathsf{0}, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{B}, \mathsf{D}\} \in Z$ and $[\delta_{M'}(\mathsf{B}, \mathsf{1})] = [\mathsf{E}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{B}, \mathsf{D}\rangle, \mathsf{1}, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

- Since $\{\mathsf{E}, \mathsf{F}\} \in Z$ and $[\delta_{M'}(\mathsf{E}, \mathsf{0})] = [\mathsf{F}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{E}, \mathsf{F}\rangle, \mathsf{0}, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

  Since $\{\mathsf{E}, \mathsf{F}\} \in Z$ and $[\delta_{M'}(\mathsf{E}, \mathsf{1})] = [\mathsf{F}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{E}, \mathsf{F}\rangle, \mathsf{1}, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

Thus our DFA N is:



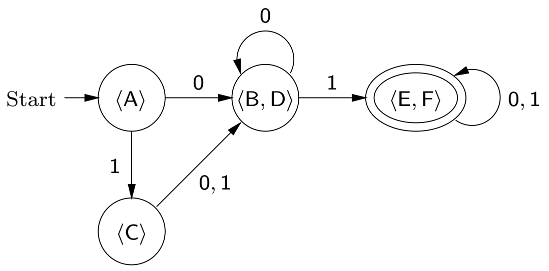
We define a function **minimize** $\in$ **DFA** $\rightarrow$ **DFA** by: **minimize**(M) is the result of running the above algorithm on input M.

We have the following theorem:

Theorem 3.12.3
*For all $M \in$ **DFA**:*

- **minimize**$(M) \approx M$;
- **alphabet**(**minimize**(M)) $=$ **alphabet**$(L(M))$;
- **minimize**(M) *is deterministically simplified;*
- *for all $N \in$ **DFA**, if $N \approx M$, then* **alphabet**(**minimize**(M)) $\subseteq$ **alphabet**(N) *and* $|Q_{\mathbf{minimize}(M)}| \leq |Q_N|$;
- *for all $N \in$ **DFA**, if $N \approx M$ and N has the same alphabet and number of states as* **minimize**(M), *then N is isomorphic to* **minimize**(M).

Thus there are no DFAs with three or fewer states that are equivalent to our example DFA M. And



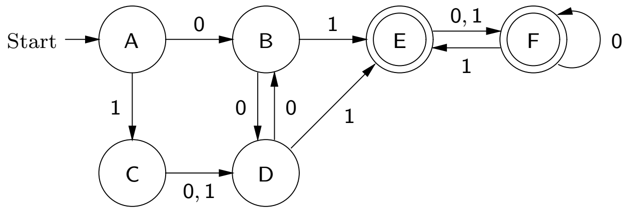
is, up to isomorphism, the only four state DFA with alphabet $\{0, 1\}$ that is equivalent to M.

The Forlan module DFA includes the function

```
val minimize : dfa -> dfa
```

for minimizing DFAs.

For example, if `dfa` of type `dfa` is bound to our example DFA



then we can minimize the alphabet size and number of states of `dfa` as follows:

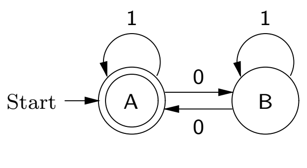
```
- val dfa' = DFA.minimize dfa;
val dfa' = - : dfa
- DFA.output("", dfa');
{states}
<A>, <C>, <B,D>, <E,F>
{start state}
<A>
{accepting states}
<E,F>
{transitions}
<A>, 0 -> <B,D>; <A>, 1 -> <C>; <C>, 0 -> <B,D>;
<C>, 1 -> <B,D>; <B,D>, 0 -> <B,D>; <B,D>, 1 -> <E,F>;
<E,F>, 0 -> <E,F>; <E,F>, 1 -> <E,F>
val it = () : unit
```

Because of DFA minimization plus the operations on automata and regular expressions of Section 3.11, we now have an alternative way of synthesizing DFAs.

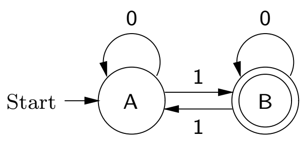
For example, suppose we wish to find a DFA M such that $L(M) = X$, where $X = \{\, w \in \{0,1\}^* \mid w$ has an even number of 0's and an odd number of 1's$\,\}$.

First, we can note that $X = Y_1 \cap Y_2$, where $Y_1 = \{\, w \in \{0,1\}^* \mid w$ has an even number of 0's$\,\}$ and $Y_2 = \{\, w \in \{0,1\}^* \mid w$ has an odd number of 1's$\,\}$. Since we have an intersection operation on DFAs, if we can find DFAs accepting Y_1 and Y_2, we can combine them into a DFA that accepts X.

Let N_1 and N_2 be the DFAs



(N_1)



(N_2)

It is easy to prove that $L(N_1) = Y_1$ and $L(N_2) = Y_2$. Let M be the DFA

$$\mathbf{renameStatesCanonically}(\mathbf{minimize}(\mathbf{inter}(N_1, N_2))).$$

Then

$$\begin{aligned}
L(M) &= L(\mathbf{renameStatesCanonically}(\mathbf{minimize}(\mathbf{inter}(N_1, N_2)))) \\
&= L(\mathbf{minimize}(\mathbf{inter}(N_1, N_2))) \\
&= L(\mathbf{inter}(N_1, N_2)) \\
&= L(N_1) \cap L(N_2) \\
&= Y_1 \cap Y_2 \\
&= X,
\end{aligned}$$

showing that M is correct.

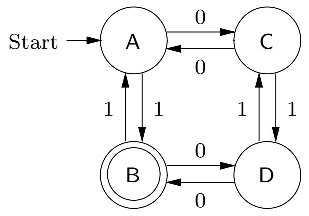
Suppose M' is a DFA that accepts X. Since $M' \approx \mathbf{inter}(N_1, N_2)$, we have that $\mathbf{minimize}(\mathbf{inter}(N_1, N_2))$, and thus M, has no more states than M'. Thus M has as few states as is possible.

But how do we figure out what the components of M are, so that, e.g., we can draw M? In a simple case like this, we could apply the definitions **inter**, **minimize** and **renameStatesCanonically**, and work out the answer. But, for more complex examples, there would be far too much detail involved for this to be a practical approach.

Instead, we can use Forlan to compute the answer. Suppose `dfa1` and `dfa2` of type `dfa` are N_1 and N_2, respectively. The we can proceed as follows:

```
- val dfa' = DFA.minimize(DFA.inter(dfa1, dfa2));
val dfa' = - : dfa
- val dfa = DFA.renameStatesCanonically dfa';
val dfa = - : dfa
- DFA.output("", dfa);
{states}
A, B, C, D
{start state}
A
{accepting states}
B
{transitions}
A, 0 -> C; A, 1 -> B; B, 0 -> D; B, 1 -> A; C, 0 -> A;
C, 1 -> D; D, 0 -> B; D, 1 -> C
val it = () : unit
```

Thus M is:



Of course, this claim assumes that Forlan is correctly implemented.

We conclude this section by considering a second, more involved example of DFA synthesis. Given a string $w \in \{0,1\}^*$, we say that:

- w *stutters* iff aa is a substring of w, for some $a \in \{0,1\}$;
- w is *long* iff $|w| \geq 5$.

So, e.g., 1001 and 10110 both stutter, but 01010 and 101 don't. Saying that strings of length 5 or more are "long" is arbitrary; what follows can be repeated with different choices of when strings are long.

Let the language **AllLongStutter** be

$$\{\, w \in \{0,1\}^* \mid \text{for all substrings } v \text{ of } w, \text{ if } v \text{ is long, then } v \text{ stutters} \,\}.$$

In other words, a string of 0's and 1's is in **AllLongStutter** iff every long substring of this string stutters. Since every substring of 0010110 of length five stutters, every long substring of this string stutters, and thus the string is in **AllLongStutter**. On the other hand, 0010100 is not in **AllLongStutter**, because 01010 is a long, non-stuttering substring of this string.

Let's consider the problem of finding a DFA that accepts this language. One possibility is to reduce this problem to that of finding a DFA that accepts the complement of **AllLongStutter**. Then we'll be able to use our set difference operation on DFAs to build a DFA that accepts **AllLongStutter**. (We'll also need a DFA accepting $\{0,1\}^*$.) To form the complement of **AllLongStutter**, we negate the formula in **AllLongStutter**'s expression. Let **SomeLongNotStutter** be the language

$$\{\, w \in \{0,1\}^* \mid \text{there is a substring } v \text{ of } w \text{ such that } v \text{ is long and doesn't stutter} \,\}.$$

Lemma 3.12.4
AllLongStutter $= \{0,1\}^* -$ **SomeLongNotStutter**.

Proof. Suppose $w \in \textbf{AllLongStutter}$, so that $w \in \{\mathsf{0},\mathsf{1}\}^*$ and, for all substrings v of w, if v is long, then v stutters. Suppose, toward a contradiction, that $w \in \textbf{SomeLongNotStutter}$. Then there is a substring v of w such that v is long and doesn't stutter—contradiction. Thus $w \not\in \textbf{SomeLongNotStutter}$, completing the proof that $w \in \{\mathsf{0},\mathsf{1}\}^* - \textbf{SomeLongNotStutter}$.

Suppose $w \in \{\mathsf{0},\mathsf{1}\}^* - \textbf{SomeLongNotStutter}$, so that $w \in \{\mathsf{0},\mathsf{1}\}^*$ and $w \not\in \textbf{SomeLongNotStutter}$. To see that $w \in \textbf{AllLongStutter}$, suppose v is a substring of w and v is long. Suppose, toward a contradiction, that v doesn't stutter. Then $w \in \textbf{SomeLongNotStutter}$—contradiction. Hence v stutters. $\square$

Next, it's convenient to work bottom-up for a bit. Let

$$\begin{aligned}
\textbf{Long} &= \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ is long}\,\},\\
\textbf{Stutter} &= \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ stutters}\,\},\\
\textbf{NotStutter} &= \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ doesn't stutter}\,\},\\
\textbf{LongAndNotStutter} &= \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ is long and doesn't stutter}\,\}.
\end{aligned}$$

The following lemma is easy to prove:

Lemma 3.12.5

(1) $\textbf{NotStutter} = \{\mathsf{0},\mathsf{1}\}^* - \textbf{Stutter}$.

(2) $\textbf{LongAndNotStutter} = \textbf{Long} \cap \textbf{NotStutter}$.

Clearly, we'll be able to find DFAs accepting **Long** and **Stutter**, respectively. Thus, we'll be able to use our set difference operation on DFAs to come up with a DFA that accepts **NotStutter**. Then, we'll be able to use our intersection operation on DFAs to come up with a DFA that accepts **LongAndNotStutter**.

What remains is to find a way of converting **LongAndNotStutter** to **SomeLongNotStutter**. Clearly, the former language is a subset of the latter one. But the two languages are not equal, since an element of the latter language may have the form xvy, where $x, y \in \{\mathsf{0},\mathsf{1}\}^*$ and $v \in \textbf{LongAndNotStutter}$. This suggests the following lemma:

Lemma 3.12.6
$\textbf{SomeLongNotStutter} = \{\mathsf{0},\mathsf{1}\}^* \, \textbf{LongAndNotStutter} \, \{\mathsf{0},\mathsf{1}\}^*$.

Proof. Suppose $w \in \textbf{SomeLongNotStutter}$, so that $w \in \{\mathsf{0},\mathsf{1}\}^*$ and there is a substring v of w such that v is long and doesn't stutter. Thus $v \in \textbf{LongAndNotStutter}$, and $w = xvy$ for some $x, y \in \{\mathsf{0},\mathsf{1}\}^*$. Hence $w = xvy \in \{\mathsf{0},\mathsf{1}\}^*\,\textbf{LongAndNotStutter}\,\{\mathsf{0},\mathsf{1}\}^*$.

Suppose $w \in \{\mathsf{0},\mathsf{1}\}^*\,\textbf{LongAndNotStutter}\,\{\mathsf{0},\mathsf{1}\}^*$, so that $w = xvy$ for some $x, y \in \{\mathsf{0},\mathsf{1}\}^*$ and $v \in \textbf{LongAndNotStutter}$. Hence v is long and doesn't stutter. Thus v is a long substring of w that doesn't stutter, showing that $w \in \textbf{SomeLongNotStutter}$. $\Box$

Because of the preceding lemma, we can build an EFA accepting **SomeLongNotStutter** from a DFA accepting $\{\mathsf{0},\mathsf{1}\}^*$ and our DFA accepting **LongAndNotStutter**, using our concatenation operation on EFAs. (We haven't given a concatenation operation on DFAs.) We can then convert this EFA to a DFA.

Now, let's take the preceding ideas and turn them into reality. First, we define functions $\textbf{regToEFA} \in \textbf{Reg} \to \textbf{EFA}$, $\textbf{efaToDFA} \in \textbf{EFA} \to \textbf{DFA}$, $\textbf{regToDFA} \in \textbf{Reg} \to \textbf{DFA}$ and $\textbf{minAndRen} \in \textbf{DFA} \to \textbf{DFA}$ by:

$$\begin{aligned}
\textbf{regToEFA} &= \textbf{faToEFA} \circ \textbf{regToFA},\\
\textbf{efaToDFA} &= \textbf{nfaToDFA} \circ \textbf{efaToNFA},\\
\textbf{regToDFA} &= \textbf{efaToDFA} \circ \textbf{regToEFA},\\
\textbf{minAndRen} &= \textbf{renameStatesCanonically} \circ \textbf{minimize}.
\end{aligned}$$

Lemma 3.12.7

(1) For all $\alpha \in \textbf{Reg}$, $L(\textbf{regToEFA}(\alpha)) = L(\alpha)$.

(2) For all $M \in \textbf{EFA}$, $L(\textbf{efaToDFA}(M)) = L(M)$.

(3) For all $\alpha \in \textbf{Reg}$, $L(\textbf{regToDFA}(\alpha)) = L(\alpha)$.

(4) For all $M \in \textbf{DFA}$, $L(\textbf{minAndRen}(M)) = L(M)$ and, for all $N \in \textbf{DFA}$, if $L(N) = L(M)$, then $\textbf{minAndRen}(M)$ has no more states than N.

Proof. We show the proof of Part (4), the proofs of the other parts being even easier. Suppose $M \in \textbf{DFA}$. By Theorem 3.12.3(1), we have that

$$\begin{aligned}
L(\textbf{minAndRen}(M)) &= L(\textbf{renameStatesCanonically}(\textbf{minimize}(M)))\\
&= L(\textbf{minimize}(M))\\
&= L(M).
\end{aligned}$$

Suppose $N \in$ **DFA** and $L(N) = L(M)$. By Theorem 3.12.3(4), we have that **minimize**(M) has no more states than N. Thus **renameStatesCanonically**(**minimize**(M)) has no more states than N, showing that **minAndRen**(M) has no more states than N. $\square$

Let the regular expression **allStrReg** be $(0+1)^*$. Clearly $L(\textbf{allStrReg}) = \{0,1\}^*$. Let the DFA **allStrDFA** be

$$\textbf{minAndRen}(\textbf{regToDFA}(\textbf{allStrReg})).$$

Lemma 3.12.8
$L(\textbf{allStrDFA}) = \{0,1\}^*$.

Proof. By Lemma 3.12.7, we have that

$$\begin{aligned} L(\textbf{allStrDFA}) &= L(\textbf{minAndRen}(\textbf{regToDFA}(\textbf{allStrReg}))) \\ &= L(\textbf{regToDFA}(\textbf{allStrReg})) \\ &= L(\textbf{allStrReg}) \\ &= \{0,1\}^*. \end{aligned}$$

$\square$

(Not surprisingly, **allStrDFA** will have a single state.) Let the EFA **allStrEFA** be the DFA **allStrDFA**. Thus $L(\textbf{allStrEFA}) = \{0,1\}^*$.

Let the regular expression **longReg** be

$$(0+1)^5(0+1)^*.$$

Lemma 3.12.9
$L(\textbf{longReg}) = \textbf{Long}$.

Proof. Since $L(\textbf{longReg}) = \{0,1\}^5\{0,1\}^*$, it will suffice to show that $\{0,1\}^5\{0,1\}^* = \textbf{Long}$.

Suppose $w \in \{0,1\}^5\{0,1\}^*$, so that $w = xy$, for some $x \in \{0,1\}^5$ and $y \in \{0,1\}^*$. Thus $w = xy \in \{0,1\}^*$ and $|w| \geq |x| = 5$, showing that $w \in \textbf{Long}$.

Suppose $w \in \textbf{Long}$, so that $w \in \{0,1\}^*$ and $|w| \geq 5$. Then $w = abcdex$, for some $a,b,c,d,e \in \{0,1\}$ and $x \in \{0,1\}^*$. Hence $w = (abcde)x \in \{0,1\}^5\{0,1\}^*$. $\square$

Let the DFA **longDFA** be

$$\mathbf{minAndRen}(\mathbf{regToDFA}(\mathbf{longReg})).$$

An easy calculation shows that $L(\mathbf{longDFA}) = \mathbf{Long}$.

Let **stutterReg** be the regular expression

$$(\mathsf{0}+\mathsf{1})^*(\mathsf{00}+\mathsf{11})(\mathsf{0}+\mathsf{1})^*.$$

Lemma 3.12.10
$L(\mathbf{stutterReg}) = \mathbf{Stutter}$.

Proof. Since $L(\mathbf{stutterReg}) = \{\mathsf{0},\mathsf{1}\}^*\{\mathsf{00},\mathsf{11}\}\{\mathsf{0},\mathsf{1}\}^*$, it will suffice to show that $\{\mathsf{0},\mathsf{1}\}^*\{\mathsf{00},\mathsf{11}\}\{\mathsf{0},\mathsf{1}\}^* = \mathbf{Stutter}$, and this is easy. $\square$

Let **stutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{regToDFA}(\mathbf{stutterReg})).$$

An easy calculation shows that $L(\mathbf{stutterDFA}) = \mathbf{Stutter}$. Let **notStutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{stutterDFA})).$$

Lemma 3.12.11
$L(\mathbf{notStutterDFA}) = \mathbf{NotStutter}$.

