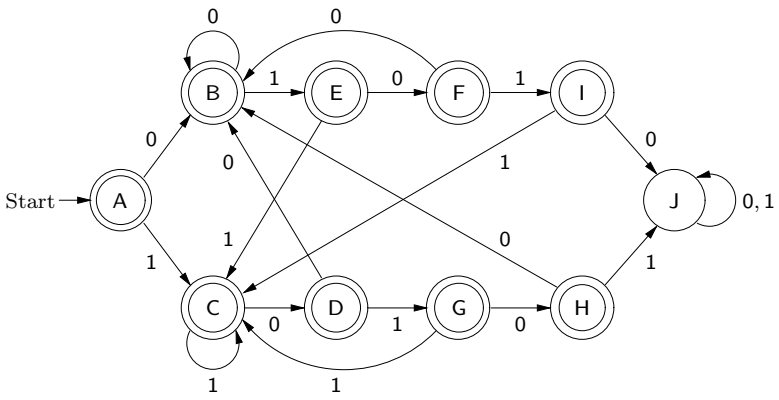
Formal Language Theory

Integrating Experimentation and Proof

Alley Stoughton

Draft of Fall 2019



Contents

List of Figures

Preface

Background

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

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

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

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize a deterministic finite automaton that accepts some language, L. With the paper-and-pencil approach, the student is obliged

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

Integrating Experimentation and Proof

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

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

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

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

In contrast to some books on formal language theory, the book emphasizes

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

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

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

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

Outline of the Book

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

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

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

In Chapter 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, top-down (recursive descent) parsing, and properties of context-free languages. We prove that the set of context-free languages is a proper superset of the set of regular languages.

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study the functional programming language that we use instead of Turing machines to define the recursive and recursively enumerable languages. We study algorithms for processing programs and for converting grammars to programs, and properties of recursive and recursively enumerable languages. We prove that the set of context-free languages is a proper subset of the set of recursive languages, that the set of recursive languages is a proper subset of the set of recursively enumerable languages, and that there are languages that are not recursively enumerable. Furthermore, we show that there are problems, like the halting problem (the problem of determining whether a program halts when run on a given input), that can't be solved by programs.

Further Reading and Related Work

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

Neil Jones [Jon97] pioneered the use of a programming language with structured data as an alternative to Turing machines for studying the limits of what is computable. In Chapter 5, we have followed Jones's approach in some ways. On the other hand, our programming language is functional, not imperative (assignment-oriented), and it has explicit support for the symbols and strings of formal language theory.

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

Notes, Exercises and Website

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

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

The Forlan website

`http://alleystoughton.us/forlan`

contains:

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

Acknowledgments

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

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

Chapter 1

Mathematical Background

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

1.1 Basic Set Theory

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

1.1.1 Review of Classical Logic

We begin by very briefly reviewing the rules of classical logic. We can build *formulas* using basic predicates (like "x = y"), plus the following quantifiers and connectives:

- **For all** $x, P(x)$ (*universal quantification*). The most typical way of proving such a formula is to add the variable x to the *variable context* (each element of which stands for some element of the universe of values)[1], and then go about proving $P(x)$ (possibly making use of any of the current *assumptions*).
- **There exists an** x, **such that** $P(x)$ (*existential quantification*). The most typical way of proving such a formula is to prove that $P(t)$ holds, where t is some term (expression), all of whose variables appear in the variable context.
- P **and** Q (*conjunction*). The most typical way of proving such a formula is to prove *both* P and Q, individually.

[1] We rename x first, if x is already in the variable context.

- P **or** Q (*disjunction*). The most typical way of proving such a formula is to prove *one* of P or Q.
- P **implies** Q (also written **if** P, **then** Q) (*implication*). The most typical way to prove such a formula is to add P to our current assumptions, and then use these assumptions to establish the truth of Q.
- **Not** P (P does not hold; negation). The most typical way to prove such a formula is to show that, if we add P to our current assumptions, we can prove a contradiction—something obviously false.

In formulas involving the connectives **not**, **and** and **or**, we give **not** the highest precedence, **and** the next highest precedence, and **or** the lowest precedence. Because (P **and** Q) **and** R is equivalent to P **and** (Q **and** R), we don't need to use parentheses when combining multiple occurrences of **and**. The same is true for **or**. When we write P **iff** Q ("if and only if"), this means that (P **implies** Q) **and** (Q **implies** P). P **implies** Q is equivalent to **not** P **or** Q.

Universal and existential quantification have lower precedence than the connectives, and extend as far as possible. E.g., **for all** x, **for all** $y, P(x)$ **and** $Q(y)$ means **for all** x, **for all** $y, (P(x)$ **and** $Q(y))$. We can rename quantified variables in formulas, maintaining equivalence, e.g., turning **for all** x, **for all** $y, P(x)$ **and** $Q(y)$ into **for all** y, **for all** $z, P(y)$ **and** $Q(z)$. We often abbreviate nested universal or existential quantifications, writing, e.g., **for all** $x, y, P(x)$ **and** $Q(y)$ instead of **for all** x, **for all** $y, P(x)$ **and** $Q(y)$.

We have that:

- **Not for all** $x, P(x)$, is equivalent to **there exists an** x **such that not** $P(x)$;
- **Not there exists an** x **such that** $P(x)$ is equivalent to **for all** x, **not** $P(x)$;
- **Not**(P **and** Q) is equivalent to **not** P **or not** Q;
- **Not**(P **or** Q) is equivalent to **not** P **and not** Q;
- **Not**(P **implies** Q) is equivalent to P **and not** Q;
- **Not not** P is equivalent to P.

Furthermore:

- If the current assumptions are contradictory, we can prove any conclusion, Q.
- If we have the assumptions P **implies** Q and P, we may conclude Q.
- If we have the assumption **for all** $x, P(x)$, and t is a term whose variables come from the variable context, then we may conclude $P(t)$.

- If one of our assumptions is a disjunction P **or** Q, and we are trying to prove R, then it suffices to:
  - show that R follows from P plus whatever other assumptions we have; and
  - show that R follows from Q plus whatever other assumptions we have.

  This rule is called *disjunction elimination*. It generalizes to when there are more than two disjuncts.

- If one of our assumptions is **there exists an** x **such that** $P(x)$, and we are trying to prove R, it suffices to introduce the assumption that $P(x)$ holds, where the variable x is added to our variable context (and so stands for some element of the value universe)[2], and then establish that R holds. This rule is called *existential elimination*.

- When trying to prove a conclusion P from some set of assumptions, we can suspend this proof, and go about proving a formula Q from the same set of assumptions. If this sub-proof succeeds, we can add Q to our original set of assumptions, and then go on with our proof of P.

An alternative way of proving P **implies** Q is to prove its *contrapositive*: **not** Q **implies not** P.

When we do a *case analysis*, we first prove (this may be obvious, in which case we may leave the proof implicit) that a disjunction P_1 **or** $\cdots$ **or** P_n holds, adding it to our assumptions. We then use disjunction elimination, showing our conclusion follows from each of $P_1, \ldots, P_n$.

Finally, when proving P using *proof by contradiction*, we temporarily add **not** P to our assumptions and then try to derive a contradiction (i.e., prove something that's obviously false). If we succeed, then P follows from our original assumptions.

1.1.2 Describing Sets by Listing Their Elements

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

1.1.3 Sets of Numbers

We write:

- $\mathbb{N}$ for the set $\{0, 1, \ldots\}$ of all natural numbers;

[2] We rename x first, if x is already in the variable context.

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

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

1.1.4 Relationships between Sets

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

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

as an abbreviation for

$$\text{for all } x, x \in X \text{ implies } P(x).$$

And we write

$$\text{there exists an } x \in X, P(x)$$

as an abbreviation for

$$\text{there exists an } x \text{ such that } x \in X \text{ and } P(x).$$

Suppose A and B are sets. We say that:

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

In other words: A is a subset of B iff every everything in A is also in B, and A is a proper subset of B iff everything in A is in B, but there is at least one element of B that is not in A. For example, $\emptyset \subsetneq \mathbb{N}$, $\mathbb{N} \subseteq \mathbb{N}$ and $\mathbb{N} \subsetneq \mathbb{Z}$.

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

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

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

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

Given sets A and B, we say that:

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

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

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

1.1.5 Set Formation

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

For the first example, let

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

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

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

Thus, e.g.,

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

Since $5 \in \mathbb{N}$ and $5^2 = 25 \geq 20$, it follows that $5 \in A$. On the other hand, $5.5 \notin A$, 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 (expression), rather than a variable. The variables n and m are existentially quantified, rather than universally quantified, so that, for all l,

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

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

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

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

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

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

1.1.6 Operations on Sets

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

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

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

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

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

$A - B$ is formed by removing the elements of B from A, if necessary. For example, $\{0, 1, 2\} - \{1, 4\} = \{0, 2\}$. $A \times B$ consists of all ordered pairs (x, y), where x comes from A and y comes from B. For example, $\{0, 1\} \times \{1, 2\} =$

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

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

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

Proposition 1.1.1
Suppose A, B and C are sets.