Proof. Let M be

$$\mathbf{minAndRen}(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{stutterDFA})).$$

By Lemma 3.12.5(1), we have that

$$\begin{aligned}
L(\mathbf{notStutterDFA}) &= L(M) \\
&= L(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{stutterDFA})) \\
&= L(\mathbf{allStrDFA}) - L(\mathbf{stutterDFA}) \\
&= \{\mathsf{0},\mathsf{1}\}^* - \mathbf{Stutter} \\
&= \mathbf{NotStutter}.
\end{aligned}$$

$\square$

Let **longAndNotStutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{inter}(\mathbf{longDFA}, \mathbf{notStutterDFA})).$$

Lemma 3.12.12
$L(\mathbf{longAndNotStutterDFA}) = \mathbf{LongAndNotStutter}$.

Proof. Let M be

$$\mathbf{minAndRen}(\mathbf{inter}(\mathbf{longDFA}, \mathbf{notStutterDFA})).$$

By Lemma 3.12.5(2), we have that

$$\begin{aligned} L(\mathbf{longAndNotStutterDFA}) &= L(M) \\ &= L(\mathbf{inter}(\mathbf{longDFA}, \mathbf{notStutterDFA})) \\ &= L(\mathbf{longDFA}) \cap L(\mathbf{notStutterDFA}) \\ &= \mathbf{Long} \cap \mathbf{NotStutter} \\ &= \mathbf{LongAndNotStutter}. \end{aligned}$$

□

Because **longAndNotStutterDFA** is an EFA, we can let the EFA **longAndNotStutterEFA** be **longAndNotStutterDFA**. Then $L(\mathbf{longAndNotStutterEFA}) = \mathbf{LongAndNotStutter}$.

Let **someLongNotStutterEFA** be the EFA

$$\begin{aligned} \mathbf{renameStatesCanonically}(\mathbf{concat}(&\mathbf{allStrEFA}, \\ &\mathbf{concat}(\mathbf{longAndNotStutterEFA}, \\ &\quad \mathbf{allStrEFA}))). \end{aligned}$$

Lemma 3.12.13
$L(\mathbf{someLongNotStutterEFA}) = \mathbf{SomeLongNotStutter}$.

Proof. We have that

$$\begin{aligned} L(\mathbf{someLongNotStutterEFA}) &= L(\mathbf{renameStatesCanonically}(M)) \\ &= L(M), \end{aligned}$$

where M is

$$\mathbf{concat}(\mathbf{allStrEFA}, \mathbf{concat}(\mathbf{longAndNotStutterEFA}, \mathbf{allStrEFA})).$$

And, by Lemma 3.12.6, we have that

$$\begin{aligned} L(M) &= L(\mathbf{allStrEFA})\, L(\mathbf{longAndNotStutterEFA})\, L(\mathbf{allStrEFA}) \\ &= \{0,1\}^*\, \mathbf{LongAndNotStutter}\, \{0,1\}^* \\ &= \mathbf{SomeLongNotStutter}. \end{aligned}$$

□

Let **someLongNotStutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{efaToDFA}(\mathbf{someLongNotStutterEFA})).$$

Lemma 3.12.14
$L(\mathbf{someLongNotStutterDFA}) = \mathbf{SomeLongNotStutter}$.

Proof. Follows by an easy calculation. □

Finally, let **allLongStutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{someLongNotStutterDFA})).$$

Lemma 3.12.15
$L(\mathbf{allLongStutterDFA}) = \mathbf{AllLongStutter}$ *and, for all* $N \in \mathbf{DFA}$*, if* $L(N) = \mathbf{AllLongStutter}$*, then* **allLongStutterDFA** *has no more states than* N*.*

Proof. We have that

$$L(\mathbf{allLongStutterDFA}) = L(\mathbf{minAndRen}(M)) = L(M),$$

where M is

$$\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{someLongNotStutterDFA}).$$

Then, by Lemma 3.12.4, we have that

$$\begin{aligned} L(M) &= L(\mathbf{allStrDFA}) - L(\mathbf{someLongNotStutterDFA}) \\ &= \{0,1\}^* - \mathbf{SomeLongNotStutter} \\ &= \mathbf{AllLongStutter}. \end{aligned}$$

Suppose $N \in \mathbf{DFA}$ and $L(N) = \mathbf{AllLongStutter}$. Thus $L(N) = L(M)$, so that **allLongStutterDFA** has no more states than N, by Lemma 3.12.7(4).
□

The preceding lemma tells us that the DFA **allLongStutterDFA** is correct and has as few states as is possible. To find out what it looks like, though, we'll have to use Forlan. First we put the text

```
val regToEFA  = faToEFA  o regToFA
val efaToDFA  = nfaToDFA o efaToNFA
val regToDFA  = efaToDFA o regToEFA
val minAndRen = DFA.renameStatesCanonically o DFA.minimize

val allStrReg = Reg.fromString "(0 + 1)*"
val allStrDFA = minAndRen(regToDFA allStrReg)
val allStrEFA = injDFAToEFA allStrDFA

val longReg =
      Reg.concat(Reg.power(Reg.fromString "0 + 1", 5),
                 Reg.fromString "(0 + 1)*")

val longDFA = minAndRen(regToDFA longReg)

val stutterReg = Reg.fromString "(0 + 1)*(00 + 11)(0 + 1)*"

val stutterDFA = minAndRen(regToDFA stutterReg)

val notStutterDFA =
      minAndRen(DFA.minus(allStrDFA, stutterDFA))

val longAndNotStutterDFA =
      minAndRen(DFA.inter(longDFA, notStutterDFA))

val longAndNotStutterEFA =
      injDFAToEFA longAndNotStutterDFA

val someLongNotStutterEFA' =
      EFA.concat(allStrEFA,
                 EFA.concat(longAndNotStutterEFA,
                            allStrEFA))
val someLongNotStutterEFA  =
      EFA.renameStatesCanonically someLongNotStutterEFA'

val someLongNotStutterDFA =
      minAndRen(efaToDFA someLongNotStutterEFA)

val allLongStutterDFA =
      minAndRen(DFA.minus(allStrDFA, someLongNotStutterDFA))
```

in the file `stutter.sml`. Then, we proceed as follows

```
- use "stutter.sml";
[opening stutter.sml]
val regToEFA = fn : reg -> efa
val efaToDFA = fn : efa -> dfa
val regToDFA = fn : reg -> dfa
val minAndRen = fn : dfa -> dfa
val allStrReg = - : reg
val allStrDFA = - : dfa
val allStrEFA = - : efa
val longReg = - : reg
val longDFA = - : dfa
val stutterReg = - : reg
val stutterDFA = - : dfa
val notStutterDFA = - : dfa
val longAndNotStutterDFA = - : dfa
val longAndNotStutterEFA = - : efa
val someLongNotStutterEFA' = - : efa
val someLongNotStutterEFA = - : efa
val someLongNotStutterDFA = - : dfa
val allLongStutterDFA = - : dfa
val it = () : unit
- DFA.output("", allLongStutterDFA);
{states}
A, B, C, D, E, F, G, H, I, J
{start state}
A
{accepting states}
A, B, C, D, E, F, G, H, I
{transitions}
A, 0 -> B; A, 1 -> C; B, 0 -> B; B, 1 -> E; C, 0 -> D;
C, 1 -> C; D, 0 -> B; D, 1 -> G; E, 0 -> F; E, 1 -> C;
F, 0 -> B; F, 1 -> I; G, 0 -> H; G, 1 -> C; H, 0 -> B;
H, 1 -> J; I, 0 -> J; I, 1 -> C; J, 0 -> J; J, 1 -> J
val it = () : unit
```

Thus, **allLongStutterDFA** is the DFA of Figure 3.2.

3.13 The Pumping Lemma for Regular Languages

In this section we consider techniques for showing that particular languages are not regular. Consider the language

$$L = \{\, 0^n1^n \mid n \in \mathbb{N} \,\} = \{\%, 01, 0011, 000111, \ldots\}.$$

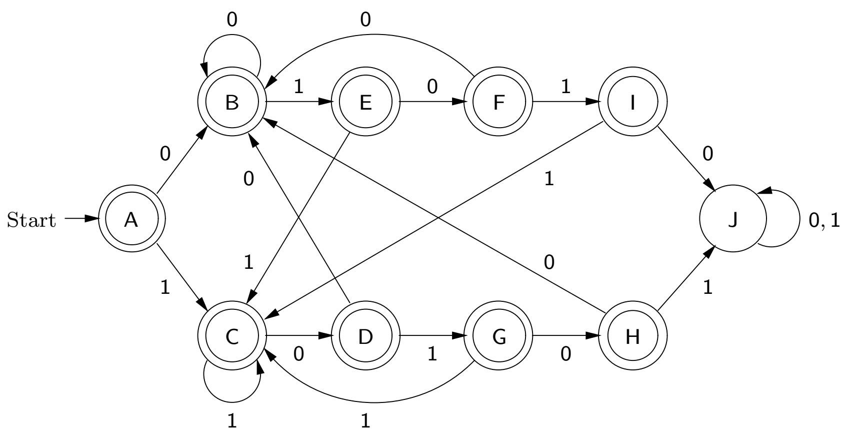


Figure 3.2: DFA Accepting **AllLongStutter**

Intuitively, an automaton would have to have infinitely many states to accept L. A finite automaton won't be able to keep track of how many 0's it has seen so far, and thus won't be able to insist that the correct number of 1's follow. We could turn the preceding ideas into a direct proof that L is not regular. Instead, we will first state a general result, called the Pumping Lemma for regular languages, for proving that languages are non-regular. Next, we will show how the Pumping Lemma can be used to prove that L is non-regular. Finally, we will prove the Pumping Lemma.

Lemma 3.13.1 (Pumping Lemma for Regular Languages)
For all regular languages L, there is a $n \in \mathbb{N}$ such that, for all $z \in \mathbf{Str}$, if $z \in L$ and $|z| \geq n$, then there are $u, v, w \in \mathbf{Str}$ such that $z = uvw$ and

(1) $|uv| \leq n$;

(2) $v \neq \%$; and

(3) $uv^iw \in L$, for all $i \in \mathbb{N}$.

When we use the Pumping Lemma, we can imagine that we are interacting with it. We can give the Pumping Lemma a regular language L, and the lemma will give us back a natural number n such that the property of the lemma holds. We have no control over the value of n. We can then give the lemma a string z that is in L and has at least n symbols. (If L is finite, though, there will be no elements of L with at least n symbols, and so we won't be able to proceed.) The lemma will then break z up into parts u,

v and w in such way that (1)–(3) hold. We have no control over how z is broken up into these parts. (1) says that uv has no more than n symbols. (2) says that v is nonempty. And (3) says that, if we "pump" (duplicate) v as many times as we like, the resulting string will still be in L.

Before proving the Pumping Lemma, let's see how it can be used to prove that $L = \{\, 0^n1^n \mid n \in \mathbb{N}\,\}$ is non-regular.

Proposition 3.13.2
L is not regular.

Proof. Suppose, toward a contradiction, that L is regular. Thus there is an $n \in \mathbb{N}$ with the property of the Pumping Lemma. Suppose $z = 0^n1^n$. Since $z \in L$ and $|z| = 2n \geq n$, it follows that there are $u, v, w \in \mathbf{Str}$ such that $z = uvw$ and properties (1)–(3) of the lemma hold. Since $0^n1^n = z = uvw$, (1) tells us that there are $i, j, k \in \mathbb{N}$ such that

$$u = 0^i, \quad v = 0^j, \quad w = 0^k1^n, \quad i + j + k = n.$$

By (2), we have that $j \geq 1$, and thus that $i + k = n - j < n$. By (3), we have that

$$0^{i+k}1^n = 0^i0^k1^n = uw = u\%w = uv^0w \in L.$$

Thus $i + k = n$—contradiction. Thus L is not regular. $\square$

Now, let's prove the Pumping Lemma.

Proof. Suppose L is a regular language. Thus there is a NFA M such that $L(M) = L$. Let $n = |Q_M|$. Suppose $z \in \mathbf{Str}$, $z \in L$ and $|z| \geq n$. Let $m = |z|$. Thus $1 \leq n \leq |z| = m$. Since $z \in L = L(M)$, there is a valid labeled path for M

$$q_1 \overset{a_1}{\Rightarrow} q_2 \overset{a_2}{\Rightarrow} \cdots q_m \overset{a_m}{\Rightarrow} q_{m+1},$$

that is labeled by z and where $q_1 = s_M$, $q_{m+1} \in A_M$ and $a_i \in \mathbf{Sym}$ for all $1 \leq i \leq m$. Since $|Q_M| = n$, not all of the states $q_1, \ldots, q_{n+1}$ are distinct. Thus, there are $1 \leq i < j \leq n + 1$ such that $q_i = q_j$.

Hence, our path looks like:

$$q_1 \overset{a_1}{\Rightarrow} \cdots q_{i-1} \overset{a_{i-1}}{\Rightarrow} q_i \overset{a_i}{\Rightarrow} \cdots q_{j-1} \overset{a_{j-1}}{\Rightarrow} q_j \overset{a_j}{\Rightarrow} \cdots q_m \overset{a_m}{\Rightarrow} q_{m+1}.$$

Let

$$u = a_1 \cdots a_{i-1}, \quad v = a_i \cdots a_{j-1}, \quad w = a_j \cdots a_m.$$

Then $z = uvw$. Since $|uv| = j - 1$ and $j \leq n + 1$, we have that $|uv| \leq n$. Since $i < j$, we have that $i \leq j - 1$, and thus that $v \neq \%$.

Finally, since

$$q_i \in \Delta(\{q_1\}, u), \quad q_j \in \Delta(\{q_i\}, v), \quad q_{m+1} \in \Delta(\{q_j\}, w)$$

and $q_i = q_j$, we have that

$$q_j \in \Delta(\{q_1\}, u), \quad q_j \in \Delta(\{q_j\}, v), \quad q_{m+1} \in \Delta(\{q_j\}, w).$$

Thus, we have that $q_{m+1} \in \Delta(\{q_1\}, uv^iw)$ for all $i \in \mathbb{N}$. But $q_1 = s_M$ and $q_{m+1} \in A_M$, and thus $uv^iw \in L(M) = L$ for all $i \in \mathbb{N}$. $\square$

Suppose $L' = \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid w$ has an equal number of $\mathsf{0}$'s and $\mathsf{1}$'s$\,\}$. We could show that L' is non-regular using the Pumping Lemma. But we can also prove this result by using some of the closure properties of Section 3.11 plus the fact that $L = \{\, \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \,\}$ is non-regular.

Suppose, toward a contradiction, that L' is regular. It is easy to see that $\{\mathsf{0}\}$ and $\{\mathsf{1}\}$ are regular (e.g., they are denoted by the regular expressions $\mathsf{0}$ and $\mathsf{1}$). Thus, by Theorem 3.11.17, we have that $\{\mathsf{0}\}^*\{\mathsf{1}\}^*$ is regular. Hence, by Theorem 3.11.17 again, it follows that $L = L' \cap \{\mathsf{0}\}^*\{\mathsf{1}\}^*$ is regular—contradiction. Thus L' is non-regular.

As a final example, let X be the least subset of $\{\mathsf{0}, \mathsf{1}\}^*$ such that

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

(2) For all $x, y \in X$, $\mathsf{0}x\mathsf{1}y \in X$.

Let's try to prove that X is non-regular, using the Pumping Lemma. We suppose, toward a contradiction, that X is regular, and give it to the Pumping Lemma, getting back the $n \in \mathbb{N}$ with the property of the lemma, where X has been substituted for L. But then, how do we go about choosing the $z \in \mathbf{Str}$ such that $z \in X$ and $|z| \geq n$? We need to find a string expression *exp* involving the variable n, such that, for all $n \in \mathbb{N}$, *exp* $\in X$ and $|exp| \geq n$.

Because $\% \in X$, we have that $\mathsf{01} = \mathsf{0}\%\mathsf{1}\% \in X$. Thus $\mathsf{0101} = \mathsf{0}\%\mathsf{1}(\mathsf{01}) \in X$. Generalizing, we can easily prove that, for all $n \in \mathbb{N}$, $(\mathsf{01})^n \in X$. Thus we could let $z = (\mathsf{01})^n$. Unfortunately, this won't lead to the needed contradiction, since the Pumping Lemma could break z up into $u = \%$, $v = \mathsf{01}$ and $w = (\mathsf{01})^{n-1}$.

Trying again, we have that $\% \in X$, $\mathsf{01} \in X$ and $\mathsf{0}(\mathsf{01})\mathsf{1}\% = \mathsf{0011} \in X$. Generalizing, it's easy to prove that, for all $n \in \mathbb{N}$, $\mathsf{0}^n\mathsf{1}^n \in X$. Thus, we can let $z = \mathsf{0}^n\mathsf{1}^n$, so that $z \in X$ and $|z| \geq n$. We can then proceed as in the proof that $\{\, \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \,\}$ is non-regular, getting to the point where we learn that $\mathsf{0}^{i+k}\mathsf{1}^n \in X$ and $i + k < n$. But an easy induction on X suffices to show that, for all $w \in X$, w has an equal number of 0's and 1's. Hence $i + k = n$, giving us the needed contradiction.

The Forlan module LP (see Section 3.3) defines a type and several functions that implement the idea behind the pumping lemma:

```
type pumping_division = lp * lp * lp

val checkPumpingDivision  : pumping_division -> unit
val validPumpingDivision  : pumping_division -> bool
val pumpingDivide         : lp -> pumping_division
val strsOfPumpingDivision : pumping_division -> str * str * str
val pump                  : pumping_division * int -> lp
```

A *pumping division* is a triple (lp_1, lp_2, lp_3), where $lp_1, lp_2, lp_3 \in \mathbf{LP}$. We say that a pumping division (lp_1, lp_2, lp_3) is *valid* iff

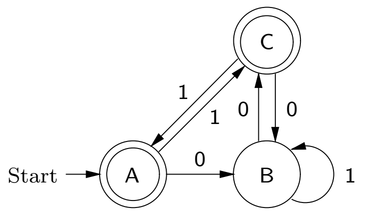
- the end state of lp_1 is equal to the start state of lp_2;
- the start state of lp_2 is equal to the end state of lp_2;
- $|lp_2| \geq 1$;
- the end state of lp_2 is equal to the start state of lp_3.

The function `pumpingDivide` takes in a labeled path lp and tries to divide it into a valid pumping division (lp_1, lp_2, lp_3), while minimizing the value of $|lp_1| + |lp_2|$. It issues an error message if lp has no repetition of states. The function `strsOfPumpingDivision` simply returns the labels of the components of a pumping division. And, the function `pump` takes in a pumping division (lp_1, lp_2, lp_3) and an integer n and returns

$$\mathbf{join}(lp_1, \mathbf{join}(lp', \mathbf{join}(lp_3))),$$

where lp' is the result of joining lp_2 with itself n times (the empty labeled path whose single state is lp_2's start/end state, if $n = 0$). The function issues an error message if the pumping division is invalid or if n is negative.