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1.7 Relations and Functions

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

Consider the relation

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

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

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

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

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

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

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

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

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

A relation R is:

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

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

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

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

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

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

The relation

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

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

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

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

Exercise 1.1.3
Suppose X is a set, and $x \in X$. What are the elements of $\emptyset \to X$, $X \to \emptyset$, $\{x\} \to X$ and $X \to \{x\}$? Prove that your answers are correct.

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 1.1.4
There are three things wrong with the following "definition"—what are they? Let $f \in \mathbb{N} \to \mathbb{N}$ be the unique function such that, for all $n \in \mathbb{N}$,

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

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

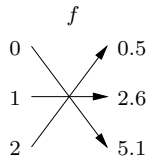
One of the forms of the Axiom of Choice says that, for all sets X of nonempty sets, there is a function f with domain X such that, for all $Y \in X$, $f\,Y \in Y$. In other words, f is a *choice function* that given an element Y of X, is able to pick an element of Y.

1.1.8 Set Cardinality

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

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

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



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

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

is injective, but the function

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

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

It is easy to see that:

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

Exercise 1.1.5
Suppose A and B are sets. Show that for all f, f is a bijection from A to B iff

- f is a function from A to B;
- $\mathbf{range}\, f = B$; and
- f is injective.

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

Exercise 1.1.6
Show that:

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

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

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

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

(1) $A \cong A$;

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

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

We say that a set X is:

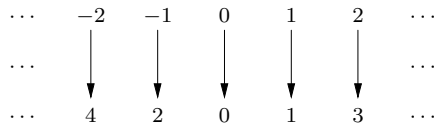
- *finite* iff $X \cong [1:n]$, for some $n \in \mathbb{N}$ (recall that $[1:n]$ is all of the natural numbers that are at least 1 and no more than n, so that $[1:0] = \emptyset$);

- *infinite* iff it is not finite;
- *countably infinite* iff $X \cong \mathbb{N}$;
- *countable* iff X is either finite or countably infinite; and
- *uncountable* iff X is not countable.

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

We have that:

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



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

To prove that $\mathbb{R}$ and $\mathcal{P}\,\mathbb{N}$ are uncountable, we can use an important technique called "diagonalization", which we will see again in Chapter 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. If we can obtain a contradiction, it will follow that $\mathcal{P}\,\mathbb{N}$ is uncountable. Since $\mathcal{P}\,\mathbb{N}$ is not finite, it follows that there is a bijection f from $\mathbb{N}$ to $\mathcal{P}\,\mathbb{N}$. Our plan is to define a subset X of $\mathbb{N}$ such that $X \notin \mathbf{range}\, f$, thus obtaining a contradiction, since this will show that f is not a bijection from $\mathbb{N}$ to $\mathcal{P}\,\mathbb{N}$.

Consider the infinite table in which both the rows and the columns are indexed by the elements of $\mathbb{N}$, listed in ascending order, and where a cell (m, n) (m is the row, n is the column) contains 1 iff $m \in f\, n$, and contains 0 iff $m \notin f\, n$. 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 $j \notin f\, i$.

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 $n \notin f\, n$. This will ensure that, for all $n \in \mathbb{N}$, $f\, n \neq X$. With our example table:

- since $i \in f\, i$, but $i \notin X$, we have that $f\, i \neq X$;

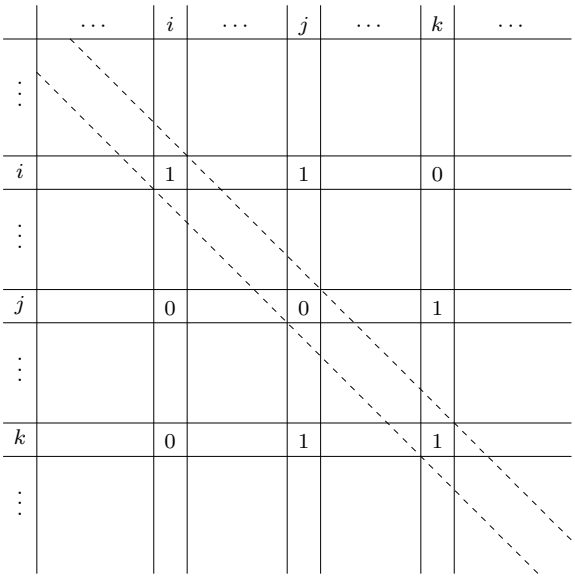


Figure 1.1: Example Diagonalization Table for Cardinality Proof

- since $j \notin f\, j$, but $j \in X$, we have that $f\, j \neq X$; and
- since $k \in f\, k$, but $k \notin X$, we have that $f\, k \neq X$.

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

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

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

- Suppose $n \in X$. By the definition of X, it follows that $n \notin f\, n = X$—contradiction.
- Suppose $n \notin X$. Because $X = f\, n$, we have that $n \notin f\, n$. Thus, since $n \in \mathbb{N}$ and $n \notin f\, n$, it follows that $n \in X$—contradiction.

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

We have seen how bijections may be used to determine whether sets have the same size. But how can we compare the relative sizes of sets, i.e., say whether one set is smaller or larger than another? The answer is to make use of injective

functions. We say that a set X is *no bigger* than a set Y ($X \preceq Y$) iff there is an injection from X to Y, i.e., an injective function whose domain is X and whose range is a subset of Y. Because identity functions are injective, we have that $X \subseteq Y$ implies $X \preceq Y$; e.g., $\mathbb{N} \preceq \mathbb{R}$. This definition makes sense, because X is no bigger than Y iff X has the same size as a subset of Y. We say that X is *strictly smaller* than Y iff $X \preceq Y$ and $X \not\cong Y$.

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

(1) $A \preceq A$;

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

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

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

Exercise 1.1.8
Use the Schröder-Bernstein Theorem to show that $\mathbb{N} \cong \mathbb{N} \times \mathbb{N}$. Hint: use the following consequence of the Fundamental Theorem of Arithmetic: if two finite, ascending (each element is $\leq$ the next) sequences of prime numbers (natural numbers that are at least 2 and have no divisors other than 1 and themselves) have the same product (the product of the empty sequence is 1), then they are equal.

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

1.1.9 Data Structures

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

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

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

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

and

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

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

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

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

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

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

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

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

For example,

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

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

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

Using this fact, it is easy to prove that:

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

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

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

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

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

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

1.1.10 Notes

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

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

1.2 Induction

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

1.2.1 Mathematical Induction

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

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

(basis step)

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

(inductive step)

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

then,

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

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

Next we give an example of a mathematical induction.

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

Proof. We proceed by mathematical induction.

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

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

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

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

□

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

1.2.2 Strong Induction

Next, we consider the principle of strong induction.

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

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

then

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

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

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

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

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

Let the property $Q(n)$ be

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

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

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

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

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

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

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. □

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

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

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

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

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

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

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

We must show that

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

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

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

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

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

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

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

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

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

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

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

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

We must show that

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

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

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

There are two cases to consider.

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

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

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

□

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

1.2.3 Well-founded Induction

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

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

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

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

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

> *for all $x \in A$,*
> *if (†) for all $y \in A$, if $y\ R\ x$, then $P(y)$,*
> *then $P(x)$,*

then

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, let R be the relation on $\mathbb{Z}$ such that, for all $n, m \in \mathbb{Z}$, $n\ R\ m$ iff $|n| < |m|$ (where we're writing $|\cdot|$ for the absolute value of an integer). Since $<$ is well-founded on $\mathbb{N}$, Proposition 1.2.10 tells us that R is well-founded on $\mathbb{Z}$.

If we do a well-founded induction on R, when proving $P(n)$, for $n \in \mathbb{Z}$, we can make use of $P(m)$ for any $m \in \mathbb{Z}$ whose absolute value is strictly less than the absolute value of n. E.g., when proving $P(-10)$, we could make use of $P(5)$ or $P(-9)$.

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

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

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

Proof. Suppose T is a nonempty subset of $A \times B$. We must show that there is an $R \triangleright S$-minimal element of T. By our assumption, T is a relation from A to B. Because T is nonempty, it follows that $\mathbf{domain}\, T$ is a nonempty subset of A. Since R is a well-founded relation on A, it follows that there is an R-minimal $x \in \mathbf{domain}\, T$. Let $Y = \{\, y \in B \mid (x, y) \in T \,\}$. Because Y is a nonempty subset of B, and S is well-founded on B, there exists an S-minimal $y \in Y$. Thus $(x, y) \in T$.