For example, suppose `dfa` of type `dfa` is bound to the DFA



Then we can proceed as follows:

```
- val lp = DFA.findAcceptingLP dfa (Str.input "");
@ 0011
@ .
val lp = - : lp
- LP.output("", lp);
A, 0 => B, 0 => C, 1 => A, 1 => C
val it = () : unit
- val pd = LP.pumpingDivide lp;
val pd = (-,-,-) : LP.pumping_division
- val (lp1, lp2, lp3) = pd;
val lp1 = - : lp
val lp2 = - : lp
val lp3 = - : lp
- LP.output("", lp1);
A
val it = () : unit
- LP.output("", lp2);
A, 0 => B, 0 => C, 1 => A
val it = () : unit
- LP.output("", lp3);
A, 1 => C
val it = () : unit
- val lp' = LP.pump(pd, 2);
val lp' = - : lp
- LP.output("", lp');
A, 0 => B, 0 => C, 1 => A, 0 => B, 0 => C, 1 => A, 1 => C
val it = () : unit
- Str.output("", LP.label lp');
0010011
val it = () : unit
```

3.14 Applications of Finite Automata and Regular Expressions

In this section we consider two applications of the material from Chapter 3: searching for regular expressions in files; and lexical analysis.

Both of our applications involve processing files whose characters come from some character set, e.g., the ASCII character set. Although not every character in a typical character set will be an element of our set **Sym** of symbols, we can *represent* all the characters of a character set by elements of **Sym**. E.g., we might represent the ASCII characters newline and space by the symbols $\langle\mathsf{newline}\rangle$ and $\langle\mathsf{space}\rangle$, respectively.

In the remainder of this section, we will work with a mostly unspecified alphabet Σ representing some character set. We assume that the symbols $\mathsf{0}$–$\mathsf{9}$, a–z, A–Z, $\langle\mathsf{space}\rangle$ and $\langle\mathsf{newline}\rangle$ are elements of Σ. A *line* is a string consisting of an element of $(\Sigma - \{\langle\mathsf{newline}\rangle\})^*$ followed by $\langle\mathsf{newline}\rangle$; and, a *file* consists of the concatenation of some number of lines.

In what follows, we write:

- [**any**] for the regular expression $a_1 + a_2 + \cdots + a_n$, where $a_1, a_2, \ldots, a_n$ are all of the elements of Σ except $\langle\mathsf{newline}\rangle$, listed in the standard order;
- [**letter**] for the regular expression

$$\mathsf{a} + \mathsf{b} + \cdots + \mathsf{z} + \mathsf{A} + \mathsf{B} + \cdots + \mathsf{Z};$$

- [**digit**] for the regular expression

$$\mathsf{0} + \mathsf{1} + \cdots + \mathsf{9}.$$

First, we consider the problem of searching for instances of regular expressions in files. Given a file and a regular expression α whose alphabet is a subset of $\Sigma - \{\langle\mathsf{newline}\rangle\}$, how can we find all lines of the file with substrings in $L(\alpha)$? (E.g., α might be $\mathsf{a}(\mathsf{b} + \mathsf{c})^*\mathsf{a}$; then we want to find all lines containing two a's, separated by some number of b's and c's.) It will be sufficient to find all lines in the file that are elements of $L(\beta)$, where $\beta = [\mathbf{any}]^*\,\alpha\,[\mathbf{any}]^*\,\langle\mathsf{newline}\rangle$. To do this, we can first translate β to a DFA M with alphabet Σ. For each line w, we simply check whether $\delta_M(s_M, w) \in A_M$, selecting the line if it is.

If the file is short, however, it may be more efficient to convert β to an FA N, and use the algorithm from Section 3.5 to find all lines that are accepted by N.

Now, we turn our attention to lexical analysis. A lexical analyzer is the part of a compiler that groups the characters of a program into lexical items or tokens. The modern approach to specifying a lexical analyzer for a programming language uses regular expressions. E.g., this is the approach taken by the lexical analyzer generator Lex.

A lexical analyzer specification consists of a list of regular expressions $\alpha_1, \alpha_2, \ldots, \alpha_n$, together with a corresponding list of code fragments (in some programming language) $code_1$, $code_2$, ..., $code_n$ that process elements of Σ^*. For example, we might have

$$\begin{aligned}
\alpha_1 &= \langle \mathsf{space} \rangle + \langle \mathsf{newline} \rangle, \\
\alpha_2 &= [\textbf{letter}]\,([\textbf{letter}] + [\textbf{digit}])^*, \\
\alpha_3 &= [\textbf{digit}]\,[\textbf{digit}]^*\,(\% + \mathsf{E}\,[\textbf{digit}]\,[\textbf{digit}]^*), \\
\alpha_4 &= [\textbf{any}].
\end{aligned}$$

The elements of $L(\alpha_1)$, $L(\alpha_2)$ and $L(\alpha_3)$ are whitespace characters, identifiers and numerals, respectively. The code associated with α_4 will probably indicate that an error has occurred.

A lexical analyzer meets such a specification iff it behaves as follows. At each stage of processing its file, the lexical analyzer should consume the *longest* prefix of the remaining input that is in the language denoted by one of the regular expressions. It should then supply the prefix to the code associated with the earliest regular expression whose language contains the prefix. However, if there is no such prefix, or if the prefix is %, then the lexical analyzer should indicate that an error has occurred.

Now, we consider what happens when the file $\mathsf{123Easy}\langle\mathsf{space}\rangle\mathsf{1E2}\langle\mathsf{newline}\rangle$ is processed by a lexical analyzer meeting our example specification.

- The longest prefix of $\mathsf{123Easy}\langle\mathsf{space}\rangle\mathsf{1E2}\langle\mathsf{newline}\rangle$ that is in one of our regular expressions is $\mathsf{123}$. Since this prefix is only in α_3, it is consumed from the input and supplied to $code_3$.

- The remaining input is now $\mathsf{Easy}\langle\mathsf{space}\rangle\mathsf{1E2}\langle\mathsf{newline}\rangle$. The longest prefix of the remaining input that is in one of our regular expressions is Easy. Since this prefix is only in α_2, it is consumed and supplied to $code_2$.

- The remaining input is then $\langle\mathsf{space}\rangle\mathsf{1E2}\langle\mathsf{newline}\rangle$. The longest prefix of the remaining input that is in one of our regular expressions is $\langle\mathsf{space}\rangle$. Since this prefix is only in α_1 and α_4, we consume it from the input and supply it to the code associated with the earlier of these regular expressions: $code_1$.

- The remaining input is then $\mathsf{1E2}\langle\mathsf{newline}\rangle$. The longest prefix of the remaining input that is in one of our regular expressions is $\mathsf{1E2}$. Since this prefix is only in α_3, we consume it from the input and supply it to code_3.

- The remaining input is then $\langle\mathsf{newline}\rangle$. The longest prefix of the remaining input that is in one of our regular expressions is $\langle\mathsf{newline}\rangle$. Since this prefix is only in α_1, we consume it from the input and supply it to the code associated with this expression: code_1.

- The remaining input is now empty, and so the lexical analyzer terminates.

We now give a simple method for generating a lexical analyzer that meets a given specification. (More sophisticated methods are described in compilers courses.) First, we convert the regular expressions $\alpha_1, \ldots, \alpha_n$ into DFAs $M_1, \ldots, M_n$. Next we determine which of the states of the DFAs are dead/live.

Given its remaining input x, the lexical analyzer consumes the next token from x and supplies the token to the appropriate code, as follows.

First, it initializes the following variables to error values:

- a string variable *acc*, which records the longest prefix of the prefix of x that has been processed so far that is accepted by one of the DFAs;

- an integer variable *mach*, which records the smallest i such that $\mathit{acc} \in L(M_i)$;

- a string variable *aft*, consisting of the suffix of x that one gets by removing *acc*.

Then, the lexical analyzer enters its main loop, in which it processes x, symbol by symbol, in *each* of the DFAs, keeping track of what symbols have been processed so far, and what symbols remain to be processed.

- If, after processing a symbol, at least one of the DFAs is in an accepting state, then the lexical analyzer stores the string that has been processed so far in the variable *acc*, stores the index of the first machine to accept this string in the integer variable *mach*, and stores the remaining input in the string variable *aft*. If there is no remaining input, then the lexical analyzer supplies *acc* to code $\mathit{code}_{\mathit{mach}}$ and returns; otherwise it continues.

- If, after processing a symbol, none of the DFAs are in accepting states, but at least one automaton is in a live state (so that, without knowing anything about the remaining input, it's possible that an automaton will again enter an accepting state), then the lexical analyzer leaves *acc*, *mach* and *aft* unchanged. If there is no remaining input, the lexical analyzer supplies *acc* to $code_{mach}$ (it signals an error if *acc* is still set to the error value), resets the remaining input to *aft*, and returns; otherwise, it continues.

- If, after processing a symbol, all of the automata are in dead states (and so could never enter accepting states again, no matter what the remaining input was), the lexical analyzer supplies string *acc* to code $code_{mach}$ (it signals an error if *acc* is still set to the error value), resets the remaining input to *aft*, and returns.

Let's see what happens when the file 123Easy⟨newline⟩ is processed by the lexical analyzer generated from our example specification.

- After processing 1, M_3 and M_4 are in accepting states, and so the lexical analyzer sets *acc* to 1, *mach* to 3, and *aft* to 23Easy⟨newline⟩. It then continues.

- After processing 2, so that 12 has been processed so far, only M_3 is in an accepting state, and so the lexical analyzer sets *acc* to 12, *mach* to 3, and *aft* to 3Easy⟨newline⟩. It then continues.

- After processing 3, so that 123 has been processed so far, only M_3 is in an accepting state, and so the lexical analyzer sets *acc* to 123, *mach* to 3, and *aft* to Easy⟨newline⟩. It then continues.

- After processing E, so that 123E has been processed so far, none of the DFAs are in accepting states, but M_3 is in a live state, since 123E is a prefix of a string that is accepted by M_3. Thus the lexical analyzer continues, but doesn't change *acc*, *mach* or *aft*.

- After processing a, so that 123Ea has been processed so far, all of the machines are in dead states, since 123Ea isn't a prefix of a string that is accepted by one of the DFAs. Thus the lexical analyzer supplies $acc =$ 123 to $code_{mach} = code_3$, and sets the remaining input to $aft =$ Easy⟨newline⟩.

- In subsequent steps, the lexical analyzer extracts Easy from the remaining input, and supplies this string to code $code_2$, and extracts

$\langle$newline$\rangle$ from the remaining input, and supplies this string to code $code_1$.

Chapter 4

Context-free Languages

In this chapter, we study context-free grammars and languages. Context-free grammars are used to describe the syntax of programming languages, i.e., to specify parsers of programming languages.

A language is called context-free iff it is generated by a context-free grammar. It will turn out that the set of all context-free languages is a proper superset of the set of all regular languages. On the other hand, the context-free languages have weaker closure properties than the regular languages, and we won't be able to give algorithms for checking grammar equivalence or minimizing the size of grammars.

4.1 (Context-free) Grammars, Parse Trees and Context-free Languages

In this section, we: say what (context-free) grammars are; use the notion of a parse tree to say what grammars mean; say what it means for a language to be context-free.

A *context-free grammar* (CFG, or just grammar) G consists of:

- a finite set Q_G of symbols (we call the elements of Q_G the *variables* of G);
- an element s_G of Q_G (we call s_G the *start variable* of G);
- a finite subset P_G of $\{\,(q,x) \mid q \in Q_G \text{ and } x \in \mathbf{Str}\,\}$ (we call the elements of P_G the *productions* of G).

In a context where we are only referring to a single CFG, G, we sometimes abbreviate Q_G, s_G and P_G to Q, s and P, respectively. Whenever possible, we will use the mathematical variables p, q and r to name variables. We write **Gram** for the set of all grammars. Since every grammar can be described by a finite sequence of ASCII characters, we have that **Gram** is countably infinite.

As an example, we can define a CFG G (of arithmetic expressions) as follows:

- $Q_G = \{\mathsf{E}\}$;
- $s_G = \mathsf{E}$;
- $P_G = \{(\mathsf{E}, \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E}), (\mathsf{E}, \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E}), (\mathsf{E}, \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle), (\mathsf{E}, \langle\mathsf{id}\rangle)\}$.

E.g., we can read the production $(E, E\langle\mathsf{plus}\rangle E)$ as "an expression can consist of an expression, followed by a $\langle\mathsf{plus}\rangle$ symbol, followed by an expression".

We typically describe a grammar by listing its productions, writing a production (q, x) as $q \rightarrow x$, and grouping productions with identical left-sides into production families. Unless we say otherwise, the grammar's variables are the left-sides of all of its productions, and its start variable is the left-side of its first production. Thus, our grammar G is

$$\begin{aligned}
&\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\\
&\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E},\\
&\mathsf{E} \rightarrow \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle,\\
&\mathsf{E} \rightarrow \langle\mathsf{id}\rangle,
\end{aligned}$$

or

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

The Forlan syntax for grammars is very similar. E.g., here is how our example grammar can be described in Forlan's syntax:

```
{variables}
E
{start variable}
E
{productions}
E -> E<plus>E | E<times>E | <openPar>E<closPar> | <id>
```

Production families are separated by semicolons.

The Forlan module `Gram` defines an abstract type `gram` (in the top-level environment) of grammars as well as a number of functions and constants for processing grammars, including:

```
val input          : string -> gram
val output         : string * gram -> unit
val numVariables   : gram -> int
val numProductions : gram -> int
val equal          : gram * gram -> bool
```

The *alphabet of* a grammar G (**alphabet**(G)) is

$$\{\, a \in \mathbf{Sym} \mid \text{there are } q, x \text{ such that } (q, x) \in P_G \text{ and } a \in \mathbf{alphabet}(x) \,\} - Q_G.$$

I.e., **alphabet**(G) is all of the symbols appearing in the strings of G's productions that aren't variables. For example, the alphabet of our example grammar G is $\{\langle\mathsf{plus}\rangle, \langle\mathsf{times}\rangle, \langle\mathsf{openPar}\rangle, \langle\mathsf{closPar}\rangle, \langle\mathsf{id}\rangle\}$.

The Forlan module `Gram` defines a function

```
val alphabet : gram -> sym set
```

for calculating the alphabet of a grammar. E.g., if `gram` of type `gram` is bound to our example grammar G, then Forlan will behave as follows:

```
- val bs = Gram.alphabet gram;
val bs = - : sym set
- SymSet.output("", bs);
<id>, <plus>, <times>, <closPar>, <openPar>
val it = () : unit
```

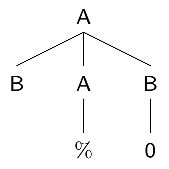
We will explain when strings are generated by grammars using the notion of a parse tree. The set **PT** of *parse trees* is the least subset of $\mathbf{Tree}_{\mathbf{Sym}\cup\{\%\}}$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$, $n \in \mathbb{N}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$, $a(pt_1, \ldots, pt_n) \in \mathbf{PT}$;

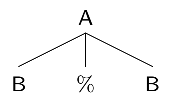
(2) for all $q \in \mathbf{Sym}$, $q(\%) \in \mathbf{PT}$.

Since n is allowed to be 0 in rule (1), for every symbol a, we have that $a()$ is a parse tree, which we normally abbreviate to a. On the other hand, $\% = \%() \notin \mathbf{PT}$. In rule (2), $q(\%)$ abbreviates $q(\%())$. It is easy to see that **PT** is countably infinite.

For example, $\mathsf{A}(\mathsf{B}, \mathsf{A}(\%), \mathsf{B}(\mathsf{0}))$, i.e.,



is a parse tree. On the other hand, although $\mathsf{A}(\mathsf{B}, \%, \mathsf{B})$, i.e.,



is a $(\mathbf{Sym} \cup \{\%\})$-tree, it's not a parse tree, since it can't be formed using rules (1) and (2).

Since the set **PT** of parse trees is defined inductively, it gives rise to an induction principle. The *principle of induction on* **PT** says that

$$\text{for all } pt \in \mathbf{PT},\ P(pt)$$

follows from showing

(1) for all $a \in \mathbf{Sym}$, $n \in \mathbb{N}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$, if

$$P(pt_1), \ldots, P(pt_n),$$

then

$$P(a(pt_1, \ldots, pt_n));$$

(2) for all $q \in \mathbf{Sym}$,

$$P(q(\%)).$$

We define the yield of a parse tree, as follows. The function $\mathbf{yield} \in \mathbf{PT} \to \mathbf{Str}$ is defined by recursion:

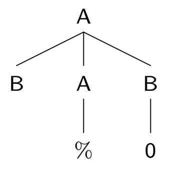
- for all $a \in \mathbf{Sym}$, $\mathbf{yield}(a) = a$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\mathbf{yield}(q(pt_1, \ldots, pt_n)) = \mathbf{yield}(pt_1) \cdots \mathbf{yield}(pt_n);$$

- for all $q \in \mathbf{Sym}$, $\mathbf{yield}(q(\%)) = \%$.

We say that w is the *yield of* pt iff $w = \mathbf{yield}(pt)$.

For example, the yield of



is

$$\mathbf{yield}(\mathsf{B})\,\mathbf{yield}(A(\%))\,\mathbf{yield}(\mathsf{B}(\mathsf{0})) = \mathsf{B}\%\mathbf{yield}(\mathsf{0}) = \mathsf{B}\%\mathsf{0} = \mathsf{B0}.$$

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \{\mathbf{true}, \mathbf{false}\}$ by recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G(a) = a \in \mathbf{alphabet}(G)\ \mathbf{or}\ a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\begin{aligned}&\mathbf{valid}_G(q(pt_1, \ldots, pt_n))\\ &= (q, \mathbf{rootLabel}(pt_1) \cdots \mathbf{rootLabel}(pt_n)) \in P_G\ \mathbf{and}\\ &\quad \mathbf{valid}_G(pt_1)\ \mathbf{and}\ \cdots\ \mathbf{and}\ \mathbf{valid}_G(pt_n);\end{aligned}$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) = (q, \%) \in P_G$.

We say that pt is *valid for* G iff $\mathbf{valid}_G(pt) = \mathbf{true}$. We sometimes abbreviate $\mathbf{valid}_G$ to $\mathbf{valid}$.