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

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

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

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

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

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

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

1.2.4 Notes

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

1.3 Inductive Definitions and Recursion

In this section, we will introduce and study ordered trees of arbitrary (finite) arity, whose nodes are labeled by elements of some set. In later chapters, we will define regular expressions (in Chapter 3), parse trees (in Chapter 4) and programs (in Chapter 5) as restrictions of the trees we consider here.

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

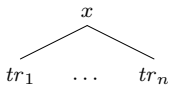
1.3.1 Inductive Definition of Trees

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

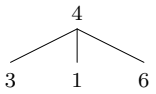
Ignoring the adjective "least" for the moment, some example elements of $\mathbf{Tree}\, \mathbb{N}$ (the case when the set X of tree labels is $\mathbb{N}$) are:

- Since $3 \in \mathbb{N}$ and $[\,] \in \mathbf{List}(\mathbf{Tree}\, \mathbb{N})$, we have that $(3, [\,]) \in \mathbf{Tree}\, \mathbb{N}$. For similar reasons, $(1, [\,])$, $(6, [\,])$ and all pairs of the form $(n, [\,])$, for $n \in \mathbb{N}$, are in $\mathbf{Tree}\, \mathbb{N}$.
- Because $4 \in \mathbb{N}$, and $[(3, [\,]), (1, [\,]), (6, [\,])] \in \mathbf{List}(\mathbf{Tree}\, \mathbb{N})$, we have that $(4, [(3, [\,]), (1, [\,]), (6, [\,])]) \in \mathbf{Tree}\, \mathbb{N}$.
- And we can continue like this forever.

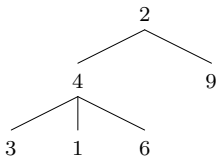
Trees are often easier to comprehend if they are drawn. We draw the X-tree $(x, [tr_1, \ldots, tr_n])$ as



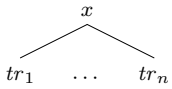
For example,



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



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



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

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

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

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

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

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

In other words, the lemma says that, given any set X, there exists a superset U of X such that every pair of elements of U is already an element of U, and every list of elements of U is already an element of U. Now, suppose X is a set,

and let U be as in the lemma. To see that U is X-closed, suppose $x \in X$ and $trs \in \mathbf{List}\, U$. Thus $x \in U$ and $trs \in U$, so that $(x, trs) \in U$, as required.

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{l}\textit{for all } x \in X \textit{ and } trs \in \mathbf{List}(\mathbf{Tree}\, X),\\ \textit{if } (\dagger)\ \textit{for all } i \in [1 : |trs|],\ P(trs\, i),\\ \textit{then } P((x, trs)),\end{array}$$

then

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

We refer to (†) as the inductive hypothesis.

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

$$\begin{array}{l}(\ddagger)\ \text{for all } x \in X \text{ and } trs \in \mathbf{List}(\mathbf{Tree}\, X),\\ \quad \text{if for all } i \in [1 : |trs|],\ P(trs\, i),\\ \quad \text{then } P((x, trs)).\end{array}$$

We must show that

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

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

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

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

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

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

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

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

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

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

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

1.3.2 Recursion

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

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

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

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

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

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

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

If we can understand F as an algorithm, then we can understand the definition of f as an algorithm. If we call f with an $x \in A$, then F may return an

element of B without consulting its second argument. Alternatively, it may call this second argument with a predecessor of x in R. This is a recursive call of f, which must complete before F continues. Once it does complete, F may make more recursive calls, but must eventually return an element of B.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

as required. □

Here are some examples of well-founded recursion:

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

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

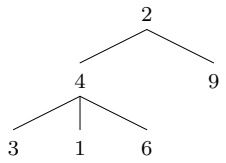
  Thus, if such a definition case-splits according to whether its input is 0 or not, it can be split into two parts:

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

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

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

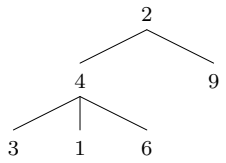
- We may define the *size* of an X-tree $(x, [tr_1, \ldots, tr_n])$ by summing the recursively computed sizes of tr_1, $\ldots$, tr_n, and then adding 1. (When $n = 0$, the sum of no sizes is 0, and so we get 1.) Then, e.g., the size of



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

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

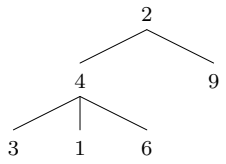
  Then, e.g., the number of leaves of



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

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

  E.g., the height of



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

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

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

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

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

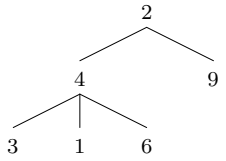
- Given a set X, we can define a well-founded relation $\mathbf{length}_{\mathbf{List}\, X}$ on $\mathbf{List}\, X$ by: for all $xs, ys \in \mathbf{List}\, X$, xs $\mathbf{length}_{\mathbf{List}\, X}$ ys iff $|xs| < |ys|$.

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

1.3.3 Paths in Trees

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

Consider the $\mathbb{N}$-tree



Then:

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

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

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

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

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

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

Let the relation R on Y_X be

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

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

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

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

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

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

1.3.4 Notes

Our treatment of trees, inductive definition, and well-founded recursion is more formal than what one finds in typical books on formal language theory. But those

with a background in set theory will find nothing surprising in this section, and our foundational approach will serve the reader well in later chapters.

Chapter 2

Formal Languages

In this chapter we say what symbols, strings, alphabets and (formal) languages are, show how to use various induction principles to prove language equalities, and give an introduction to the Forlan toolset. In subsequent chapters, we will study four more restricted kinds of languages: the regular (Chapter 3), context-free (Chapter 4), recursive and recursively enumerable (Chapter 5) languages.

2.1 Symbols, Strings, Alphabets and (Formal) Languages

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

2.1.1 Symbols

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

- the comma (",");
- the *digits* 0–9;
- the *letters* a–z and A–Z; and
- the angle brackets ("⟨" and "⟩").

We order **Char** as follows:

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

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

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

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

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

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

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

Thus a symbol never starts or ends with another symbol.

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

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

$$0 < \cdots < 9 < A < \cdots < Z < a < \cdots < z,$$

and, e.g.,

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

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

2.1.2 Strings

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

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

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

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

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

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

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

From our study of lists, we know that:

- Concatenation is associative: for all $x, y, z \in \mathbf{Str}$,

$$(xy)z = x(yz).$$

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

$$\%x = x = x\%.$$

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

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

On the other hand:

Exercise 2.1.1
Disprove the following statement: for all $x, y, x', y' \in \mathbf{Str}$, $xy = x'y'$ iff $x = x'$ and $y = y'$.

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

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

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 2.1.2
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

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

□

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

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

and

$$x^{n+1} = x^n x^1 = x^n x \qquad \text{(Proposition 2.1.2)}.$$

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

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

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

For example:

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

Proposition 2.1.3
For all $x, y, x', y' \in \mathbf{Str}$, $xy = x'y'$ iff either

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

Proof. Straightforward. □

As a consequence of this proposition, we have that:

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

Exercise 2.1.4
Suppose $a \in \mathbf{Sym}$, $x, y \in \mathbf{Str}$. Prove that:

(1) If $a^i x = a^j y$ and neither x nor y begins with a, then $i = j$ and $x = y$;

(2) If $xa^i = ya^j$ and neither x nor y ends with a, then $x = y$ and $i = j$.

Exercise 2.1.5
Suppose $a, b \in \mathbf{Sym}$ and $a \neq b$. Disprove the following statement: for all $i, i', j, j', k, k' \in \mathbb{N}$, $a^i b^j a^k = a^{i'} b^{j'} a^{k'}$ iff $i = i'$, $j = j'$ and $k = k'$.

2.1.3 Alphabets

Having said what symbols and strings are, we now come to alphabets. An *alphabet* is a finite subset of **Sym**. We use Σ (upper case Greek letter sigma) to name alphabets. For example, $\emptyset$, $\{0\}$ and $\{0, 1\}$ 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 recursion on (the length of) strings:

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

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

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

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

2.1.4 Languages

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

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

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

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

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

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

Exercise 2.1.6
Show that $\mathcal{P}(\mathbb{N})$ has the same size as $\mathcal{P}(\{\mathsf{0}\}^*)$.

Given a language L, we write $\mathbf{alphabet}\, L$ for the *alphabet*

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

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

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

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

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

2.1.5 Notes

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

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

2.2 Using Induction to Prove Language Equalities

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

2.2.1 String Induction Principles

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

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

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

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

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

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

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

(basis step)

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

(inductive step)

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

then,

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

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

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

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

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

(basis step)

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

(inductive step)

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

then,

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

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

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

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

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

then,

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

2.2.2 Proving Language Equalities

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

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

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

(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 0's and 1's that can be constructed using (1) and (2). For example,

by (1) and (2), we have that $\mathsf{00} = \mathsf{0\%0} \in X$. Thus, by (2), we have that $\mathsf{1001} = \mathsf{1(00)1} \in X$. 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 $Y = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ is a palindrome and } |w| \text{ is even} \,\}$.

Lemma 2.2.7
$Y \subseteq X$.

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

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

We proceed by strong string induction.

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

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

We must show that

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

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

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

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

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

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

(1)

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

(2)

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

then,

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

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

Lemma 2.2.9
$X \subseteq Y$.

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

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

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

□

Proposition 2.2.10
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.7 and 2.2.9. □

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

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

Then:

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

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

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

(1) $\% \in X$;

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

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

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

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

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

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

Lemma 2.2.11
$X \subseteq Y$.

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

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

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

(3) Suppose $x \in X$, and assume the inductive hypothesis: $x \in Y$. We must show that $0x1 \in Y$. Since $x \in Y$, we have that $0x1 \in \{0,1\}^*$ and $\mathbf{diff}(0x1) = \mathbf{diff}\, 0 + \mathbf{diff}\, x + \mathbf{diff}\, 1 = -1 + 0 + 1 = 0$. Thus $0x1 \in Y$.

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

□

Lemma 2.2.12
$Y \subseteq X$.

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

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

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

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

We must show that

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

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

- Suppose $w = \%$. Then $w = \% \in X$, by Part (1) of the definition of X.
- Suppose $w = 0t$ for some $t \in \{0,1\}^*$. Since $w \in Y$, we have that $-1 + \mathbf{diff}\, t = \mathbf{diff}\, 0 + \mathbf{diff}\, t = \mathbf{diff}(0t) = \mathbf{diff}\, w = 0$, and thus that $\mathbf{diff}\, t = 1$.

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

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

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

$1+\mathbf{diff}\, z = 0+1+\mathbf{diff}\, z = \mathbf{diff}\, y+\mathbf{diff}\, 1+\mathbf{diff}\, z = \mathbf{diff}(y1z) = \mathbf{diff}\, t = 1$, it follows that $\mathbf{diff}\, z = 0$. Thus $z \in Y$.

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

- Suppose $w = 1t$ for some $t \in \{0,1\}^*$. Since $w \in Y$, we have that $1+\mathbf{diff}\, t = \mathbf{diff}\, 1 + \mathbf{diff}\, t = \mathbf{diff}(1t) = \mathbf{diff}\, w = 0$, 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 and z are proper substrings of w, and $y, z \in Y$, the inductive hypothesis tells us that $y, z \in X$. Thus, by Part (4) of the definition of X, we have that $1y0 \in X$. Hence, Part (2) of the definition of X tells us that $w = 1y0z = (1y0)z \in X$.

□

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

Proposition 2.2.13
$X = Y$.

Proof. Follows immediately from Lemmas 2.2.11 and 2.2.12. □

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

(1) $\% \in X$;

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

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

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

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

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

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

(1) $\% \in X$;

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

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

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

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

2.2.3 Notes

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

2.3 Introduction to Forlan

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

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

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

A manual for Forlan is available on the Forlan website, and describes Forlan's modules in considerably more detail than does this book. See the manual for instructions for setting system parameters controlling such things as the search

path used for loading files, the line length used by Forlan's pretty printer, and the number of elements of a list that the Forlan top-level displays.

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

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

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

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

2.3.1 Invoking Forlan

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

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

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

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

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

2.3.2 The SML Core of Forlan

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

```
- 4 + 5;
```

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

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

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

```
- "hello" ^ " " ^ "there";
val it = "hello there" : string
- not true;
val it = false : bool
- true andalso (false orelse true);
val it = true : bool
- if 5 < 7 then "hello" else "bye";
val it = "hello" : string
- (3 + 1, 4 = 4, "a" ^ "b");
val it = (4,true,"ab") : int * bool * string
- #2 it;
val it = true : bool
- [1, 3, 5] @ [7, 9, 11];
val it = [1,3,5,7,9,11] : int list
- rev it;
val it = [11,9,7,5,3,1] : int list
- length it;
val it = 6 : int
- null[];
val it = true : bool
- null[1, 2];
val it = false : bool
- hd[1, 2, 3];
val it = 1 : int
- tl[1, 2, 3];
val it = [2,3] : int list
```

The operator `^` is string concatenation. The conjunction `andalso` evaluates its left-hand side first, and yields `false` without evaluating its right-hand side, if the

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
- val (x, y, z) = (3 + 1, 4 = 4, "a" ^ "b");
val x = 4 : int
val y = true : bool
val z = "ab" : string
- val SOME n = SOME(4 * 25);
stdIn:41.5-41.26 Warning: binding not exhaustive
          SOME n = ...
val n = 100 : int
```

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

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

```
- val x = 3;
```

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

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

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

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

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

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

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

Given functions f and g of types t_1 `->` t_2 and t_2 `->` t_3, respectively, g `o` f is the composition of g and f, the function of type t_1 `->` t_3 that, when given an

argument x of type t_1, evaluates the expression $g(f\ x)$. For example, we have that:

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

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

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

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

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

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

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

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

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

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

```
val it = [true,false,true,false,true] : bool list
```

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

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

```
- val f =
=       let val x = 2 * 10
=       in fn y => y * x end;
val f = fn : int -> int
- f 4;
val it = 80 : int
- f 7;
val it = 140 : int
```

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

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

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

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

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

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

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

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

```
- fun fact n =
=       case n of
```

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

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

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

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

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

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

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

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

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

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

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

2.3.3 Symbols

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

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

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

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

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

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

The module `Sym` also provides the functions

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

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

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

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

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

2.3.4 Sets

The module `Set` defines the abstract type

```
type 'a set
```

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

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

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

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

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

2.3.5 Sets of Symbols

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

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

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

The function `fromList` returns a set with the same elements of the list of symbols it is called with. The function `memb` tests whether a symbol is a member (element) of a set of symbols, `subset` tests whether a first set of symbols is a subset of a second one, and `equal` tests whether two sets of symbols are equal. The functions `union`, `inter` and `minus` compute the union, intersection

and difference of two sets of symbols. The function `genUnion` computes the generalized intersection of a list of sets of symbols *xss*, returning the set of all symbols appearing in at least one element of *xss*. And, the function `genInter` computes the generalized intersection of a nonempty list of sets of symbols *xss*, returning the set of all symbols appearing in all elements of *xss*.

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

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

val it = () : unit
```

2.3.6 Strings

We will be working with two kinds of strings:

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

The module `Str` defines the type `str` of formal language strings, which is bound in the top-level environment, and is equal to `sym list`, the type of lists of

symbols. Because strings are lists, we can use SML's list processing functions on them. Strings are expressed in Forlan's syntax as either a single `%` or a nonempty sequence of symbols.

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

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

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

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

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

```
- val z = Str.fromString "h" @ y;
val z = [-,-,-,-,-] : sym list
- Str.prefix(z, x);
val it = true : bool
- val x = Str.fromString "hellothere";
val x = [-,-,-,-,-,-,-,-,-,-] : str
- null x;
val it = false : bool
- Sym.output("", hd x);
h
val it = () : unit
- Str.output("", tl x);
ellothere
val it = () : unit
- Sym.output("", Str.last x);
e
val it = () : unit
- Str.output("", Str.allButLast x);
hellother
val it = () : unit
```

2.3.7 Sets of Strings

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

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

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

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

```
- val xs = StrSet.input "";
@ hello, <id><num>, %
@ .
val xs = - : str set
- val ys = StrSet.input "";
```