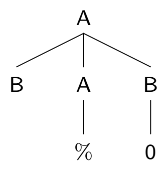
Suppose G is the grammar

$$\begin{aligned}&\mathsf{A} \to \mathsf{BAB} \mid \%,\\ &\mathsf{B} \to \mathsf{0}\end{aligned}$$

(by convention, its variables are A and B and its start variable is A). Let's see why the parse tree $\mathsf{A}(\mathsf{B}, \mathsf{A}(\%), \mathsf{B}(\mathsf{0}))$ is valid for G.

- Since $\mathsf{A} \to \mathsf{BAB} \in P_G$ and the concatenation of the root labels of the sub-trees B, $\mathsf{A}(\%)$ and $\mathsf{B}(\mathsf{0})$ is BAB, the overall tree will be valid for G if these sub-trees are valid for G.
- The parse tree B is valid for G since $\mathsf{B} \in Q_G$.
- Since $\mathsf{A} \to \% \in P_G$, the parse tree $\mathsf{A}(\%)$ is valid for G.
- Since $\mathsf{B} \to \mathsf{0} \in P_G$ and the root label of the sub-tree $\mathsf{0}$ is $\mathsf{0}$, the parse tree $\mathsf{B}(\mathsf{0})$ will be valid for G if the sub-tree $\mathsf{0}$ is valid for G.
- The sub-tree $\mathsf{0}$ is valid for G since $\mathsf{0} \in \mathbf{alphabet}(G)$.

Thus, we have that

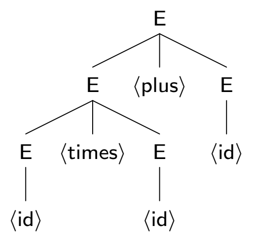


is valid for G.

Suppose G is our grammar of arithmetic expressions

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Then the parse tree



is valid for G.

Now we can say what grammars mean. A string w is *generated by* a grammar G iff $w \in \mathbf{alphabet}(G)^*$ and there is a parse tree pt such that

- pt is valid for G;
- $\mathbf{rootLabel}(pt) = s_G$;
- $\mathbf{yield}(pt) = w$.

The *language generated by* a grammar G ($L(G)$) is

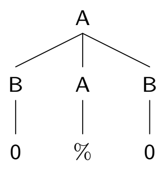
$$\{\, w \in \mathbf{Str} \mid w \text{ is generated by } G \,\}.$$

Proposition 4.1.1
For all grammars G, $\mathbf{alphabet}(L(G)) \subseteq \mathbf{alphabet}(G)$.

Let G be the example grammar

$$\mathsf{A} \rightarrow \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \rightarrow \mathsf{0}.$$

Then $\mathsf{00}$ is generated by G since $\mathsf{00} \in \{\mathsf{0}\}^* = \mathbf{alphabet}(G)^*$ and the parse tree

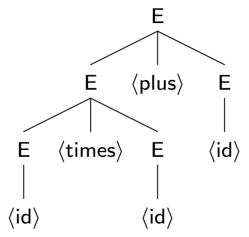


is valid for G, has $s_G = \mathsf{A}$ as its root label, and has $\mathsf{00}$ as its yield.

Suppose G is our grammar of arithmetic expressions:

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Then $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ is generated by G since $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle \in$ **alphabet**$(G)^*$ and the parse tree



is valid for G, has $s_G = \mathsf{E}$ as its root label, and has $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ as its yield.

A language L is *context-free* iff $L = L(G)$ for some $G \in$ **Gram**. We define

$$\begin{aligned}\mathbf{CFLan} &= \{\, L(G) \mid G \in \mathbf{Gram} \,\} \\ &= \{\, L \in \mathbf{Lan} \mid L \text{ is context-free} \,\}.\end{aligned}$$

Since $\{\mathsf{0}^0\}$, $\{\mathsf{0}^1\}$, $\{\mathsf{0}^2\}$, $\ldots$, are all context-free languages, we have that **CFLan** is infinite. But, since **Gram** is countably infinite, it follows that **CFLan** is also countably infinite. Since **Lan** is uncountable, it follows that **CFLan** $\subsetneq$ **Lan**, i.e., there are non-context-free languages. Later, we will see that **RegLan** $\subsetneq$ **CFLan**.

We say that grammars G and H are *equivalent* iff $L(G) = L(H)$. In other words, G and H are equivalent iff G and H generate the same language. We define a relation $\approx$ on **Gram** by: $G \approx H$ iff G and H are equivalent. It is easy to see that $\approx$ is reflexive on **Gram**, symmetric and transitive.

The Forlan module `PT` defines an abstract type `pt` of parse trees (in the top-level environment) along with some functions for processing parse trees:

```
val input     : string -> pt
val output    : string * pt -> unit
val height    : pt -> int
val size      : pt -> int
val equal     : pt * pt -> bool
val rootLabel : pt -> sym
val yield     : pt -> str
```

The Forlan syntax for parse trees is simply the linear syntax that we've been using in this section.

The Forlan module `Gram` also defines the functions

```
val checkPT : gram -> pt -> unit
val validPT : gram -> pt -> bool
```

The function `checkPT` is used to check whether a parse tree is valid for a grammar; if the answer is "no", it explains why not and raises an exception; otherwise it simply returns `()`. The function `validPT` checks whether a parse tree is valid for a grammar, silently returning `true` if it is, and silently returning `false` if it isn't.

Suppose the identifier `gram` of type `gram` is bound to the grammar

$$\begin{aligned}&\mathsf{A} \to \mathsf{BAB} \mid \%,\\&\mathsf{B} \to \mathsf{0}.\end{aligned}$$

And, suppose that the identifier `gram'` of type `gram` is bound to our grammar of arithmetic expressions

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Here are some examples of how we can process parse trees using Forlan:

```
- val pt = PT.input "";
@ A(B, A(%), B(0))
@ .
val pt = - : pt
- Sym.output("", PT.rootLabel pt);
A
val it = () : unit
- Str.output("", PT.yield pt);
B0
val it = () : unit
- Gram.validPT gram pt;
```

```
val it = true : bool
- val pt' = PT.input "";
@ E(E(E(<id>), <times>, E(<id>)), <plus>, E(<id>))
@ .
val pt' = - : pt
- Sym.output("", PT.rootLabel pt');
E
val it = () : unit
- Str.output("", PT.yield pt');
<id><times><id><plus><id>
val it = () : unit
- Gram.validPT gram' pt';
val it = true : bool
- Gram.checkPT gram pt';
invalid production : "E -> E<plus>E"

uncaught exception Error
- Gram.checkPT gram' pt;
invalid production : "A -> BAB"

uncaught exception Error
- PT.input "";
@ A(B,%,B)
@ .
% labels inappropriate node

uncaught exception Error
```

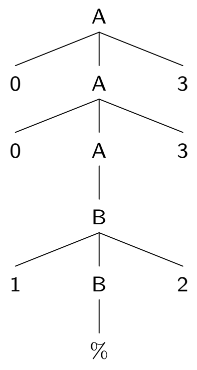
We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$? The key is to think of generating the strings of X from the outside in, in two phases. In the first phase, one generates pairs of 0's and 3's, and, in the second phase, one generates pairs of 1's and 2's. E.g., a string could be formed in the following stages:

$$
\begin{array}{l}
\mathsf{0}\quad\;\;\mathsf{3},\\
\mathsf{00}\;\;\mathsf{33},\\
\mathsf{001233}.
\end{array}
$$

This analysis leads us to the grammar

$$\begin{aligned}
&\mathsf{A} \rightarrow \mathsf{0A3},\\
&\mathsf{A} \rightarrow \mathsf{B},\\
&\mathsf{B} \rightarrow \mathsf{1B2},\\
&\mathsf{B} \rightarrow \%,
\end{aligned}$$

where A corresponds to the first phase, and B to the second phase. For example, here is how the string 001233 may be parsed using G:



4.2 Isomorphism of Grammars

In the section we study the isomorphism of grammars. Suppose G is the grammar with variables A and B, start variable A and productions:

$$\begin{aligned}
&\mathsf{A} \rightarrow \mathsf{0A1} \mid \mathsf{B},\\
&\mathsf{B} \rightarrow \% \mid \mathsf{2A}.
\end{aligned}$$

And, suppose H is the grammar with variables B and A, start variable B and productions:

$$\begin{aligned}
&\mathsf{B} \rightarrow \mathsf{0B1} \mid \mathsf{A},\\
&\mathsf{A} \rightarrow \% \mid \mathsf{2B}.
\end{aligned}$$

H can be formed from G by renaming the variables A and B of G to B and A, respectively. As a result, we say that G and H are isomorphic.

Suppose G is as before, but that H is the grammar with variables 2 and A, start variable 2 and productions:

$$\begin{aligned}
&\mathsf{2} \rightarrow \mathsf{021} \mid \mathsf{A},\\
&\mathsf{A} \rightarrow \% \mid \mathsf{22}.
\end{aligned}$$

Then H can be formed from G by renaming the variables A and B to $\mathsf{2}$ and A, respectively. But, because the symbol $\mathsf{2}$ is in both $\mathbf{alphabet}(G)$ and Q_H, we shouldn't consider G and H to be isomorphic. In fact, G and H generate different languages. A grammar's variables (e.g., A) can't be renamed to elements of the grammar's alphabet (e.g., $\mathsf{2}$).

An *isomorphism* h from a grammar G to a grammar H is a bijection from Q_G to Q_H such that:

- h turns G into H;
- $\mathbf{alphabet}(G) \cap Q_H = \emptyset$, i.e., none of the symbols in G's alphabet are variables of H.

We say that G and H are *isomorphic* iff there is an isomorphism between G and H. As expected, we have that isomorphism implies equivalence.

Let $X = \{\,(G, f) \mid G \in \mathbf{Gram}\,,\ f$ is a bijection from Q_G to some set of symbols, and $\{\,f(q) \mid q \in Q_G\,\} \cap \mathbf{alphabet}(G) \neq \emptyset\,\}$. The function $\mathbf{renameVariables} \in X \to \mathbf{Gram}$ takes in a pair (G, f) and returns the grammar produced from G by renaming G's variables using the bijection f. Then, if G is a grammar and f is a bijection from Q_G to some set of symbols such that $\{\,f(q) \mid q \in Q_G\,\} \cap \mathbf{alphabet}(G) \neq \emptyset$, then $\mathbf{renameVariables}(G, f)$ is isomorphic to G.

The following function is a special case of **renameVariables**. The function $\mathbf{renameVariablesCanonically} \in \mathbf{Gram} \to \mathbf{Gram}$ renames the variables of a grammar G to:

- A, B, etc., when the grammar has no more than 26 variables (the smallest variable of G will be renamed to A, the next smallest one to B, etc.); or
- $\langle 1\rangle$, $\langle 2\rangle$, etc., otherwise.

These variables will actually be surrounded by a uniform number of extra brackets, if this is needed to make the new grammar's variables and the original grammar's alphabet be disjoint.

The Forlan module `Gram` contains the following functions for finding and processing isomorphisms in Forlan:

```
val isomorphism                : gram * gram * sym_rel -> bool
val findIsomorphism            : gram * gram -> sym_rel
val isomorphic                 : gram * gram -> bool
val renameVariables            : gram * sym_rel -> gram
val renameVariablesCanonically : gram -> gram
```

The function `findIsomorphism` is defined using a procedure that is similar to the one used for finding isomorphisms between finite automata, and `isomorphic` is defined using `findIsomorphism`.

Suppose the identifier `gram` of type `gram` is bound to the grammar with variables A and B, start variable A and productions:

$$\begin{aligned}\mathsf{A} &\rightarrow \mathsf{0A1} \mid \mathsf{B},\\ \mathsf{B} &\rightarrow \% \mid \mathsf{2A}.\end{aligned}$$

Suppose the identifier `gram'` of type `gram` is bound to the grammar with variables B and A, start variable B and productions:

$$\begin{aligned}\mathsf{B} &\rightarrow \mathsf{0B1} \mid \mathsf{A},\\ \mathsf{A} &\rightarrow \% \mid \mathsf{2B}.\end{aligned}$$

And, suppose the identifier `gram''` of type `gram` is bound to the grammar with variables $\mathsf{2}$ and A, start variable $\mathsf{2}$ and productions:

$$\begin{aligned}\mathsf{2} &\rightarrow \mathsf{021} \mid \mathsf{A},\\ \mathsf{A} &\rightarrow \% \mid \mathsf{22}.\end{aligned}$$

Here are some examples of how the above functions can be used:

```
- val rel = Gram.findIsomorphism(gram, gram');
val rel = - : sym_rel
- SymRel.output("", rel);
(A, B), (B, A)
val it = () : unit
- Gram.isomorphism(gram, gram', rel);
val it = true : bool
- Gram.isomorphic(gram, gram'');
val it = false : bool
- Gram.isomorphic(gram', gram'');
val it = false : bool
```

4.3 A Parsing Algorithm

In this section, we consider a simple, fairly inefficient parsing algorithm that works for all context-free grammars. Compilers courses cover efficient algorithms that work for various subsets of the context free grammars. The parsing algorithm takes in a grammar G and a string w, and attempts to find a minimally-sized parse tree pt such that:

- pt is valid for G;
- $\textbf{rootLabel}(pt) = s_G$;
- $\textbf{yield}(pt) = w$.

If there is no such pt, then the algorithm reports failure.

Let's start by considering an algorithm for checking whether $w \in L(G)$, for a string w and grammar G. Let $A = Q_G \cup \textbf{alphabet}(w)$ and $B = \{\, x \in \textbf{Str} \mid x \text{ is a substring of } w \,\}$. We generate the least subset X of $A \times B$ such that:

- For all $a \in \textbf{alphabet}(w)$, $(a, a) \in X$;
- For all $q \in Q_G$, if $q \to \% \in P_G$, then $(q, \%) \in X$;
- For all $q \in Q_G$, $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in A$ and $x_1, \ldots, x_n \in B$, if
  - $q \to a_1 \cdots a_n \in P_G$,
  - for all $1 \leq i \leq n$, $(a_i, x_i) \in X$, and
  - $x_1 \cdots x_n \in B$,

  then $(q, x_1 \cdots x_n) \in X$.

Since $A \times B$ is finite, this process terminates.

For example, let G be the grammar

$$\begin{aligned}
\mathsf{A} &\to \mathsf{BC} \mid \mathsf{CD}, \\
\mathsf{B} &\to \mathsf{0} \mid \mathsf{CB}, \\
\mathsf{C} &\to \mathsf{1} \mid \mathsf{DD}, \\
\mathsf{D} &\to \mathsf{0} \mid \mathsf{BC},
\end{aligned}$$

and let $w = \mathsf{0010}$. We have that:

- $(\mathsf{0}, \mathsf{0}) \in X$;
- $(\mathsf{1}, \mathsf{1}) \in X$;
- $(\mathsf{B}, \mathsf{0}) \in X$, since $\mathsf{B} \to \mathsf{0} \in P_G$, $(\mathsf{0}, \mathsf{0}) \in X$ and $\mathsf{0} \in B$;
- $(\mathsf{C}, \mathsf{1}) \in X$, since $\mathsf{C} \to \mathsf{1} \in P_G$, $(\mathsf{1}, \mathsf{1}) \in X$ and $\mathsf{1} \in B$;
- $(\mathsf{D}, \mathsf{0}) \in X$, since $\mathsf{D} \to \mathsf{0} \in P_G$, $(\mathsf{0}, \mathsf{0}) \in X$ and $\mathsf{0} \in B$;
- $(\mathsf{A}, \mathsf{01}) \in X$, since $\mathsf{A} \to \mathsf{BC} \in P_G$, $(\mathsf{B}, \mathsf{0}) \in X$, $(\mathsf{C}, \mathsf{1}) \in X$ and $\mathsf{01} \in B$;

- $(\mathsf{A}, \mathsf{10}) \in X$, since $\mathsf{A} \to \mathsf{CD} \in P_G$, $(\mathsf{C}, \mathsf{1}) \in X$, $(\mathsf{D}, \mathsf{0}) \in X$ and $\mathsf{10} \in B$;
- $(\mathsf{B}, \mathsf{10}) \in X$, since $\mathsf{B} \to \mathsf{CB} \in P_G$, $(\mathsf{C}, \mathsf{1}) \in X$, $(\mathsf{B}, \mathsf{0}) \in X$ and $\mathsf{10} \in B$;
- $(\mathsf{C}, \mathsf{00}) \in X$, since $\mathsf{C} \to \mathsf{DD} \in P_G$, $(\mathsf{D}, \mathsf{0}) \in X$, $(\mathsf{D}, \mathsf{0}) \in X$ and $\mathsf{00} \in B$;
- $(\mathsf{D}, \mathsf{01}) \in X$, since $\mathsf{D} \to \mathsf{BC} \in P_G$, $(\mathsf{B}, \mathsf{0}) \in X$, $(\mathsf{C}, \mathsf{1}) \in X$ and $\mathsf{01} \in B$;
- $(\mathsf{C}, \mathsf{001}) \in X$, since $\mathsf{C} \to \mathsf{DD} \in P_G$, $(\mathsf{D}, \mathsf{0}) \in X$, $(\mathsf{D}, \mathsf{01}) \in X$ and $\mathsf{0}(\mathsf{01}) \in B$;
- $(\mathsf{C}, \mathsf{010}) \in X$, since $\mathsf{C} \to \mathsf{DD} \in P_G$, $(\mathsf{D}, \mathsf{01}) \in X$, $(\mathsf{D}, \mathsf{0}) \in X$ and $(\mathsf{01})\mathsf{0} \in B$;
- $(\mathsf{A}, \mathsf{0010}) \in X$, since $\mathsf{A} \to \mathsf{BC} \in P_G$, $(\mathsf{B}, \mathsf{0}) \in X$, $(\mathsf{C}, \mathsf{010}) \in X$ and $\mathsf{0}(\mathsf{010}) \in B$;
- $(\mathsf{B}, \mathsf{0010}) \in X$, since $\mathsf{B} \to \mathsf{CB} \in P_G$, $(\mathsf{C}, \mathsf{00}) \in X$, $(\mathsf{B}, \mathsf{10}) \in X$ and $(\mathsf{00})(\mathsf{10}) \in B$;
- $(\mathsf{D}, \mathsf{0010}) \in X$, since $\mathsf{D} \to \mathsf{BC} \in P_G$, $(\mathsf{B}, \mathsf{0}) \in X$, $(\mathsf{C}, \mathsf{010}) \in X$ and $\mathsf{0}(\mathsf{010}) \in B$;
- Nothing more can be added to X.