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

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

2.3.8 Relations on Symbols

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

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

```
val input            : string -> sym_rel
val output           : string * sym_rel -> unit
val fromList         : (sym * sym)list -> sym_rel
val memb             : (sym * sym) * sym_rel -> bool
val subset           : sym_rel * sym_rel -> bool
val equal            : sym_rel * sym_rel -> bool
val union            : sym_rel * sym_rel -> sym_rel
val inter            : sym_rel * sym_rel -> sym_rel
val minus            : sym_rel * sym_rel -> sym_rel
val genUnion         : sym_rel list -> sym_rel
val genInter         : sym_rel list -> sym_rel
val domain           : sym_rel -> sym set
val range            : sym_rel -> sym set
val relationFromTo   : sym_rel * sym_set * sym_set -> bool
val reflexive        : sym_rel * sym set -> bool
```

```
val symmetric        : sym_rel -> bool
val antisymmetric    : sym_rel -> bool
val transitive       : sym_rel -> bool
val total            : sym_rel -> bool
val inverse          : sym_rel -> sym_rel
val compose          : sym_rel * sym_rel -> sym_rel
val function         : sym_rel -> bool
val functionFromTo   : sym_rel * sym_set * sym_set -> bool
val injection        : sym_rel -> bool
val bijectionFromTo  : sym_rel * sym_set * sym_set -> bool
val applyFunction    : sym_rel -> sym -> sym
val restrictFunction : sym_rel * sym_set -> sym_rel
val updateFunction   : sym_rel * sym * sym -> sym_rel
```

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

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

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

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

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

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

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

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

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

```
- val rel = SymRel.input "";
@ (1, 2), (2, 3), (3, 4)
@ .
val rel = - : sym_rel
- SymRel.output("", rel);
(1, 2), (2, 3), (3, 4)
val it = () : unit
- SymSet.output("", SymRel.domain rel);
1, 2, 3
val it = () : unit
- SymSet.output("", SymRel.range rel);
2, 3, 4
val it = () : unit
- SymRel.function rel;
val it = true : bool
```

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

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

2.3.9 Notes

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

Chapter 3

Regular Languages

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

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

3.1 Regular Expressions and Languages

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

3.1.1 Operations on Languages

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

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

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

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

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

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

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 identity 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 $L_1 L_2$.

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

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

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

For example, we have that

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

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

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

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

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

and

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

Another useful fact about language exponentiation is:

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

Proof. By mathematical induction on n. □

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

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

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

Thus, for all w,

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

Or, in other words:

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

For example,

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

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

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

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

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

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

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

We assign our operations on languages relative precedences as follows:

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

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

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

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

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

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

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

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

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

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

3.1.2 Regular Expressions

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

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

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

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

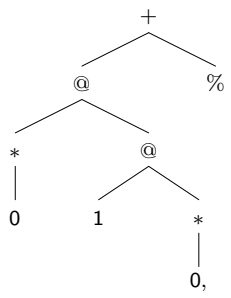
This is yet another example of an inductive definition. The elements of **Reg** are precisely those **RegLab**-trees (trees (See Section 1.3) whose labels come from **RegLab**) that can be built using these six rules.

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

For example,

$$+(@(*(0), @(1, *(0))), \%),$$

i.e.,



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

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

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

It is important that $+$ be the greatest element of **RegLab**; if this were not so, then the definition of weakly simplified regular expressions (see Section 3.3) would have to be altered.

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

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

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

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

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

then

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

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

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

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

Highest: $(\cdot)^*$;

Intermediate: @ (right associative); and

Lowest: + (right associative).

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

For example, we can abbreviate the regular expression

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

to $\mathsf{0}^* \mathbin{@} \mathsf{1} \mathbin{@} \mathsf{0}^* + \%$ or $\mathsf{0}^*\mathsf{1}\mathsf{0}^* + \%$. On the other hand, the regular expression $((\mathsf{0} + \mathsf{1})\mathsf{2})^*$ can't be further abbreviated, since removing either pair of parentheses would result in a different regular expression. Removing the outer pair would result in $(\mathsf{0} + \mathsf{1})(\mathsf{2}^*) = (\mathsf{0} + \mathsf{1})\mathsf{2}^*$, and removing the inner pair would yield $(\mathsf{0} + (\mathsf{12}))^* = (\mathsf{0} + \mathsf{12})^*$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. An easy induction on regular expressions. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now we are able to say what it means for a language to be regular: a language L is *regular* iff $L = L(\alpha)$ for some $\alpha \in \mathbf{Reg}$. We define

$$\begin{aligned}\mathbf{RegLan} &= \{\,L(\alpha) \mid \alpha \in \mathbf{Reg}\,\}\\ &= \{\,L \in \mathbf{Lan} \mid L \text{ is regular}\,\}.\end{aligned}$$

Since every regular expression can be described, e.g., in fully parenthesized form, by a finite sequence of ASCII characters, we can enumerate the regular expressions, and consequently we have that **Reg** is countably infinite. Since $\{0^0\}, \{0^1\}, \{0^2\}, \ldots$, are all regular languages, we have that **RegLan** is infinite. Furthermore, we can establish an injection h from **RegLan** to **Reg**: $h\,L$ is the first (in our enumeration of regular expressions) α such that $L(\alpha) = L$. Because **Reg** is countably infinite, it follows that there is an injection from **Reg** to $\mathbb{N}$. Composing these injections, gives us an injection from **RegLan** to $\mathbb{N}$. And when observing that **RegLan** is infinite, we implicity gave an injection from $\mathbb{N}$

to **RegLan**. Thus, by the Schröder-Bernstein Theorem, we have that **RegLan** and $\mathbb{N}$ have the same size, so that **RegLan** is countably infinite.

Since **RegLan** is countably infinite but **Lan** is uncountable, it follows that **RegLan** $\subsetneq$ **Lan**, i.e., there are non-regular languages. In Section 3.14, we will see a concrete example of a non-regular language.

3.1.3 Processing Regular Expressions in Forlan

Now, we turn to the Forlan implementation of regular expressions. The Forlan module `Reg` defines the abstract type `reg` (in the top-level environment) of regular expressions, as well as various functions and constants for processing regular expressions, including:

```
val input         : string -> reg
val output        : string * reg -> unit
val size          : reg -> int
val numLeaves     : reg -> int
val height        : reg -> int
val emptyStr      : reg
val emptySet      : reg
val fromSym       : sym -> reg
val closure       : reg -> reg
val concat        : reg * reg -> reg
val union         : reg * reg -> reg
val compare       : reg * reg -> order
val equal         : reg * reg -> bool
val fromStr       : str -> reg
val power         : reg * int -> reg
val alphabet      : reg -> sym set
val genConcat     : reg list -> reg
val genUnion      : reg list -> reg
val rightConcat   : reg * reg -> reg
val rightUnion    : reg * reg -> reg
val concatsToList : reg -> reg list
val unionsToList  : reg -> reg list
val sortUnions    : reg -> reg
val allSym        : sym set -> reg
val allStr        : sym set -> reg
val fromStrSet    : str set -> reg
```

The Forlan syntax for regular expressions is the infix/postfix one introduced in the preceding subsection, where $\alpha \mathbin{@} \beta$ is always written as $\alpha\beta$, and we use parentheses to override default precedences/associativities, or simply for clarity. For example, $0^*10^* + \%$ and $(0^*(1(0^*))) + \%$ are the same regular expression. And, $((0^*)1)0^* + \%$ is a different regular expression, but one with the same meaning. Furthermore, $0^*1(0^* + \%)$ is not only different from the two preceding regular expressions, but it has a different meaning (it fails to generate %.) When regular expressions are outputted, as few parentheses as possible are used.

The functions `size`, `numLeaves` and `height` return the size, number of leaves and height, respectively, of a regular expression. The values `emptyStr` and `emptySet` are % and $, respectively. The function `fromSym` takes in a symbol a and returns the regular expression a. It is available in the top-level environment as `symToReg`. The function `closure` takes in a regular expression α and returns $*(\alpha)$. The function `concat` takes a pair (α, β) of regular expressions and returns $@(\alpha, \beta)$. The function `union` takes a pair (α, β) of regular expressions and returns $+(\alpha, \beta)$. The function `compare` implements our total ordering on regular expressions, and `equal` tests whether two regular expressions are equal. The function `fromStr` implements the function **strToReg**, and is also available in the top-level environment as `strToReg`. The function `power` raises a regular expression to a power, and the function `alphabet` returns the alphabet of a regular expression. Finally, the functions `genConcat`, `genUnion`, `rightConcat`, `rightUnion`, `concatsToList`, `unionsToList`, `sortUnions`, `allSym` and `allStr` implement the functions with the same names. The function `fromStrSet` returns $, if called with the empty set. Otherwise, it returns $\texttt{fromStr}\, x_1 + \cdots + \texttt{fromStr}\, x_n$, where $x_1, \ldots, x_n$ are the elements of its argument, listed in strictly ascending order.