The following lemmas concerning X are easy to prove:

Lemma 4.3.1
For all $(a, x) \in X$, there is a $pt \in \mathbf{PT}$ such that

- *pt is valid for G,*
- **rootLabel**$(pt) = a$,
- **yield**$(pt) = x$.

Lemma 4.3.2
For all $a \in A$ and $x \in B$, if there is a $pt \in \mathbf{PT}$ such that

- *pt is valid for G,*
- **rootLabel**$(pt) = a$,
- **yield**$(pt) = x$,

then $(a, x) \in X$.

Thus, to determine if $w \in L(G)$, we just have to check whether $(s_G, w) \in X$. In the case of our example grammar, we have that $w = \mathsf{0010} \in L(G)$, since $(\mathsf{A}, \mathsf{0010}) \in X$.

If we label each element (a, x) of our set X with a parse tree pt such that

- pt is valid for G,
- $\textbf{rootLabel}(pt) = a$,
- $\textbf{yield}(pt) = x$,

then we can return the parse tree labeling (s_G, w), if this pair is in X. Otherwise, we report failure.

With some more work, we can arrange that the parse trees returned by our parsing algorithm are minimally-sized, and this is what the official version of our parsing algorithm guarantees. This goal is a little tricky to achieve, since some pairs will first be labeled by parse trees that aren't minimally sized.

The Forlan module `Gram` defines a function

```
val parseStr : gram -> str -> pt
```

that implements our algorithm for parsing strings according to grammars.

Suppose that `gram` of type `gram` is bound to the grammar

$$\begin{aligned}\mathsf{A} &\to \mathsf{BC} \mid \mathsf{CD},\\ \mathsf{B} &\to \mathsf{0} \mid \mathsf{CB},\\ \mathsf{C} &\to \mathsf{1} \mid \mathsf{DD},\\ \mathsf{D} &\to \mathsf{0} \mid \mathsf{BC}.\end{aligned}$$

We can attempt to parse some strings according to this grammar, as follows.

```
- fun test s =
=       PT.output("",
=                 Gram.parseStr gram (Str.fromString s));
val test = fn : string -> unit
- test "0010";
A(B(0), C(D(B(0), C(1)), D(0)))
val it = () : unit
- test "0100";
A(C(D(B(0), C(1)), D(0)), D(0))
```

```
val it = () : unit
- test "0101";
no such parse exists

uncaught exception Error
```

4.4 Simplification of Grammars

In this section, we say what it means for a grammar to be simplified, give a simplification algorithm for grammars, and see how to use this algorithm in Forlan.

Suppose G is the grammar

$$\begin{aligned}
&\mathsf{A} \rightarrow \mathsf{BB1},\\
&\mathsf{B} \rightarrow \mathsf{0} \mid \mathsf{A} \mid \mathsf{CD},\\
&\mathsf{C} \rightarrow \mathsf{12},\\
&\mathsf{D} \rightarrow \mathsf{1D2}.
\end{aligned}$$

There are two things that are odd about this grammar. First, the there isn't a valid parse tree for G that starts at D and whose yield is in $\mathbf{alphabet}(G)^* = \{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$. Second, there is no valid parse tree that starts at G's start variable A, has a yield that is in $\{\mathsf{0},\mathsf{1},\mathsf{2}\}^*$, and makes use of C. As a result, we will say that both C and D are "useless".

Suppose G is a grammar. We say that a variable q of G is:

- *reachable* iff there is a parse tree pt such that pt is valid for G, $\mathbf{rootLabel}(pt) = s_G$ and q is one of the leaves of pt;
- *generating* iff there is a parse tree pt such that pt is valid for G, $\mathbf{rootLabel}(pt) = q$ and $\mathbf{yield}(pt) \in \mathbf{alphabet}(G)^*$;
- *useful* iff there is a parse tree pt such that pt is valid for G, $\mathbf{rootLabel}(pt) = s_G$, $\mathbf{yield}(pt) \in \mathbf{alphabet}(G)^*$, and q appears in pt.

Thus every useful variable is both reachable and generating, but the converse is false. For example, the variable C of our example grammar is reachable and generating, but isn't useful.

A grammar G is *simplified* iff either

- all of G's variables are useful; or

- G has a single variable and no productions.

E.g., the grammar with variable A, start variable A and no productions is simplified, even though A is useless. Of course, this grammar generates the empty language.

To simplify a grammar G, we proceed as follows.

- First, we determine which variables of G are generating. If s_G isn't one of these variables, then we return the grammar with variable s_G and no productions.

- Next, we turn G into a grammar G' by deleting all non-generating variables, and deleting all productions involving such variables.

- Then, we determine which variables of G' are reachable.

- Finally, we turn G' into a grammar G'' by deleting all non-reachable variables, and deleting all productions involving such variables.

Suppose G, once again, is the grammar

$$\begin{aligned}
&\mathsf{A} \to \mathsf{BB1},\\
&\mathsf{B} \to \mathsf{0} \mid \mathsf{A} \mid \mathsf{CD},\\
&\mathsf{C} \to \mathsf{12},\\
&\mathsf{D} \to \mathsf{1D2}.
\end{aligned}$$

Here is what happens if we apply our simplification algorithm to G.

First, we determine which variables are generating. Clearly B and C are. And, since B is, it follows that A is, because of the production $\mathsf{A} \to \mathsf{BB1}$. (If this production had been $\mathsf{A} \to \mathsf{BD1}$, we wouldn't have added A to our set.) Thus, we form G' from G by deleting the variable D, yielding the grammar

$$\begin{aligned}
&\mathsf{A} \to \mathsf{BB1},\\
&\mathsf{B} \to \mathsf{0} \mid \mathsf{A},\\
&\mathsf{C} \to \mathsf{12}.
\end{aligned}$$

Next, we determine which variables of G' are reachable. Clearly A is, and thus B is, because of the production $\mathsf{A} \to \mathsf{BB1}$. Note that, if we carried out the two stages of our simplification algorithm in the other order, then C and its productions would never be deleted. Finally, we form G'' from G' by deleting the variable C, yielding the grammar

$$\begin{aligned}
&\mathsf{A} \to \mathsf{BB1},\\
&\mathsf{B} \to \mathsf{0} \mid \mathsf{A}.
\end{aligned}$$

We define a function **simplify** $\in$ **Gram** $\rightarrow$ **Gram** by: for all $G \in$ **Gram**, **simplify**(G) is the result of running the above algorithm on G.

Theorem 4.4.1
For all $G \in$ **Gram***:*

(1) **simplify**(G) *is simplified;*

(2) **simplify**$(G) \approx G$*;*

(3) **alphabet**(**simplify**(G)) $\subseteq$ **alphabet**(G).

The Forlan module `Gram` defines the function

```
val simplify : gram -> gram
```

for simplifying grammars.

Suppose `gram` of type `gram` is bound to the grammar

$$\begin{aligned}
&\mathsf{A} \rightarrow \mathsf{BB1},\\
&\mathsf{B} \rightarrow \mathsf{0} \mid \mathsf{A} \mid \mathsf{CD},\\
&\mathsf{C} \rightarrow \mathsf{12},\\
&\mathsf{D} \rightarrow \mathsf{1D2}.
\end{aligned}$$

We can simplify our grammar as follows:

```
- val gram' = Gram.simplify gram;
val gram' = - : gram
- Gram.output("", gram');
{variables}
A, B
{start variable}
A
{productions}
A -> BB1; B -> 0 | A
val it = () : unit
```

4.5 Proving the Correctness of Grammars

In this section, we consider a technique for proving the correctness of grammars. We begin with a useful definition.

Suppose G is a grammar and $a \in Q_G \cup$ **alphabet**(G). Then $\Pi_{G,a} =$ $\{\, w \in$ **alphabet**$(G)^* \mid$ there is a $pt \in$ **PT** such that pt is valid for

G, $\mathbf{rootLabel}(pt) = a$ and $\mathbf{yield}(pt) = w\,\}$. If it's clear which grammar we are talking about, we sometimes abbreviate $\Pi_{G,a}$ to Π_a.

For example, if G is the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{0A3}, \qquad &&\mathsf{A} \to \mathsf{B}, \\ &\mathsf{B} \to \mathsf{1B2}, \qquad &&\mathsf{B} \to \%, \end{aligned}$$

then $\Pi_{\mathsf{0}} = \{\mathsf{0}\}$, $\Pi_{\mathsf{1}} = \{\mathsf{1}\}$, $\Pi_{\mathsf{2}} = \{\mathsf{2}\}$, $\Pi_{\mathsf{3}} = \{\mathsf{3}\}$, $\Pi_{\mathsf{A}} = \{\, \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \mid n, m \in \mathbb{N}\,\} = L(G)$ and $\Pi_{\mathsf{B}} = \{\, \mathsf{1}^m\mathsf{2}^m \mid m \in \mathbb{N}\,\}$.

Proposition 4.5.1
Suppose G is a grammar.

(1) For all $a \in \mathbf{alphabet}(G)$, $\Pi_{G,a} = \{a\}$.

(2) For all $q \in Q_G$, $\Pi_{G,q} = \{\, w_1 \cdots w_n \mid$ there are $a_1, \ldots, a_n \in \mathbf{Sym}$ such that $q \to a_1, \ldots, a_n \in P_G$ and $w_1 \in \Pi_{G,a_1}, \ldots, w_n \in \Pi_{G,a_n}\,\}$.

Suppose G is a grammar, and $\mathsf{A}\to\%$ and $\mathsf{A}\to\mathsf{0B1C}$ are productions of G, where $\mathsf{B}, \mathsf{C} \in Q_G$ and $\mathsf{0}, \mathsf{1} \in \mathbf{alphabet}(G)$. By Part (2) of the proposition, in the case when $n = 0$, we have that, since $\mathsf{A} \to \% \in P_G$, then $\% \in \Pi_{\mathsf{A}}$. Suppose $w_2 \in \Pi_{\mathsf{B}}$ and $w_4 \in \Pi_{\mathsf{C}}$. By Part (1) of the proposition, we have that $\mathsf{0} \in \Pi_{\mathsf{0}}$ and $\mathsf{1} \in \Pi_{\mathsf{1}}$. Thus, since $\mathsf{A}\to\mathsf{0B1C} \in P_G$, we have that $\mathsf{0}w_2\mathsf{1}w_4 \in \Pi_{\mathsf{A}}$.

If a grammar has no productions of the form $q \to r$, for a variable r, we could use strong string induction and Proposition 4.5.1 to prove it correct. Because this technique doesn't work in the general case, we will introduce an induction principle that will work in general.

Suppose G is a grammar and that, for all $a \in Q_G \cup \mathbf{alphabet}(G)$, $P_a(w)$ is a property of a string $w \in \Pi_{G,a}$. The *principle of induction on* Π says that

$$\text{for all } a \in Q_G \cup \mathbf{alphabet}(G), \text{ for all } w \in \Pi_{G,a},\ P_a(w)$$

follows from showing

(1) for all $a \in \mathbf{alphabet}(G)$, $P_a(a)$;

(2) for all $q \in Q_G$, $n \in \mathbb{N}$ and $a_1, \ldots, a_n \in \mathbf{Sym}$,
if $q \to a_1 \cdots a_n \in P_G$, then

$$\begin{aligned} &\text{for all } \ w_1 \in \Pi_{G,a_1},\ \ldots,\ w_n \in \Pi_{G,a_n}, \\ &\text{if } \ (\dagger)\ P_{a_1}(w_1),\ \ldots,\ P_{a_n}(w_n), \\ &\text{then } \ P_q(w_1 \cdots w_n). \end{aligned}$$

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

If $a \in \textbf{alphabet}(G)$, then $\Pi_{G,a} = \{a\}$. We will only apply the property $P_a(\cdot)$ to elements of $\Pi_{G,a}$, i.e., to a, and Part (1) requires that $P_a(a)$ holds. Thus, when applying our induction principle, we can implicitly assume that $P_a(w)$ says "$w = a$". Given this assumption, we won't have to explicitly prove Part (1).

Furthermore, when proving Part (2), given a symbol $a_i \in \textbf{alphabet}(G)$, we will have that $w_i = a_i$, and it will be unnecessary to assume that $P_{a_i}(a_i)$, since this will always be true. For example, given the production $A \rightarrow \mathsf{0B1C}$, where $\mathsf{B}, \mathsf{C} \in Q_G$ and $\mathsf{0}, \mathsf{1} \in \textbf{alphabet}(G)$, we will proceed as follows. We will assume that $w_2 \in \Pi_\mathsf{B}$ and $w_4 \in \Pi_\mathsf{C}$, and that the inductive hypothesis holds: $P_\mathsf{B}(w_2)$ and $P_\mathsf{C}(w_4)$. Then, we will prove that $P_\mathsf{A}(\mathsf{0}w_2\mathsf{1}w_4)$. Of course, we could use the variables x and y instead of w_2 and w_4.

Now, let's do an example correctness proof. Let G be the grammar

$$\begin{aligned} &\mathsf{A} \rightarrow \mathsf{0A3}, \qquad &&\mathsf{A} \rightarrow \mathsf{B}, \\ &\mathsf{B} \rightarrow \mathsf{1B2}, \qquad &&\mathsf{B} \rightarrow \%, \end{aligned}$$

Let

$$\begin{aligned} X &= \{\, \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \mid n, m \in \mathbb{N} \,\}, \\ Y &= \{\, \mathsf{1}^m\mathsf{2}^m \mid m \in \mathbb{N} \,\}. \end{aligned}$$

To prove that $L(G) = X$, it will suffice to show that $X \subseteq L(G) \subseteq X$.

Lemma 4.5.2
$X \subseteq L(G)$.

Proof. First, we prove that $Y \subseteq \Pi_\mathsf{B}$. It will suffice to show that, for all $m \in \mathbb{N}$, $\mathsf{1}^m\mathsf{2}^m \in \Pi_\mathsf{B}$. We proceed by mathematical induction.

(Basis Step) Because $\mathsf{B} \rightarrow \% \in P$, we have that $\mathsf{1}^0\mathsf{2}^0 = \%\% = \% \in \Pi_\mathsf{B}$, by Proposition 4.5.1.

(Inductive Step) Suppose $m \in \mathbb{N}$, and assume the inductive hypothesis: $\mathsf{1}^m\mathsf{2}^m \in \Pi_\mathsf{B}$. Because $\mathsf{B} \rightarrow \mathsf{1B2} \in P$, it follows that $\mathsf{1}^{m+1}\mathsf{2}^{m+1} = \mathsf{1}(\mathsf{1}^m\mathsf{2}^m)\mathsf{2} \in \Pi_\mathsf{B}$, by Proposition 4.5.1.

Next, we prove that $X \subseteq \Pi_\mathsf{A} = L(G)$. Suppose $m \in \mathbb{N}$. It will suffice to show that, for all $n, m \in \mathbb{N}$, $\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \in \Pi_\mathsf{A}$. Suppose $m \in \mathbb{N}$. It will suffice to show that, for all $n \in \mathbb{N}$, $\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \in \Pi_\mathsf{A}$. We proceed by mathematical induction.

(Basis Step) Because $Y \subseteq \Pi_\mathsf{B}$, we have that $\mathsf{1}^m\mathsf{2}^m \in \Pi_\mathsf{B}$. Then, since $\mathsf{A} \rightarrow \mathsf{B} \in P$, we have that $\mathsf{0}^0\mathsf{1}^m\mathsf{2}^m\mathsf{3}^0 = \%\mathsf{1}^m\mathsf{2}^m\% = \mathsf{1}^m\mathsf{2}^m \in \Pi_\mathsf{A}$, by Proposition 4.5.1.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \in \Pi_\mathsf{A}$. Because $\mathsf{A} \to \mathsf{0A3} \in P$, it follows that $\mathsf{0}^{n+1}\mathsf{1}^m\mathsf{2}^m\mathsf{3}^{n+1} = \mathsf{0}(\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n)\mathsf{3} \in \Pi_\mathsf{A}$, by Proposition 4.5.1. □

Lemma 4.5.3
$L(G) \subseteq X$.

Proof. Since $\Pi_\mathsf{A} = L(G)$, it will suffice to show that

(A) for all $w \in \Pi_\mathsf{A}$, $w \in X$;

(B) for all $w \in \Pi_\mathsf{B}$, $w \in Y$.

We proceed by induction on Π.

Formally, this means that we let the properties $P_\mathsf{A}(w)$ and $P_\mathsf{B}(w)$ be "$w \in X$" and "$w \in Y$", respectively, and then use the induction principle to prove that, for all $a \in Q_G \cup \mathbf{alphabet}(G)$, for all $w \in \Pi_a$, $P_a(w)$. But we will actually work more informally.

There are four productions to consider.

- ($\mathsf{A} \to \mathsf{0A3}$) Suppose $w \in \Pi_\mathsf{A}$, and assume the inductive hypothesis: $w \in X$. We must show that $\mathsf{0}w\mathsf{3} \in X$. Because $w \in X$, we have that $w = \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n$, for some $n, m \in \mathbb{N}$. Thus $\mathsf{0}w\mathsf{3} = \mathsf{0}(\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n)\mathsf{3} = \mathsf{0}^{n+1}\mathsf{1}^m\mathsf{2}^m\mathsf{3}^{n+1} \in X$.

- ($\mathsf{A} \to \mathsf{B}$) Suppose $w \in \Pi_\mathsf{B}$, and assume the inductive hypothesis: $w \in Y$. We must show that $w \in X$. Because $w \in Y$, we have that $w = \mathsf{1}^m\mathsf{2}^m$, for some $m \in \mathbb{N}$. Thus $w = \%w\% = \mathsf{0}^0\mathsf{1}^m\mathsf{2}^m\mathsf{3}^0 \in X$.

- ($\mathsf{B} \to \mathsf{1B2}$) Suppose $w \in \Pi_\mathsf{B}$, and assume the inductive hypothesis: $w \in Y$. We must show that $\mathsf{1}w\mathsf{2} \in Y$. Because $w \in Y$, we have that $w = \mathsf{1}^m\mathsf{2}^m$, for some $m \in \mathbb{N}$. Thus $\mathsf{1}w\mathsf{2} = \mathsf{1}(\mathsf{1}^m\mathsf{2}^m)\mathsf{2} = \mathsf{1}^{m+1}\mathsf{2}^{m+1} \in Y$.

- ($\mathsf{B} \to \%$) We must show that $\% \in Y$, and this follows since $\% = \%\% = \mathsf{1}^0\mathsf{2}^0 \in Y$.