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

```
- val reg = Reg.input "";
@ 0*10* + %
@ .
val reg = - : reg
- Reg.size reg;
val it = 9 : int
- Reg.numLeaves reg;
val it = 4 : int
- val reg' = Reg.fromStr(Str.power(Str.input "", 3));
@ 01
@ .
val reg' = - : reg
- Reg.output("", reg');
010101
val it = () : unit
- Reg.size reg';
val it = 11 : int
- Reg.numLeaves reg';
val it = 6 : int
- Reg.compare(reg, reg');
val it = GREATER : order
- val reg'' = Reg.concat(Reg.closure reg, reg');
val reg'' = - : reg
- Reg.output("", reg'');
(0*10* + %)*010101
val it = () : unit
- SymSet.output("", Reg.alphabet reg'');
```

```
0, 1
val it = () : unit
- val reg''' = Reg.power(reg, 3);
val reg''' = - : reg
- Reg.output("", reg''');
(0*10* + %)(0*10* + %)(0*10* + %)
val it = () : unit
- Reg.size reg''';
val it = 29 : int
- Reg.numLeaves reg''';
val it = 12 : int
- Reg.output("", Reg.fromString "(0*(1(0*))) + %");
0*10* + %
val it = () : unit
- Reg.output("", Reg.fromString "(0*1)0* + %");
(0*1)0* + %
val it = () : unit
- Reg.output("", Reg.fromString "0*1(0* + %)");
0*1(0* + %)
val it = () : unit
- Reg.equal
= (Reg.fromString "0*10* + %",
=  Reg.fromString "0*1(0* + %)");
val it = false : bool
```

We can use the functions `genConcat`, `genUnion`, `rightConcat`, `rightUnion`, `concatsToList`, `unionsToList` and `sortUnions` as follows:

```
- Reg.output("", Reg.genConcat nil);
%
val it = () : unit
- Reg.output("", Reg.genUnion nil);
$
val it = () : unit
- val regs =
=       [Reg.fromString "01", Reg.fromString "01 + 12",
=        Reg.fromString "(1 + 2)*", Reg.fromString "3 + 4"];
val regs = [-,-,-,-] : reg list
- Reg.output("", Reg.genConcat regs);
(01)(01 + 12)(1 + 2)*(3 + 4)
val it = () : unit
- Reg.output("", Reg.genUnion regs);
01 + (01 + 12) + (1 + 2)* + 3 + 4
val it = () : unit
- Reg.output
= ("",
=  Reg.rightConcat
=  (Reg.fromString "0123", Reg.fromString "4567"));
01234567
```

```
val it = () : unit
- Reg.output
= ("",
=  Reg.rightUnion
=  (Reg.fromString "0 + 1 + 2", Reg.fromString "1 + 2 + 3"));
0 + 1 + 2 + 1 + 2 + 3
val it = () : unit
- map
= Reg.toString
= (Reg.concatsToList(Reg.fromString "0(12)34"));
val it = ["0","12","3","4"] : string list
- map
= Reg.toString
= (Reg.unionsToList(Reg.fromString "0 + (1 + 2) + 3 + 0"));
val it = ["0","1 + 2","3","0"] : string list
- Reg.output
= ("", Reg.sortUnions(Reg.fromString "12 + 0 + 3 + 0"));
0 + 3 + 12
val it = () : unit
```

We can use the functions `allSym`, `allStr` and `fromStrSet` like this:

```
- Reg.output("", Reg.allSym(SymSet.fromString ""));
$
val it = () : unit
- Reg.output("", Reg.allSym(SymSet.fromString "0"));
0
val it = () : unit
- Reg.output("", Reg.allSym(SymSet.fromString "0, 1, 2"));
0 + 1 + 2
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString ""));
$*
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString "0"));
0*
val it = () : unit
- Reg.output("", Reg.allStr(SymSet.fromString "2, 1, 0"));
(0 + 1 + 2)*
val it = () : unit
- Reg.output
= ("",
=  Reg.fromStrSet(StrSet.fromString "one, two, three, four"));
one + two + four + three
val it = () : unit
```

3.1.4 JForlan

The Java program JForlan can be used to view and edit regular expression trees. It can be invoked directly, or run via Forlan. See the Forlan website for more

information.

3.1.5 Notes

A novel feature of this book is that regular expressions are trees, so that our linear syntax for regular expressions is derived rather than primary. Thus regular expression equality is just tree equality, and it's easy to explain when parentheses are necessary in a linear description of a regular expression. Furthermore, tree-oriented concepts, notation and operations automatically apply to regular expressions, letting us, e.g., give definitions by structural recursion.

3.2 Equivalence and Correctness of Regular Expressions

In this section, we say what it means for regular expressions to be equivalent, show a series of results about regular expression equivalence, and consider how regular expressions may be designed and proved correct.

3.2.1 Equivalence of Regular Expressions

Regular expressions α and β are *equivalent* iff $L(\alpha) = L(\beta)$. In other words, α and β are equivalent iff α and β generate the same language. We define a relation $\approx$ on **Reg** by: $\alpha \approx \beta$ iff α and β are equivalent. For example, $L((\mathsf{00})^* + \%) = L((\mathsf{00})^*)$, and thus $(\mathsf{00})^* + \% \approx (\mathsf{00})^*$.

One approach to showing that $\alpha \approx \beta$ is to show that $L(\alpha) \subseteq L(\beta)$ and $L(\beta) \subseteq L(\alpha)$. The following proposition is useful for showing language inclusions, not just ones involving regular languages.

Proposition 3.2.1

*(1) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 \cup A_2 \subseteq B_1 \cup B_2$.*

*(2) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 \cap A_2 \subseteq B_1 \cap B_2$.*

*(3) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $B_2 \subseteq A_2$, then $A_1 - A_2 \subseteq B_1 - B_2$.*

*(4) For all $A_1, A_2, B_1, B_2 \in$ **Lan**, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then $A_1 A_2 \subseteq B_1 B_2$.*

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

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

In Part (3), note that the second part of the sufficient condition for knowing $A_1 - A_2 \subseteq B_1 - B_2$ is $B_2 \subseteq A_2$, not $A_2 \subseteq B_2$.

Proof. (1) and (2) are straightforward. We show (3) as an example, below. (4) is easy. (5) is proved by mathematical induction, using (4). (6) is proved using (5).

For (3), suppose that $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$, $A_1 \subseteq B_1$ and $B_2 \subseteq A_2$. To show that $A_1 - A_2 \subseteq B_1 - B_2$, suppose $w \in A_1 - A_2$. We must show that $w \in B_1 - B_2$. It will suffice to show that $w \in B_1$ and $w \notin B_2$.

Since $w \in A_1 - A_2$, we have that $w \in A_1$ and $w \notin A_2$. Since $A_1 \subseteq B_1$, it follows that $w \in B_1$. Thus, it remains to show that $w \notin B_2$.

Suppose, toward a contradiction, that $w \in B_2$. Since $B_2 \subseteq A_2$, it follows that $w \in A_2$—contradiction. Thus we have that $w \notin B_2$. □

Next we show that our relation $\approx$ has some of the familiar properties of equality.

Proposition 3.2.2
(1) $\approx$ *is reflexive on* **Reg**, *symmetric and transitive.*

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

(3) For all $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$, *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 $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$, *if* $\alpha_1 \approx \beta_1$ *and* $\alpha_2 \approx \beta_2$, *then* $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$.

Proof. Follows from the properties of $=$. As an example, we show Part (4).

Suppose $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$, and assume that $\alpha_1 \approx \beta_1$ and $\alpha_2 \approx \beta_2$. Then $L(\alpha_1) = L(\beta_1)$ and $L(\alpha_2) = L(\beta_2)$, so that

$$\begin{aligned} L(\alpha_1 + \alpha_2) &= L(\alpha_1) \cup L(\alpha_2) = L(\beta_1) \cup L(\beta_2) \\ &= L(\beta_1 + \beta_2). \end{aligned}$$

Thus $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$. □

A consequence of Proposition 3.2.2 is the following proposition, which says that, if we replace a subtree of a regular expression α by an equivalent regular expression, that the resulting regular expression is equivalent to α.

Proposition 3.2.3
Suppose $\alpha, \beta, \beta' \in \mathbf{Reg}$, $\beta \approx \beta'$, $pat \in \mathbf{Path}$ *is valid for* α, *and* β *is the subtree of* α *at position pat. Let* α' *be the result of replacing the subtree at position pat in* α *by* β'. *Then* $\alpha \approx \alpha'$.

Proof. By induction on α. □

Next, we state and prove some equivalences involving union.

Proposition 3.2.4
(1) For all $\alpha, \beta \in \mathbf{Reg}$, $\alpha + \beta \approx \beta + \alpha$.

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$.

(3) For all $\alpha \in \mathbf{Reg}$, $\$ + \alpha \approx \alpha$.

(4) For all $\alpha \in \mathbf{Reg}$, $\alpha + \alpha \approx \alpha$.

(5) If $L(\alpha) \subseteq L(\beta)$, *then* $\alpha + \beta \approx \beta$.