□

Proposition 4.5.4
$L(G) = X$.

Proof. Follows from Lemmas 4.5.2 and 4.5.3. □

If we look at the proofs of Lemmas 4.5.2 and 4.5.3, we can conclude that, for all $w \in \mathbf{Str}$:

(A) $w \in \Pi_{\mathsf{A}}$ iff $w \in X$; and

(B) $w \in \Pi_{\mathsf{B}}$ iff $w \in Y$.

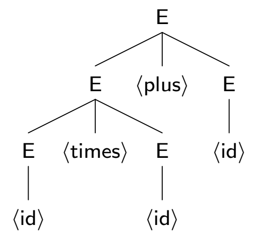
4.6 Ambiguity of Grammars

In this section, we say what it means for a grammar to be ambiguous. We also consider a straightforward method for disambiguating some commonly occurring grammars.

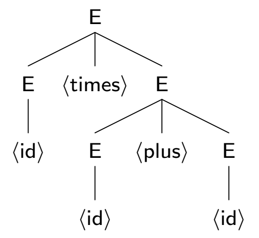
Suppose G is our grammar of arithmetic expressions:

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

There multiple ways of parsing the string $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ according to this grammar:



(pt_1)



(pt_2)

In pt_1, multiplication has higher precedence than addition; in pt_2, the situation is reversed. Because there are multiple ways of parsing this string, we say that our grammar is "ambiguous". A grammar G is *ambiguous* iff there is a $w \in \mathbf{alphabet}(G)^*$ such that w is the yield of multiple valid parse trees for G whose root labels are s_G; otherwise, G is *unambiguous*.

Not every ambiguous grammar can be turned into an equivalent unambiguous one. However, we can use a simple technique to disambiguate our grammar of arithmetic expressions. Since there are two binary operators in our language of arithmetic expressions, we have to decide:

- whether multiplication has higher or lower precedence than addition;
- whether multiplication and addition are left or right associative.

As usual, we'll make multiplication have higher precedence than addition, and make both multiplication and addition be left associative.

As a first step towards disambiguating our grammar, we can form a new grammar with the three variables: E (expressions), T (terms) and F (factors), start variable E and productions:

$$\begin{aligned}
&\mathsf{E} \rightarrow \mathsf{T} \mid \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E}, \\
&\mathsf{T} \rightarrow \mathsf{F} \mid \mathsf{T}\langle\mathsf{times}\rangle\mathsf{T}, \\
&\mathsf{F} \rightarrow \langle\mathsf{id}\rangle \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle.
\end{aligned}$$

The idea is that the lowest precedence operator "lives" at the highest level of the grammar, that the highest precedence operator lives at the middle level of the grammar, and that the basic expressions, including the parenthesized expressions, live at the lowest level of the grammar.

Now, there is only one way to parse the string $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$, since, if we begin by using the production $\mathsf{E} \rightarrow \mathsf{T}$, our yield will only include a $\langle\mathsf{plus}\rangle$ if this symbol occurs within parentheses. If we had more levels of precedence in our language, we would simply add more levels to our grammar.

On the other hand, there are still two ways of parsing the string $\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$: with left associativity or right associativity. To finish disambiguating our grammar, we must break the symmetry of the right-sides of the productions

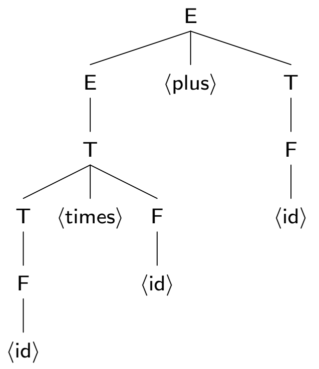
$$\begin{aligned}
&\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E}, \\
&\mathsf{T} \rightarrow \mathsf{T}\langle\mathsf{times}\rangle\mathsf{T},
\end{aligned}$$

turning one of the E's into T, and one of the T's into F. To make our operators be left associative, we must change the second E to T, and the second T to F; right associativity would result from making the opposite choices.

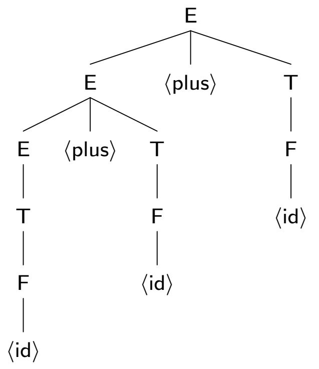
Thus, our unambiguous grammar of arithmetic expressions is

$$\begin{aligned}
&\mathsf{E} \rightarrow \mathsf{T} \mid \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{T}, \\
&\mathsf{T} \rightarrow \mathsf{F} \mid \mathsf{T}\langle\mathsf{times}\rangle\mathsf{F}, \\
&\mathsf{F} \rightarrow \langle\mathsf{id}\rangle \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle.
\end{aligned}$$

It can be proved that this grammar is indeed unambiguous, and that it is equivalent to the original grammar. Now, the only parse of $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ is



And, the only parse of $\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ is



4.7 Closure Properties of Context-free Languages

In this section, we consider several operations on grammars, including union, concatenation and closure operations. As a result, we will have that the context-free languages are closed under union, concatenation and closure. Later, we will see that it is impossible to define intersection, complementation, and set difference operations on grammars. As a result, the context-free languages won't be closed under these operations.

First, we consider some basic grammars and operations on grammars. The grammar with variable A and production $\mathsf{A} \rightarrow \%$ generates the language $\{\%\}$. The grammar with variable A and no productions generates the language $\emptyset$. If w is a string, then the grammar with variable A and production $\mathsf{A} \rightarrow w$ generates the language $\{w\}$. Actually, we must be careful to chose a variable that doesn't occur in w.

Next, we define union, concatenation and closure operations on grammars. Suppose G_1 and G_2 are grammars. We can define a grammar H

such that $L(H) = L(G_1) \cup L(G_2)$ by unioning together the variables and productions of G_1 and G_2, and adding a new start variable q, along with productions

$$q \rightarrow s_{G_1} \mid s_{G_2}.$$

Unfortunately, for the above to be valid, we need to know that:

- $Q_{G_1} \cap Q_{G_2} = \emptyset$ and $q \notin Q_{G_1} \cup Q_{G_2}$;
- $\mathbf{alphabet}(G_1) \cap Q_{G_2} = \emptyset$, $\mathbf{alphabet}(G_2) \cap Q_{G_1} = \emptyset$ and $q \notin \mathbf{alphabet}(G_1) \cup \mathbf{alphabet}(G_2)$.

Our official union operation for grammars renames the variables of G_1 and G_2, and chooses the start variable q, in a uniform way that makes the preceding properties hold. To keep things simple, when talking about the concatenation and closure operations on grammars, we'll just assume that conflicts between variables and alphabet elements don't occur.

Suppose G_1 and G_2 are grammars. We can define a grammar H such that $L(H) = L(G_1)L(G_2)$ by unioning together the variables and productions of G_1 and G_2, and adding a new start variable q, along with production

$$q \rightarrow s_{G_1} s_{G_2}.$$

Suppose G is a grammar. We can define a grammar H such that $L(H) = L(G)^*$ by adding to the variables and productions of G a new start variable q, along with productions

$$q \rightarrow \% \mid s_G q.$$

Next, we consider reversal and alphabet renaming operations on grammars. Given a grammar G, we can define a grammar H such that $L(H) = L(G)^R$ by simply reversing the right-sides of G's productions.

Given a grammar G and a bijection f from a set of symbols that is a superset of $\mathbf{alphabet}(G)$ to some set of symbols, we can define a grammar H such that $L(H) = L(G)^f$ by renaming the elements of $\mathbf{alphabet}(G)$ in the right-sides of G's productions using f. Actually, we may have to rename the variables of G to avoid clashes with the elements of the renamed alphabet.

The Forlan module `Gram` defines the following constants and operations on grammars:

```
val emptyStr       : gram
val emptySet       : gram
val fromStr        : str -> gram
val fromSym        : sym -> gram
val union          : gram * gram -> gram
val concat         : gram * gram -> gram
val closure        : gram -> gram
val rev            : gram -> gram
val renameAlphabet : gram * sym_rel -> gram
```

For example, we can construct a grammar G such that $L(G) = \{01\} \cup \{10\}\{11\}^*$, as follows.

```
- val gram1 = Gram.fromStr(Str.fromString "01");
val gram1 = - : gram
- val gram2 = Gram.fromStr(Str.fromString "10");
val gram2 = - : gram
- val gram3 = Gram.fromStr(Str.fromString "11");
val gram3 = - : gram
- val gram =
= Gram.union(gram1,
=            Gram.concat(gram2,
=                        Gram.closure gram3));
val gram = - : gram
- Gram.output("", gram);
{variables}
A, <1,A>, <2,A>, <2,<1,A>>, <2,<2,A>>, <2,<2,<A>>>
{start variable}
A
{productions}
A -> <1,A> | <2,A>; <1,A> -> 01;
<2,A> -> <2,<1,A>><2,<2,A>>; <2,<1,A>> -> 10;
<2,<2,A>> -> % | <2,<2,<A>>><2,<2,A>>; <2,<2,<A>>> -> 11
val it = () : unit
- val gram' = Gram.renameVariablesCanonically gram;
val gram' = - : gram
- Gram.output("", gram');
{variables}
A, B, C, D, E, F
{start variable}
A
{productions}
A -> B | C; B -> 01; C -> DE; D -> 10; E -> % | FE;
F -> 11
val it = () : unit
```

Continuing our Forlan session, the grammar reversal and alphabet renaming operations can be used as follows:

```
- val gram'' = Gram.rev gram';
val gram'' = - : gram
- Gram.output("", gram'');
{variables}
A, B, C, D, E, F
{start variable}
A
{productions}
A -> B | C; B -> 10; C -> ED; D -> 01; E -> % | EF;
F -> 11
val it = () : unit
- val rel = SymRel.fromString "(0, A), (1, B)";
val rel = - : sym_rel
- val gram''' = Gram.renameAlphabet(gram'', rel);
val gram''' = - : gram
- Gram.output("", gram''');
{variables}
<A>, <B>, <C>, <D>, <E>, <F>
{start variable}
<A>
{productions}
<A> -> <B> | <C>; <B> -> BA; <C> -> <E><D>; <D> -> AB;
<E> -> % | <E><F>; <F> -> BB
val it = () : unit
```

4.8 Converting Regular Expressions and Finite Automata to Grammars

There are simple algorithms for converting regular expressions and finite automata to grammars. Since we have algorithms for converting between regular expressions and finite automata, it is tempting to only define one of these algorithms. But translating FAs to regular expressions is expensive, and often yields large regular expressions. Thus it would be impractical to translate FAs to grammars by first translating them to regular expressions, and then translating the regular expressions to grammars. Hence it is important to give a direct translation from FAs to grammars.

Although it would be satisfactory to translate regular expressions to grammars by first translating them to FAs, and then translating the FAs to grammars, it's easy to define a direct translation, and the results of the direct translation will be a little nicer.

Regular expressions are converted to grammars using a recursive algorithm that makes use of the operations on grammars that were defined in Section 4.7. The structure of the algorithm is very similar to the structure of our algorithm for converting regular expressions to finite automata.

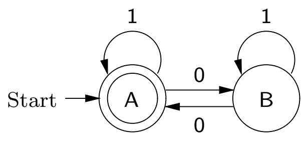
The algorithm is implemented in Forlan by the function

```
val fromReg : reg -> gram
```

of the `Gram` module. It's available in the top-level environment with the name `regToGram`. Here is how we can convert the regular expression $\mathsf{01} + \mathsf{10}(\mathsf{11})^*$ to a grammar using Forlan:

```
- val gram = regToGram(Reg.input "");
@ 01 + 10(11)*
@ .
val gram = - : gram
- Gram.output("", Gram.renameVariablesCanonically gram);
{variables}
A, B, C, D, E, F
{start variable}
A
{productions}
A -> B | C; B -> 01; C -> DE; D -> 10; E -> % | FE;
F -> 11
val it = () : unit
```

We'll explain the process of converting finite automata to grammars using an example. Suppose M is the DFA



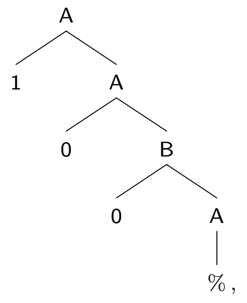
The variables of our grammar G consist of the states of M, and its start variable is the start state A of M. (If the symbols of the labels of M's transitions conflict with M's states, we'll have to rename the states of M first.) We can translate each transition (q, x, r) to a production $q \rightarrow xr$. And, since A is an accepting state of M, we add the production $\mathsf{A} \rightarrow \%$. This gives us the grammar

$$\mathsf{A} \rightarrow \% \mid \mathsf{0B} \mid \mathsf{1A},$$
$$\mathsf{B} \rightarrow \mathsf{0A} \mid \mathsf{1B}.$$

Consider, e.g., the valid labeled path for M

$$\mathsf{A} \overset{1}{\Rightarrow} \mathsf{A} \overset{0}{\Rightarrow} \mathsf{B} \overset{0}{\Rightarrow} \mathsf{A},$$

which explains why $100 \in L(M)$. It corresponds to the valid parse tree for G



which explains why $100 \in L(G)$.

The Forlan module `Gram` contains the function

```
val fromFA : fa -> gram
```

which implements our algorithm for converting finite automata to grammars. It's available in the top-level environment with the name `faToGram`. Suppose `fa` of type `fa` is bound to M. Here is how we can convert M to a grammar using Forlan:

```
- val gram = faToGram fa;
val gram = - : gram
- Gram.output("", gram);
{variables}
A, B
{start variable}
A
{productions}
A -> % | 0B | 1A; B -> 0A | 1B
val it = () : unit
```

Because of the existence of our conversion functions, we have that every regular language is a context-free language. On the other hand, the language $\{\, 0^n1^n \mid n \in \mathbb{N} \,\}$ is context-free, because of the grammar

$$\mathsf{A} \rightarrow \% \mid \mathsf{0A1},$$

but is not regular, as we proved in Section 3.13. Thus the regular languages are a proper subset of the context-free languages: **RegLan** $\subsetneq$ **CFLan**.

4.9 Chomsky Normal Form

In this section, we study a special form of grammars called Chomsky Normal Form (CNF). CNF was invented by, and named after, the linguist Noam Chomsky. Grammars in CNF have very nice formal properties. In particular, valid parse trees for grammars in CNF are very close to being binary trees. Any grammar that doesn't generate $\%$ can be put in CNF. And, if G is a grammar that does generate $\%$, it can be turned into a grammar in CNF that generates $L(G) - \{\%\}$. In the next section, we will use this fact when proving the pumping lemma for context-free languages, a method for showing the certain languages are not context-free.

When converting a grammar to CNF, we will first get rid of productions of the form $\mathsf{A} \rightarrow \%$ and $\mathsf{A} \rightarrow \mathsf{B}$, where A and B are variables.

A $\%$-*production* is a production of the form $q \rightarrow \%$. We will show by example how to turn a grammar G into a simplified grammar with no $\%$-productions that generates $L(G) - \{\%\}$. Suppose G is the grammar

$$\begin{aligned}\mathsf{A} &\rightarrow \mathsf{0A1} \mid \mathsf{BB},\\ \mathsf{B} &\rightarrow \% \mid \mathsf{2B}.\end{aligned}$$

First, we determine which variables q are *nullable* in the sense that $\% \in \Pi_q$, i.e., that $\%$ is the yield of a valid parse tree for G whose root label is q. Clearly, B is nullable. And, since $\mathsf{A} \rightarrow \mathsf{BB} \in P_G$, it follows that A is nullable.

Now we use this information as follows:

- Since A is nullable, we replace the production $\mathsf{A} \rightarrow \mathsf{0A1}$ with the productions $\mathsf{A} \rightarrow \mathsf{0A1}$ and $\mathsf{A} \rightarrow \mathsf{01}$. The idea is that this second production will make up for the fact that A won't be nullable in the new grammar.

- Since B is nullable, we replace the production $\mathsf{A} \rightarrow \mathsf{BB}$ with the productions $\mathsf{A} \rightarrow \mathsf{BB}$ and $\mathsf{A} \rightarrow \mathsf{B}$ (the result of deleting either one of the B's).

- The production $\mathsf{B} \rightarrow \%$ is deleted.

- Since B is nullable, we replace the production $\mathsf{B} \rightarrow \mathsf{2B}$ with the productions $\mathsf{B} \rightarrow \mathsf{2B}$ and $\mathsf{B} \rightarrow \mathsf{2}$.

This give us the grammar

$$\begin{aligned}\mathsf{A} &\rightarrow \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{B},\\ \mathsf{B} &\rightarrow \mathsf{2B} \mid \mathsf{2}.\end{aligned}$$

In general, we finish by simplifying our new grammar. The new grammar of our example is already simplified, however.

A *unit production* for a grammar G is a production of the form $q \to r$, where r is a variable (possibly equal to q). We now show by example how to turn a grammar G into a simplified grammar with no %-productions or unit productions that generates $L(G) - \{\%\}$.

Suppose G is the grammar

$$\begin{aligned}\mathsf{A} &\to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{B},\\ \mathsf{B} &\to \mathsf{2B} \mid \mathsf{2}.\end{aligned}$$

We begin by applying our algorithm for removing %-productions to our grammar; the algorithm has no effect in this case. Next, we generate the productions of a new grammar as follows. If

- q and r are variables of G,
- there is a valid parse tree for G whose root label is q and yield is r,
- $r \to w$ is a production of G, and
- w is not a single variable of G,

then we add $q \to w$ to the set of productions of our new grammar. (Determining whether there is a valid parse whose root label is q and yield is r is easy, since we are working with a grammar with no %-productions.) This process results in the grammar

$$\begin{aligned}\mathsf{A} &\to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{2B} \mid \mathsf{2},\\ \mathsf{B} &\to \mathsf{2B} \mid \mathsf{2}.\end{aligned}$$

Finally, we simplify our grammar, which has no effect in this case.

A grammar G is in *Chomsky Normal Form* (CNF) iff each of its productions has one of the following forms:

- $q \to a$, where a is not a variable;
- $q \to pr$, where p and r are variables.

We explain by example how a grammar G can be turned into a simplified grammar in CNF that generates $L(G) - \{\%\}$.

Suppose G is the grammar

$$\begin{aligned}\mathsf{A} &\to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{2B} \mid \mathsf{2},\\ \mathsf{B} &\to \mathsf{2B} \mid \mathsf{2}.\end{aligned}$$

We begin by applying our algorithm for removing %-productions and unit productions to this grammar. In this case, it has no effect. Then, we proceed as follows:

- Since the productions $\mathsf{A} \rightarrow \mathsf{BB}$, $\mathsf{A} \rightarrow \mathsf{2}$ and $\mathsf{B} \rightarrow \mathsf{2}$ are legal CNF productions, we simply transfer them to our new grammar.
- Next we add the variables $\langle\mathsf{0}\rangle$, $\langle\mathsf{1}\rangle$ and $\langle\mathsf{2}\rangle$ to our grammar, along with the productions
$$\langle\mathsf{0}\rangle \rightarrow \mathsf{0}, \quad \langle\mathsf{1}\rangle \rightarrow \mathsf{1}, \quad \langle\mathsf{2}\rangle \rightarrow \mathsf{2}.$$
- Now, we can replace the production $\mathsf{A} \rightarrow \mathsf{01}$ with $\mathsf{A} \rightarrow \langle\mathsf{0}\rangle\langle\mathsf{1}\rangle$. We can replace the production $\mathsf{A} \rightarrow \mathsf{2B}$ with $\mathsf{A} \rightarrow \langle\mathsf{2}\rangle\mathsf{B}$. And, we can replace the production $\mathsf{B} \rightarrow \mathsf{2B}$ with the production $\mathsf{B} \rightarrow \langle\mathsf{2}\rangle\mathsf{B}$.
- Finally, we replace the production $\mathsf{A} \rightarrow \mathsf{0A1}$ with the productions
$$\mathsf{A} \rightarrow \langle\mathsf{0}\rangle\mathsf{C}, \quad \mathsf{C} \rightarrow \mathsf{A}\langle\mathsf{1}\rangle,$$
and add C to the set of variables of our new grammar.

Summarizing, our new grammar is

$$\begin{aligned}
\mathsf{A} &\rightarrow \mathsf{BB} \mid \mathsf{2} \mid \langle\mathsf{0}\rangle\langle\mathsf{1}\rangle \mid \langle\mathsf{2}\rangle\mathsf{B} \mid \langle\mathsf{0}\rangle\mathsf{C}, \\
\mathsf{B} &\rightarrow \mathsf{2} \mid \langle\mathsf{2}\rangle\mathsf{B}, \\
\langle\mathsf{0}\rangle &\rightarrow \mathsf{0}, \\
\langle\mathsf{1}\rangle &\rightarrow \mathsf{1}, \\
\langle\mathsf{2}\rangle &\rightarrow \mathsf{2}, \\
\mathsf{C} &\rightarrow \mathsf{A}\langle\mathsf{1}\rangle.
\end{aligned}$$

The official version of our algorithm names variables in a different way.

The Forlan module `Gram` defines the following functions:

```
val removeEmptyProductions        : gram -> gram
val removeEmptyAndUnitProductions : gram -> gram
val chomskyNormalForm             : gram -> gram
```

Suppose `gram` of type `gram` is bound to the grammar with variables A and B, start variable A, and productions

$$\begin{aligned}
\mathsf{A} &\rightarrow \mathsf{0A1} \mid \mathsf{BB}, \\
\mathsf{B} &\rightarrow \% \mid \mathsf{2B}.
\end{aligned}$$

Here is how Forlan can be used to turn this grammar into a CNF grammar that generates the nonempty strings that are generated by `gram`:

```
- val gram' = Gram.chomskyNormalForm gram;
val gram' = - : gram
- Gram.output("", gram');
{variables}
<1,A>, <1,B>, <2,0>, <2,1>, <2,2>, <3,A1>
{start variable}
<1,A>
{productions}
<1,A> ->
2 | <1,B><1,B> | <2,0><2,1> | <2,0><3,A1> | <2,2><1,B>;
<1,B> -> 2 | <2,2><1,B>; <2,0> -> 0; <2,1> -> 1;
<2,2> -> 2; <3,A1> -> <1,A><2,1>
val it = () : unit
- val gram'' = Gram.renameVariablesCanonically gram';
val gram'' = - : gram
- Gram.output("", gram'');
{variables}
A, B, C, D, E, F
{start variable}
A
{productions}
A -> 2 | BB | CD | CF | EB; B -> 2 | EB; C -> 0; D -> 1;
E -> 2; F -> AD
val it = () : unit
```

4.10 The Pumping Lemma for Context-free Languages

Let L be the language $\{\, 0^n1^n2^n \mid n \in \mathbb{N}\,\}$. Is L context-free? I.e., is there a grammar that generates L? It seems that the answer is "no". Although it's easy to keep the 0's and 1's matched, or to keep the 1's and 2's matched, or to keep the 0's and 2's matched, there is no obvious way to keep all three symbols matched simultaneously.

In this section, we will study the pumping lemma for context-free languages, which can be used to show that many languages are not context-free. We will use the pumping lemma to prove that L is not context-free, and then we will prove the lemma. Building on this result, we'll be able to show that the context-free languages are not closed under intersection, complementation or set-difference.

Lemma 4.10.1 (Pumping Lemma for Context Free Languages)
For all context-free languages L, there is a $n \in \mathbb{N}$ such that, for all $z \in$ **Str**,

*if $z \in L$ and $|z| \geq n$, then there are $u, v, w, x, y \in$ **Str** such that $z = uvwxy$ and*

(1) $|vwx| \leq n$;

(2) $vx \neq \%$; and

(3) $uv^iwx^iy \in L$, for all $i \in \mathbb{N}$.

Before proving the pumping lemma, let's see how it can be used to show that $L = \{\, 0^n1^n2^n \mid n \in \mathbb{N}\,\}$ is not context-free.

Proposition 4.10.2
L is not regular.

Proof. Suppose, toward a contradiction that L is context-free. Thus there is an $n \in \mathbb{N}$ with the property of the lemma. Let $z = 0^n1^n2^n$. Since $z \in L$ and $|z| = 3n \geq n$, we have that there are $u, v, w, x, y \in$ **Str** such that $z = uvwxy$ and

(1) $|vwx| \leq n$;

(2) $vx \neq \%$; and

(3) $uv^iwx^iy \in L$, for all $i \in \mathbb{N}$.

Since $0^n1^n2^n = z = uvwxy$, (1) tells us that vwx doesn't contain both a 0 and a 2. Thus, either vwx has no 0's, or vwx has no 2's, so that there are two cases to consider.

Suppose vwx has no 0's. Thus vx has no 0's. By (2), we have that vx contains a 1 or a 2. Thus uwy:

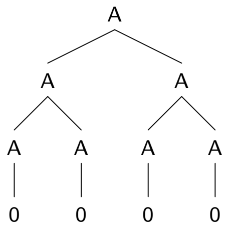
- has n 0's;
- either has less than n 1's or has less than n 2's.

But (3) tells us that $uwy = uv^0wx^0y \in L$, so that uwy has an equal number of 0's, 1's and 2's—contradiction.

The case where vwx has no 2's is similar.

Since we obtained a contradiction in both cases, we have an overall contradiction. Thus L is not context-free. $\square$

When we prove the pumping lemma for context-free languages, we will make use of a fact about grammars in Chomsky Normal Form. Suppose G is a grammar in CNF and that $w \in \textbf{alphabet}(G)^*$ is the yield of a valid parse tree pt for G whose root label is a variable. For instance, if G is the grammar with variable A and productions $\mathsf{A} \rightarrow \mathsf{AA}$ and $\mathsf{A} \rightarrow \mathsf{0}$, then w could be $\mathsf{0000}$ and pt could be the following tree of height 3:



Generalizing from this example, we can see that, if pt has height 3, $|w|$ will never be greater than $4 = 2^2 = 2^{3-1}$.

Lemma 4.10.3
Suppose G is a grammar in CNF, pt is a valid parse tree for G of height k, the root label of pt is a variable of G, and the yield w of pt is in $\textbf{alphabet}(G)^*$. *Then $|w| \leq 2^{k-1}$.*

Proof. By induction on pt. □

Now, let's prove the pumping lemma.

Proof. Suppose L is a context-free language. By the results of the preceding section, there is a grammar G in Chomsky Normal Form such that $L(G) = L - \{\%\}$. Let $k = |Q_G|$ and $n = 2^k$. Suppose $z \in \mathit{Str}$, $z \in L$ and $|z| \geq n$. Since $n \geq 2$, we have that $z \neq \%$. Thus $z \in L - \{\%\} = L(G)$, so that there is a parse tree pt such that pt is valid for G, $\textbf{rootLabel}(pt) = s_G$ and $\textbf{yield}(pt) = z$. By Lemma 4.10.3, we have that the height of pt is at least $k + 1$. (If pt's height were only k, then $|z| \leq 2^{k-1} < n$, which is impossible.) The rest of the proof can be visualized using the diagram in Figure 4.1.

Let pat be a valid path for pt whose length is equal to the height of pt. Thus the length of pt is at least $k + 1$, so that the path visits at least $k + 1$ variables, with the consequence that at least one variable must be visited twice. Working from the last variable visited upwards, we look for the first repetition of variables. Suppose q is this repeated variable, and let pat' and pat'' be the initial parts of pat that take us to the upper and lower occurrences of q, respectively.

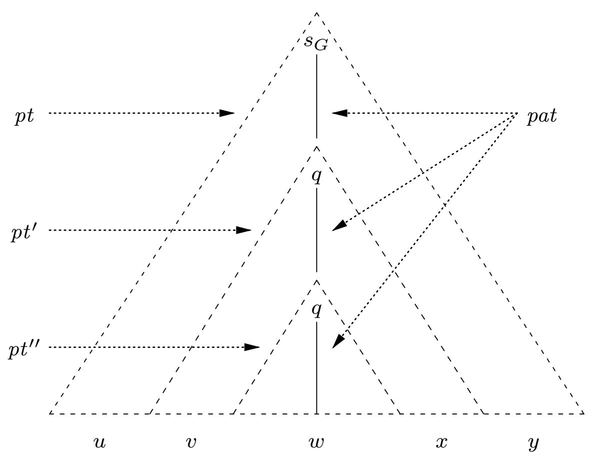


Figure 4.1: Visualization of Proof of Pumping Lemma for Context-free Languages

Let pt' and pt'' be the subtrees of pt at positions pat' and pat'', i.e., the positions of the upper and lower occurrences of q, respectively.

Consider the tree formed from pt by replacing the subtree at position pat' by q. This tree has yield uqy, for unique strings u and y.

Consider the tree formed from pt' by replacing the subtree pt'' by q. More precisely, form the path pat''' by removing pat' from the beginning of pat''. Then replace the subtree of pt' at position pat''' by q. This tree has yield vqx, for unique strings v and x.

Furthermore, since $|pat|$ is the height of pt, the length of the path formed by removing pat' from pat will be the height of pt'. But we know that this length is at most $k{+}1$, because, when working upwards through the variables visited by pat, we stopped as soon as we found a repetition of variables. Thus the height of pt' is at most $k + 1$.

Let w be the yield of pt''. Thus vwx is the yield of pt', so that $z = uvwxy$ is the yield of pt. Because the height of pt' is at most $k + 1$, our fact about valid parse trees of grammars in CNF, tells us that $|vwx| \leq 2^{(k+1)-1} = 2^k = n$, showing that Part (1) holds.

Because G is in CNF, pt', which has q as its root label, has two children. The child whose root node isn't visited by pat''' will have a non-empty yield, and this yield will be a prefix of v, if this child is the left child, and will be a suffix of x, if this child is the right child. Thus $vx \neq \%$, showing that Part (2) holds.

It remains to show Part (3), i.e., that $uv^iwx^iy \in L(G) \subseteq L$, for all $i \in \mathbb{N}$. We define a valid parse tree pt_i for G, with root label q and yield v^iwx^i, by recursion on $i \in \mathbb{N}$. We let pt_0 be pt''. Then, if $i \in \mathbb{N}$, we form pt_{i+1} from pt' by replacing the subtree at position pat'''' by pt_i.

Suppose $i \in \mathbb{N}$. Then the parse tree formed from pt by replacing the subtree at position pat' by pt_i is valid for G, has root label s_G, and has yield uv^iwx^iy, showing that $uv^iwx^iy \in L(G)$. $\square$

We conclude this section by considering some consequence of the pumping lemma. Suppose

$$\begin{aligned} L &= \{\, \mathsf{0}^n\mathsf{1}^n\mathsf{2}^n \mid n \in \mathbb{N}\,\}, \\ A &= \{\, \mathsf{0}^n\mathsf{1}^n\mathsf{2}^m \mid n, m \in \mathbb{N}\,\}, \\ B &= \{\, \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m \mid n, m \in \mathbb{N}\,\}. \end{aligned}$$

Of course, L is not context-free. It is easy to find grammars generating A and B, and so A and B are context-free. But $A \cap B = L$, and thus the context-free languages are not closed under intersection.

We can build on this example, in order to show that the context-free languages are not closed under complementation or set difference. The language

$$\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - A$$

is context-free, since it is the union of the context-free languages

$$\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - \{\mathsf{0}\}^*\{\mathsf{1}\}^*\{\mathsf{2}\}^*$$

and

$$\{\, \mathsf{0}^{n_1}\mathsf{1}^{n_2}\mathsf{2}^m \mid n_1, n_2, m \in \mathbb{N} \text{ and } n_1 \neq n_2 \,\},$$

(the first of these languages is regular), and the context-free languages are closed under union. Similarly, we have that $\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - B$ is context-free. Let

$$C = (\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - A) \cup (\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - B).$$

Thus C is a context-free subset of $\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^*$. Since $A, B \subseteq \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^*$, it is easy to show that

$$\begin{aligned} A \cap B &= \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - ((\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - A) \cup (\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - B)) \\ &= \{\mathsf{0}, \mathsf{1}, \mathsf{2}\}^* - C. \end{aligned}$$

Thus

$$\{0,1,2\}^* - C = A \cap B = L$$

is not context-free. Thus the context-free languages aren't closed under complementation. And, since $\{0,1,2\}^*$ is regular and thus context-free, it follows that the context-free languages are not closed under set difference.

Chapter 5

Recursive and Recursively Enumerable Languages

In this chapter, we will study a universal programming language, which we will use to define the recursive and recursively enumerable languages. We will see that the context-free languages are a proper subset of the recursive languages, that the recursive languages are a proper subset of the recursively enumerable languages, and that there are languages that are not recursively enumerable. Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Traditionally, one uses Turing machines for the universal programming language. Turing machines are finite automata that manipulate infinite tapes. Although Turing machines are very appealing in some ways, they are rather far-removed from conventional programming languages, and are hard to build and reason about.

Instead, we will work with a variant of the programming language Lisp. This programming language will have the same power as Turing machines, but it will be much easier to program in this language than with Turing machines. An "implementation" of our language (or of Turing machines) on a real computer will run out of resources on some programs.

5.1 A Universal Programming Language, and Recursive and Recursively Enumerable Languages

We will work with a variant of the programming language Lisp, with the following properties and features:

- The language is statically scoped, dynamically typed, deterministic and functional.
- The language's types are infinite precision integers, booleans, formal language symbols and arbitrary-length strings, arbitrary-length lists, and an error type (whose only value is **error**). There are the usual functions for going back and forth between integers and symbols.
- A program consists of one of more function definitions. The last function definition of a program is called its principal function. This function should take in some number of strings, and return a boolean.

The set **Prog** of *programs* is a subset of **Syn***, where the alphabet **Syn** consists of:

- the digits 0–9;
- the letters a–z and A–Z; and
- the symbols ⟨space⟩, ⟨newline⟩, ⟨openPar⟩ and ⟨closPar⟩.

When we present programs, however, we typically substitute a blank for ⟨space⟩, a newline for ⟨newline⟩, "(" for ⟨openPar⟩, and ")" for ⟨closPar⟩.

More detail about our programming language will eventually be given in this book. For example, here is a program that tests whether its argument string is nonempty:

```
(defun test (x) (not (equal (size x) 0)))
```

Given an $n \in \mathbb{N}$, the function $\mathbf{run}_n \in \mathbf{Str} \times \mathbf{Str} \uparrow n \to \{\mathbf{true}, \mathbf{false}, \mathbf{error}, \mathbf{nonterm}\}$, where $\mathbf{Str} \uparrow n$ consists of all n-tuples of strings, returns the following answer when called with arguments $(P, (x_1, \ldots, x_n))$:

- If P is not a program, or the principal function of P doesn't have exactly n arguments, then it returns **error**.

- Otherwise, if running P's principal function with arguments $x_1, \ldots, x_n$ results in the value **true** or **false** being returned, then it returns this value.
- Otherwise, if running P's principal function with these arguments causes some other value to be returned, then it returns **error**.
- Otherwise, running P's principal function with these arguments never terminates, and it returns **nonterm**.

Eventually, the book will contain a formal definition of the $\mathbf{run}_n$ functions.

It is possible to write a function in our programming language that checks whether a string is a valid program whose principal function has a given number of arguments. This will involve writing a function for parsing programs, where parse trees will be represented using lists.

With some effort, it is possible to write a function in our programming language that acts as an *interpreter*. It takes in a string w and a list of strings $(x_1, \ldots, x_n)$. If w is not a program whose principal function has n arguments, then the interpreter returns **error**. Otherwise, it simulates the running of w with input $(x_1, \ldots, x_n)$, returning what w returns, if w returns a boolean, and returning **error**, if w returns something else. Of course, w may also run forever, in which case the interpreter will also run forever.

We can also write a function in our programming language that acts as an *incremental* interpreter. Like an ordinary interpreter, it takes in a string w and a list of strings $(x_1, \ldots, x_n)$. If w is not a program whose principal function has n arguments, then the incremental interpreter returns **error**. Otherwise, it carries out a fixed number of steps of the running of w with input $(x_1, \ldots, x_n)$. If w has returned a boolean by this point, then the incremental interpreter returns this boolean. If w has returned something else by this point, then the incremental interpreter returns **error**. But if w hasn't yet terminated, the incremental interpreter returns a function that, when called, will continue the incremental interpretation process.