Proof.

(1) Follows from the commutativity of $\cup$.

(2) Follows from the associativity of $\cup$.

(3) Follows since $\emptyset$ is the identity for $\cup$.

(4) Follows since $\cup$ is idempotent: $A \cup A = A$, for all sets A.

(5) Follows since, if $L_1 \subseteq L_2$, then $L_1 \cup L_2 = L_2$.

□

Next, we consider equivalences for concatenation.

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

(2) For all $\alpha \in \mathbf{Reg}$, $\%\alpha \approx \alpha \approx \alpha\%$.

(3) For all $\alpha \in \mathbf{Reg}$, $\$\alpha \approx \$ \approx \alpha\$$.

Proof.

(1) Follows from the associativity of language concatenation.

(2) Follows since $\{\%\}$ is the identity for language concatenation.

(3) Follows since $\emptyset$ is the zero for language concatenation.

□

Next we consider the distributivity of concatenation over union. First, we prove a proposition concerning languages. Then, we use this proposition to show the corresponding proposition for regular expressions.

Proposition 3.2.6
(1) For all $L_1, L_2, L_3 \in \mathbf{Lan}$, $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$.

(2) For all $L_1, L_2, L_3 \in \mathbf{Lan}$, $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$.

Proof. We show the proof of Part (1); the proof of the other part is similar. Suppose $L_1, L_2, L_3 \in \mathbf{Lan}$. It will suffice to show that

$$L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3).$$

To see that $L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3$, suppose $w \in L_1(L_2 \cup L_3)$. We must show that $w \in L_1L_2 \cup L_1L_3$. By our assumption, $w = xy$ for some $x \in L_1$ and $y \in L_2 \cup L_3$. There are two cases to consider.

- Suppose $y \in L_2$. Then $w = xy \in L_1L_2 \subseteq L_1L_2 \cup L_1L_3$.
- Suppose $y \in L_3$. Then $w = xy \in L_1L_3 \subseteq L_1L_2 \cup L_1L_3$.

To see that $L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3)$, suppose $w \in L_1L_2 \cup L_1L_3$. We must show that $w \in L_1(L_2 \cup L_3)$. There are two cases to consider.

- Suppose $w \in L_1L_2$. Then $w = xy$ for some $x \in L_1$ and $y \in L_2$. Thus $y \in L_2 \cup L_3$, so that $w = xy \in L_1(L_2 \cup L_3)$.
- Suppose $w \in L_1L_3$. Then $w = xy$ for some $x \in L_1$ and $y \in L_3$. Thus $y \in L_2 \cup L_3$, so that $w = xy \in L_1(L_2 \cup L_3)$.

□

Proposition 3.2.7
(1) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$.

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $(\alpha + \beta)\gamma \approx \alpha\gamma + \beta\gamma$.

Proof. Follows from Proposition 3.2.6. Consider, e.g., the proof of Part (1). By Proposition 3.2.6(1), we have that

$$\begin{aligned} L(\alpha(\beta + \gamma)) &= L(\alpha)L(\beta + \gamma) \\ &= L(\alpha)(L(\beta) \cup L(\gamma)) \\ &= L(\alpha)L(\beta) \cup L(\alpha)L(\gamma) \\ &= L(\alpha\beta) \cup L(\alpha\gamma) \\ &= L(\alpha\beta + \alpha\gamma) \end{aligned}$$

Thus $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$. □

Finally, we turn our attention to equivalences for Kleene closure, first stating and proving some results for languages, and then stating and proving the corresponding results for regular expressions.

Proposition 3.2.8
- *For all $L \in \mathbf{Lan}$, $LL^* \subseteq L^*$.*
- *For all $L \in \mathbf{Lan}$, $L^*L \subseteq L^*$.*

Proof. E.g., to see that $LL^* \subseteq L^*$, suppose $w \in LL^*$. Then $w = xy$ for some $x \in L$ and $y \in L^*$. Hence $y \in L^n$ for some $n \in \mathbb{N}$. Thus $w = xy \in LL^n = L^{n+1} \subseteq L^*$. □

Proposition 3.2.9
(1) $\emptyset^* = \{\%\}$.

(2) $\{\%\}^* = \{\%\}$.

(3) For all $L \in \mathbf{Lan}$, $L^*L = LL^*$.

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

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

(6) For all $L_1L_2 \in \mathbf{Lan}$, $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$.

Proof. The six parts can be proven in order using Proposition 3.2.1. All parts but (2), (5) and (6) can be proved without using induction.

As an example, we show the proof of (5). To show that $(L^*)^* = L^*$, it will suffice to show that $(L^*)^* \subseteq L^* \subseteq (L^*)^*$.

To see that $(L^*)^* \subseteq L^*$, we use mathematical induction to show that, for all $n \in \mathbb{N}$, $(L^*)^n \subseteq L^*$.

(Basis Step) We have that $(L^*)^0 = \{\%\} = L^0 \subseteq L^*$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $(L^*)^n \subseteq L^*$. We must show that $(L^*)^{n+1} \subseteq L^*$. By the inductive hypothesis, Proposition 3.2.1(4) and Part (4), we have that $(L^*)^{n+1} = L^*(L^*)^n \subseteq L^*L^* = L^*$.

Now, we use the result of the induction to prove that $(L^*)^* \subseteq L^*$. Suppose $w \in (L^*)^*$. We must show that $w \in L^*$. Since $w \in (L^*)^*$, we have that $w \in (L^*)^n$ for some $n \in \mathbb{N}$. Thus, by the result of the induction, $w \in (L^*)^n \subseteq L^*$.

Finally, for the other inclusion, we have that $L^* = (L^*)^1 \subseteq (L^*)^*$. □

Exercise 3.2.10
Prove Proposition 3.2.9(6), i.e., for all $L_1L_2 \in \mathbf{Lan}$, $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$.

Proposition 3.2.11
(1) $\$^* \approx \%$.

(2) $\%^* \approx \%$.

(3) For all $\alpha \in \mathbf{Reg}$, $\alpha^*\alpha \approx \alpha\,\alpha^*$.

(4) For all $\alpha \in \mathbf{Reg}$, $\alpha^*\alpha^* \approx \alpha^*$.

(5) For all $\alpha \in \mathbf{Reg}$, $(\alpha^*)^* \approx \alpha^*$.

(6) For all $\alpha, \beta \in \mathbf{Reg}$, $(\alpha\beta)^*\alpha \approx \alpha(\beta\alpha)^*$.

Proof. Follows from Proposition 3.2.9. Consider, e.g., the proof of Part (5). By Proposition 3.2.9(5), we have that

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

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

3.2.2 Proving the Correctness of Regular Expressions

In this subsection, we use two examples to show how regular expressions can be designed and proved correct.

For our first example, define a function $\mathbf{zeros} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{N}$ by recursion:

$$\begin{aligned} \mathbf{zeros}\,\% &= 0, \\ \mathbf{zeros}(\mathsf{0}w) &= \mathbf{zeros}\,w + 1, \text{for all } w \in \{\mathsf{0},\mathsf{1}\}^*, \text{ and} \\ \mathbf{zeros}(\mathsf{1}w) &= \mathbf{zeros}\,w, \text{for all } w \in \{\mathsf{0},\mathsf{1}\}^*. \end{aligned}$$

Thus $\mathbf{zeros}\,w$ is the number of occurrences of $\mathsf{0}$ in w. It is easy to show that:

- $\mathbf{zeros}\,\mathsf{0} = 1$;
- $\mathbf{zeros}\,\mathsf{1} = 0$;
- for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{zeros}(xy) = \mathbf{zeros}\,x + \mathbf{zeros}\,y$;
- for all $n \in \mathbb{N}$, $\mathbf{zeros}(\mathsf{0}^n) = n$; and
- for all $n \in \mathbb{N}$, $\mathbf{zeros}(\mathsf{1}^n) = 0$.

Let

$$X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{zeros}\,w \text{ is even} \,\},$$

so that X is all strings of $\mathsf{0}$'s and $\mathsf{1}$'s with an even number of $\mathsf{0}$'s. Clearly, $\% \in X$ and $\{\mathsf{1}\}^* \subseteq X$.