Given $n \in \mathbb{N}$, we say that a program P is *n-total* iff, for all $x_1, \ldots, x_n \in \mathbf{Str}$, $\mathbf{run}_n(P, (x_1, \ldots, x_n)) \in \{\mathbf{true}, \mathbf{false}\}$. A string w is *accepted by* a program P iff $\mathbf{run}_1(P, w) = \mathbf{true}$, i.e., iff running P with input w results in **true** being returned. (We write the 1-tuple whose single component is w as w.) The language *accepted by* a program P ($L(P)$) is

$$\{\, w \in \mathbf{Str} \mid w \text{ is accepted by } P \,\},$$

if this set of strings is a language (i.e., $\bigcup\{\, \mathbf{alphabet}(w) \mid w \in \mathbf{Str} \text{ and } w$ is accepted by $P \,\}$ is finite); otherwise $L(P)$ is undefined.

For example, if P's principal function takes in more than one argument, then $L(P) = \emptyset$, but if P's principal function takes in a single argument but always returns **true**, then $L(P)$ is undefined.

We say that a language L is:

- *recursive* iff $L = L(P)$ for some 1-total program P;
- *recursively enumerable* (r.e.) iff $L = L(P)$ for some program P.

(When we say $L = L(P)$, this means that $L(P)$ is defined, and L and $L(P)$ are equal.) We define

$$\begin{aligned}\mathbf{RecLan} &= \{\, L \in \mathbf{Lan} \mid L \text{ is recursive}\,\},\\ \mathbf{RELan} &= \{\, L \in \mathbf{Lan} \mid L \text{ is recursively enumerable}\,\}.\end{aligned}$$

Hence $\mathbf{RecLan} \subseteq \mathbf{RELan}$. Because **Prog** is countably infinite, we have that **RecLan** and **RELan** are countably infinite, so that $\mathbf{RELan} \subsetneq \mathbf{Lan}$. Later we will see that $\mathbf{RecLan} \subsetneq \mathbf{RELan}$.

Proposition 5.1.1
For all $L \in \mathbf{Lan}$, L is recursive iff there is a program P such that, for all $w \in \mathbf{Str}$:

- *if $w \in L$, then $\mathbf{run}_1(P, w) = \mathbf{true}$;*
- *if $w \notin L$, then $\mathbf{run}_1(P, w) = \mathbf{false}$.*

Proof. ("only if" direction) Since L is recursive, $L = L(P)$ for some 1-total program P. Suppose $w \in \mathbf{Str}$. There are two cases to show.

Suppose $w \in L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) = \mathbf{true}$.

Suppose $w \notin L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) \neq \mathbf{true}$. But P is 1-total, and thus $\mathbf{run}_1(P, w) = \mathbf{false}$.

("if" direction) To see that P is 1-total, suppose $w \in \mathbf{Str}$. Since $w \in L$ or $w \notin L$, we have that $\mathbf{run}_1(P, w) \in \{\mathbf{true}, \mathbf{false}\}$. Let $X = \{\, w \in \mathbf{Str} \mid w$ is accepted by $P\,\}$ (so far, we don't know that X is a language). We will show that $L = X$.

Suppose $w \in L$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in X$.

Suppose $w \in X$, so that $\mathbf{run}_1(P, w) = \mathbf{true}$. If $w \notin L$, then $\mathbf{run}_1(P, w) = \mathbf{false}$—contradiction. Thus $w \in L$.

Since $L = X$, we have that X is a language. Thus $L(P)$ is defined and is equal to X. Hence $L = L(P)$, finishing the proof that L is recursive. $\square$

Proposition 5.1.2
For all $L \in \mathbf{Lan}$, L is recursively enumerable iff there is a program P such that, for all $w \in \mathbf{Str}$,

$$w \in L \;\textit{ iff }\; \mathbf{run}_1(P, w) = \mathbf{true}.$$

Proof. ("only if" direction) Since L is recursively enumerable, $L = L(P)$ for some program P. Suppose $w \in \mathbf{Str}$.

Suppose $w \in L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) = \mathbf{true}$.

Suppose $\mathbf{run}_1(P, w) = \mathbf{true}$. Thus $w \in L(P) = L$.

("if" direction) Let $X = \{ w \in \mathbf{Str} \mid w \text{ is accepted by } P \}$ (so far, we don't know that X is a language). We will show that $L = X$.

Suppose $w \in L$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in X$. Suppose $w \in X$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in L$.

Since $L = X$, we have that X is a language. Thus $L(P)$ is defined and is equal to X. Hence $L = L(P)$, completing the proof that L is recursively enumerable. □

We have that every context-free language is recursive, but that not every recursive language is context-free, i.e., $\mathbf{CFLan} \subsetneq \mathbf{RecLan}$.

To see that every context-free language is recursive, let L be a context-free language. Thus there is a grammar G such that $L = L(G)$. With some work, we can write and prove the correctness of a program P that implements our algorithm (see Section 4.3) for checking whether a string is generated by a grammar. Thus L is recursive.

To see that not every recursive language is context-free, let $L = \{ \mathsf{0}^n\mathsf{1}^n\mathsf{2}^n \mid n \in \mathbb{N} \}$. In Section 4.10, we learned that L is not context-free. And it is easy to write a program P that tests whether a string is in L. Thus L is recursive.

5.2 Closure Properties of Recursive and Recursively Enumerable Languages

In this section, we will see that the recursive and recursively enumerable languages are closed under union, concatenation, closure and intersection. The recursive languages are also closed under set difference and complementation. In the next section, we will see that the recursively enumerable languages are not closed under complementation or set difference. On the other hand, we will see in this section that, if a language and its complement are both r.e., then the language is recursive.

Theorem 5.2.1
If L, L_1 and L_2 are recursive languages, then so are $L_1 \cup L_2$, L_1L_2, L^*, $L_1 \cap L_2$ and $L_1 - L_2$.

Proof. Let's consider the concatenation case as an example. Since L_1 and L_2 are recursive languages, there are programs P_1 and P_2 that test whether strings are in L_1 and L_2, respectively. We combine the functions of P_1 and P_2 to form a program Q that takes in a string w and behaves as follows. First, it generates all of the pairs of strings (x, y) such that $xy = w$. Then it works though these pairs, one by one. Given such a pair (x, y), Q calls the principal function of P_1 to check whether $x \in L_1$. If the answer is "no", then it goes on to the next pair. Otherwise, it calls the principal function of P_2 to check whether $y \in L_2$. If the answer is "no", then it goes on to the next pair. Otherwise, it returns **true**. If Q runs out of pairs to check, then it returns **false**.

We can check that, for all $w \in \mathbf{Str}$, Q tests whether $w \in L_1L_2$. Thus L_1L_2 is recursive. $\square$

Corollary 5.2.2
If Σ is an alphabet and $L \subseteq \Sigma^*$ is recursive, then so is $\Sigma^* - L$.

Proof. Follows from Theorem 5.2.1, since Σ^* is recursive. $\square$

Theorem 5.2.3
If L, L_1 and L_2 are recursively enumerable languages, then so are $L_1 \cup L_2$, L_1L_2, L^* and $L_1 \cap L_2$.

Proof. We consider the concatenation case as an example. Since L_1 and L_2 are recursively enumerable, there are programs P_1 and P_2 such that, for all $w \in \mathbf{Str}$, $w \in L_1$ iff $\mathbf{run}_1(P_1, w) = \mathbf{true}$, and $w \in L_2$ iff $\mathbf{run}_1(P_2, w) = \mathbf{true}$. (Remember that P_1 and P_2 may fail to terminate on some inputs.) To show that L_1L_2 is recursively enumerable, we will construct a program Q such that, for all $w \in \mathbf{Str}$, $w \in L_1L_2$ iff $\mathbf{run}_1(Q, w) = \mathbf{true}$.

When Q is called with a string w, it behaves as follows. First, it generates all the pairs of strings (x, y) such that $w = xy$. Let these pairs be $(x_1, y_1), \ldots, (x_n, y_n)$. Now, Q uses our *incremental* interpretation function to work its way through all the string pairs, running P_1 with input x_1 (for some fixed number of steps), P_2 with input y_1, P_1 with input x_2, P_2 with input y_2, ..., P_1 with input x_n and P_2 with input y_n. It then begins a second round of all of these incremental interpretations, followed by a third round, and so on.

- If, at some stage, the incremental interpretation of P_1 on input x_i returns **true**, then x_i is marked as being in L_1.
- If, at some stage, the incremental interpretation of P_2 on input y_i returns **true**, then the y_i is marked as being in L_2.
- If, at some stage, the incremental interpretation of P_1 on input x_i terminates with **false** or **error**, then the i'th pair is marked as discarded.
- If, at some stage, the incremental interpretation of P_2 on input y_i terminates with **false** or **error**, then the i'th pair is marked as discarded.
- If, at some stage, x_i is marked as in L_1 and y_i is marked as in L_2, then Q returns **true**.
- If, at some stage, there are no remaining pairs, then Q returns **false**.

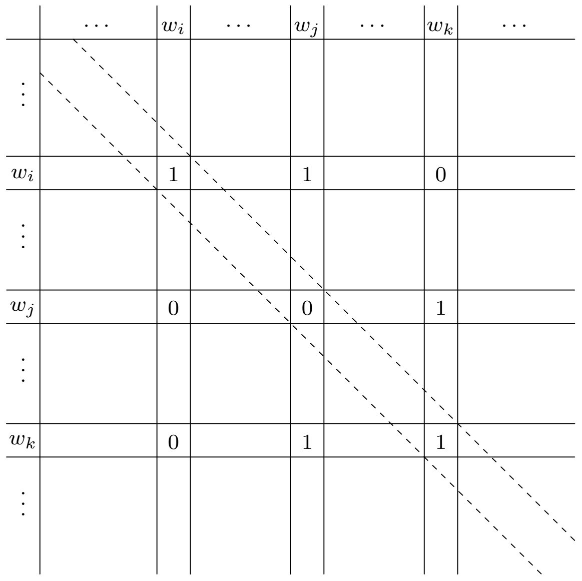
We can check that for all $w \in \textbf{Str}$, $w \in L_1L_2$ iff $\textbf{run}_1(Q, w) = \textbf{true}$. $\square$

Theorem 5.2.4
If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is recursive.

Proof. Since L and $\Sigma^* - L$ are recursively enumerable languages, there are programs P and P' such that, for all $w \in \textbf{Str}$, $w \in L$ iff $\textbf{run}_1(P, w) = \textbf{true}$, and $w \in \Sigma^* - L$ iff $\textbf{run}_1(P', w) = \textbf{true}$. We construct a program Q that behaves as follows when called with a string w. If $w \notin \Sigma^*$, then Q returns **false**. Otherwise, Q alternates between incrementally interpreting P with input w and incrementally interpreting P' with input w.

- If, at some stage, the incremental interpretation of P returns **true**, then Q returns **true**.
- If, at some stage, the incremental interpretation of P returns **false** or **error**, then Q returns **false**.
- If, at some stage, the incremental interpretation of P' returns **true**, then Q returns **false**.
- If, at some stage, the incremental interpretation of P' returns **false** or **error**, then Q returns **true**.

We can check that, for all $w \in \textbf{Str}$, Q tests whether $w \in L$. $\square$



	$\cdots$	w_i	$\cdots$	w_j	$\cdots$	w_k	$\cdots$
$\vdots$							
w_i		1		1		0	
$\vdots$							
w_j		0		0		1	
$\vdots$							
w_k		0		1		1	
$\vdots$							

Figure 5.1: Example Diagonalization Table for R.E. Languages

5.3 Diagonalization and Undecidable Problems

In this section, we will use a technique called diagonalization to find a natural language that isn't recursively enumerable. This will lead us to a language that is recursively enumerable but is not recursive. It will also enable us to prove the undecidability of the halting problem.

To find a non-r.e. language, we can use a technique called "diagonalization". Remember that the alphabet **Syn** consists of the digits, the lowercase and uppercase letters, and the symbols $\langle\mathsf{space}\rangle$, $\langle\mathsf{newline}\rangle$, $\langle\mathsf{openPar}\rangle$ and $\langle\mathsf{closPar}\rangle$. Furthermore $\mathbf{Prog} \subseteq \mathbf{Syn}^*$.

Consider the infinite table in which both the rows and the columns are indexed by the elements of $\mathbf{Syn}^*$, listed in some order $w_1, w_2, \ldots$, and where a cell (w_n, w_m) contains 1 iff $\mathbf{run}_1(w_n, w_m) = \mathbf{true}$, and contains 0 iff $\mathbf{run}_1(w_n, w_m) \neq \mathbf{true}$. Each recursively enumerable language is $L(w_n)$ for some n.

Figure 5.1 shows how part of this table might look, where w_i, w_j and w_k are sample elements of $\mathbf{Syn}^*$. Because of the table's data, we have that $\mathbf{run}_1(w_i, w_i) = \mathbf{true}$ and $\mathbf{run}_1(w_i, w_j) \neq \mathbf{true}$.

To define a non-r.e. language, we work our way down the diagonal of the table, putting w_n into our language just when cell (w_n, w_n) of the table is

0, i.e., when $\mathbf{run}_1(w_n, w_n) \neq \mathbf{true}$. Thus, if w_n is a program, we will have that w_n is in our language exactly when w_n doesn't accept w_n. This will ensure that $L(w_n)$ (if it is defined) is not our language.

With our example table:

- $L(w_i)$ (if it is defined) is not our language, since $w_i \in L(w_i)$, but w_i is not in our language;
- $L(w_j)$ (if it is defined) is not our language, since $w_j \notin L(w_j)$, but w_j is in our language;
- $L(w_k)$ (if it is defined) is not our language, since $w_k \in L(w_k)$, but w_k is not in our language.

We formalize the above ideas as follows. Define languages L_d ("d" for "diagonal") and L_a ("a" for "accepted") by:

$$\begin{aligned} L_d &= \{\, w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) \neq \mathbf{true} \,\}, \\ L_a &= \{\, w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) = \mathbf{true} \,\}. \end{aligned}$$

Thus $L_d = \mathbf{Syn}^* - L_a$. Because of the way $\mathbf{run}_1$ is defined, we have that every element of L_a is a program whose principal function has a single argument.

Theorem 5.3.1
L_d is not recursively enumerable.

Proof. Suppose, toward a contradiction, that L_d is recursively enumerable. Thus, there is a program P such that $L_d = L(P)$. There are two cases to consider.

Suppose $P \in L_d$. Then $\mathbf{run}_1(P, P) \neq \mathbf{true}$, i.e., P is not accepted by P. But then $P \notin L(P) = L_d$—contradiction.

Suppose $P \notin L_d$. Since $P \in \mathbf{Syn}^*$, we have that $\mathbf{run}_1(P, P) = \mathbf{true}$, i.e., P is accepted by P. But then $P \in L(P) = L_d$—contradiction.

Since we obtained a contradiction in both cases, we have an overall contradiction. Thus L_d is not recursively enumerable. $\square$

Theorem 5.3.2
L_a is recursively enumerable.

Proof. Let P be the program that, when given a string w, uses our interpreter function to simulate the execution of w on input w, returning **true** if the interpreter returns **true**, returning **false** if the interpreter returns **false** or **error**, and running forever, if the interpreter runs forever.

We can check that, for all $w \in \mathbf{Str}$, $w \in L_a = \{\, w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) = \mathbf{true}\,\}$ iff $\mathbf{run}_1(P, w) = \mathbf{true}$. Thus L_a is recursively enumerable. $\Box$

Corollary 5.3.3
There is an alphabet Σ and a recursively enumerable language $L \subseteq \Sigma^*$ such that $\Sigma^* - L$ is not recursively enumerable.

Proof. $L_a \subseteq \mathbf{Syn}^*$ is recursively enumerable, but $\mathbf{Syn}^* - L_a = L_d$ is not recursively enumerable. $\Box$

Corollary 5.3.4
There are recursively enumerable languages L_1 and L_2 such that $L_1 - L_2$ is not recursively enumerable.

Proof. Follows from Corollary 5.3.3, since Σ^* is recursively enumerable. $\Box$

Corollary 5.3.5
L_a is not recursive.

Proof. Suppose, toward a contradiction, that L_a is recursive. Since the recursive languages are closed under complementation, and $L_a \subseteq \mathbf{Syn}^*$, we have that $L_d = \mathbf{Syn}^* - L_a$ is recursive—contradiction. Thus L_a is not recursive. $\Box$

Since $L_a \in \mathbf{RELan}$, but $L_a \notin \mathbf{RecLan}$, we have that $\mathbf{RecLan} \subsetneq \mathbf{RELan}$. Combining this fact with facts learned in Sections 4.8 and 5.1, we have that

$$\mathbf{RegLan} \subsetneq \mathbf{CFLan} \subsetneq \mathbf{RecLan} \subsetneq \mathbf{RELan} \subsetneq \mathbf{Lan}.$$

Finally, we consider the famous halting problem. We say that a program P *halts on* a string w iff $\mathbf{run}_1(P, w) \neq \mathbf{nonterm}$.

Theorem 5.3.6
There is no program H such that, for all programs P and strings w:

- *If P halts on w, then* $\mathbf{run}_2(H,(P,w)) = \mathbf{true}$*;*
- *If P does not halt on w, then* $\mathbf{run}_2(H,(P,w)) = \mathbf{false}$.

Proof. Suppose, toward a contradiction, that such an H does exist. We use H to construct a program Q that behaves as follows when run on a string w. If w is not a program whose principal function has a single argument, then it returns **false**; otherwise it continues. Next, it uses our interpretation function to simulate the execution of the program H with inputs (w, w).

If H returns **true**, then Q uses our interpreter function to run w with input w. Since w halts on w, we know that this interpretation will terminate. If it terminates with value **true**, then Q returns **true**. Otherwise, it returns **false**.

Otherwise, H returns **false**. Then Q returns **false**.

We can check that, for all $w \in \mathbf{Str}$, Q tests whether $w \in L_a = \{\, w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w,w) = \mathbf{true}\,\}$. Thus L_a is recursive—contradiction. Thus no such H exists. $\square$

Here are two other undecidable problems:

- Determining whether two grammars generate the same language. (In contrast, we gave an algorithm for checking whether two FAs are equivalent, and this algorithm can be implemented as a program.)
- Determining whether a grammar is ambiguous.

Appendix A

GNU Free Documentation License

0 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1 Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they

4 Modifications

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the

7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Bibliography

[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.

[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.

[Lei00] H. Leiß. The Automata Library. `http://www.cis.uni-muenchen.de/~leiss/sml-automata.html`, 2000.

[LP98] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, second edition, 1998.

[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.* 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.

[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.

[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.

[Yu02] S. Yu. Grail+: A symbolic computation environment for finite-state machines, regular expressions, and finite languages. `http://www.csd.uwo.ca/research/grail/`, 2002.

Index