Let's consider the problem of finding a regular expression that generates X. A string with this property would begin with some number of $\mathsf{1}$'s (possibly none). After this, the string would have some number of parts (possibly none), each consisting of a $\mathsf{0}$, followed by some number of $\mathsf{1}$'s, followed by a $\mathsf{0}$, followed by some number of $\mathsf{1}$'s. The above considerations lead us to the regular expression

$$\alpha = \mathsf{1}^*(\mathsf{0}\mathsf{1}^*\mathsf{0}\mathsf{1}^*)^*.$$

To prove $L(\alpha) = X$, it's helpful to give names to the meanings of two parts of α. Let

$$Y = \{\mathsf{0}\}\{\mathsf{1}\}^*\{\mathsf{0}\}\{\mathsf{1}\}^* \quad \text{and} \quad Z = \{\mathsf{1}\}^*Y^*,$$

so that $L(\mathsf{0}\mathsf{1}^*\mathsf{0}\mathsf{1}^*) = Y$ and $L(\alpha) = Z$. Thus it will suffice to prove that $Z = X$, and we do this by showing $Z \subseteq X \subseteq Z$. We begin by showing $Z \subseteq X$.

Lemma 3.2.12
(1) $Y \subseteq X$.

(2) $XX \subseteq X$.

Proof.

(1) Suppose $w \in Y$, so that $w = \mathsf{0}x\mathsf{0}y$ for some $x, y \in \{\mathsf{1}\}^*$. Thus $\mathbf{zeros}\, w = \mathbf{zeros}(\mathsf{0}x\mathsf{0}y) = \mathbf{zeros}\,\mathsf{0} + \mathbf{zeros}\, x + \mathbf{zeros}\,\mathsf{0} + \mathbf{zeros}\, y = 1 + 0 + 1 + 0 = 2$ is even, so that $w \in X$.

(2) Suppose $w \in XX$, so that $w = xy$ for some $x, y \in X$. Then $\mathbf{zeros}\, x$ and $\mathbf{zeros}\, y$ are even, so that $\mathbf{zeros}\, w = \mathbf{zeros}\, x + \mathbf{zeros}\, y$ is even. Hence $w \in X$.

□

Lemma 3.2.13
$Y^* \subseteq X$.

Proof. It will suffice to show that, for all $n \in \mathbb{N}$, $Y^n \subseteq X$, and we show this using mathematical induction.

(Basis Step) We have that $Y^0 = \{\%\} \subseteq X$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $Y^n \subseteq X$. Then $Y^{n+1} = YY^n \subseteq XX \subseteq X$, by Lemma 3.2.12

□

Lemma 3.2.14
$Z \subseteq X$.

Proof. By Lemmas 3.2.12 and 3.2.13, we have that $Z = \{\mathsf{1}\}^*Y^* \subseteq XX \subseteq X$.
□

To prove $X \subseteq Z$, it's helpful to define another language:

$$U = \{\, w \in X \mid \mathsf{1} \text{ is not a prefix of } w \,\}.$$

Lemma 3.2.15
$U \subseteq Y^*$.

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

$$\text{if } w \in U, \text{ then } w \in Y^*.$$

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

$$\text{if } x \in U, \text{ then } x \in Y^*.$$

We must show that

$$\text{if } w \in U, \text{ then } w \in Y^*.$$

Suppose $w \in U$, so that $\mathbf{zeros}\, w$ is even and $\mathsf{1}$ is not a prefix of w. We must show that $w \in Y^*$. If $w = \%$, then $w \in Y^*$. Otherwise, $w = \mathsf{0}x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since $1 + \mathbf{zeros}\, x = \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, x = \mathbf{zeros}(\mathsf{0}x) = \mathbf{zeros}\, w$ is even, we have that $\mathbf{zeros}\, x$ is odd. Let n be the largest element of $\mathbb{N}$ such that $\mathsf{1}^n$ is a prefix of x. (n is well-defined, since $\mathsf{1}^0 = \%$ is a prefix of x.) Thus $x = \mathsf{1}^n y$ for some $y \in \{\mathsf{0},\mathsf{1}\}^*$. Since $\mathbf{zeros}\, y = 0 + \mathbf{zeros}\, y = \mathbf{zeros}\, \mathsf{1}^n + \mathbf{zeros}\, y = \mathbf{zeros}(\mathsf{1}^n y) = \mathbf{zeros}\, x$ is odd, we have that $y \neq \%$. And, by the definition of n, $\mathsf{1}$ is not a prefix of y. Hence $y = \mathsf{0}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Since $1 + \mathbf{zeros}\, z = \mathbf{zeros}\, \mathsf{0} + \mathbf{zeros}\, z = \mathbf{zeros}(\mathsf{0}z) = \mathbf{zeros}\, y$ is odd, we have that $\mathbf{zeros}\, z$ is even. Let m be the largest element of $\mathbb{N}$ such that $\mathsf{1}^m$ is a prefix of z. Thus $z = \mathsf{1}^m u$ for some $u \in \{\mathsf{0},\mathsf{1}\}^*$, and $\mathsf{1}$ is not a prefix of u. Since $\mathbf{zeros}\, u = 0 + \mathbf{zeros}\, u = \mathbf{zeros}\, \mathsf{1}^m + \mathbf{zeros}\, u = \mathbf{zeros}(\mathsf{1}^m u) = \mathbf{zeros}\, z$ is even, it follows that $u \in U$. Summarizing, we have that $w = \mathsf{0}x = \mathsf{0}\mathsf{1}^n y = \mathsf{0}\mathsf{1}^n \mathsf{0}z = \mathsf{0}\mathsf{1}^n \mathsf{0}\mathsf{1}^m u$ and $u \in U$. Since u is a proper substring of w, the inductive hypothesis tells us that $u \in Y^*$. Hence $w = \mathsf{0}\mathsf{1}^n \mathsf{0}\mathsf{1}^m u = (\mathsf{0}\mathsf{1}^n \mathsf{0}\mathsf{1}^m)u \in YY^* \subseteq Y^*$. □

Lemma 3.2.16
$X \subseteq Z$.

Proof. Suppose $w \in X$. Let n be the largest element of $\mathbb{N}$ such that $\mathsf{1}^n$ is a prefix of w. Thus $w = \mathsf{1}^n x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since $w \in X$, we have that $\mathbf{zeros}\, x = 0 + \mathbf{zeros}\, x = \mathbf{zeros}\, \mathsf{1}^n + \mathbf{zeros}\, x = \mathbf{zeros}\, w$ is even, so that $x \in X$. By the definition of n, we have that $\mathsf{1}$ is not a prefix of x, and thus $x \in U$. Hence $w = \mathsf{1}^n x \in \{\mathsf{1}\}^* U \subseteq \{\mathsf{1}\}^* Y^* = Z$, by Lemma 3.2.15. □

By Lemmas 3.2.14 and 3.2.16, we have that $Z \subseteq X \subseteq Z$, completing our proof that α is correct.

Our second example of regular expression design and proof of correction involves the languages

$A = \{\mathsf{001}, \mathsf{011}, \mathsf{101}, \mathsf{111}\}$, and

$B = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \text{for all } x, y \in \{\mathsf{0},\mathsf{1}\}^*, \text{ if } w = x\mathsf{0}y, \text{then there is a } z \in A$
such that z is a prefix of y $\}$.

The elements of A can be thought of as the odd numbers between 1 and 7, expressed in binary, and B consists of those strings of $\mathsf{0}$'s and $\mathsf{1}$'s in which every occurrence of $\mathsf{0}$ is immediately followed by an element of A.

E.g., $\% \in B$, since the empty string has no occurrences of $\mathsf{0}$, and $\mathsf{00111}$ is in B, since its first $\mathsf{0}$ is followed by $\mathsf{011}$ and its second $\mathsf{0}$ is followed by $\mathsf{111}$. But $\mathsf{0000111}$ is not in B, since its first $\mathsf{0}$ is followed by $\mathsf{000}$, which is not in A. And $\mathsf{011}$ is not in B, since $|11| < 3$.

Note that, for all $x, y \in B$, $xy \in B$, i.e., $BB \subseteq B$. This holds, since: each occurrence of $\mathsf{0}$ in x is followed by an element of A in x, and is thus followed by the same element of A in xy; and each occurrence of $\mathsf{0}$ in y is followed by an element of A in y, and is thus followed by the same element of A in xy.

Furthermore, for all strings x, y, if $xy \in B$, then y is in B, i.e., every suffix of an element of B is also in B. This holds since if there was an occurrence of $\mathsf{0}$ in y that wasn't followed by an element of A, then this same occurrence of $\mathsf{0}$ in the suffix y of xy would also not be followed by an element of A, contradicting $xy \in B$.

How should we go about finding a regular expression α such that $L(\alpha) = B$? Because $\% \in B$, for all $x, y \in B$, $xy \in B$, and for all strings x, y, if $xy \in B$ then $y \in B$, our regular expression can have the form β^*, where β generates all the strings that are *basic* in the sense that they are nonempty elements of B with no non-empty proper prefixes that are in B.

Let's try to understand what the basic strings look like. Clearly, $\mathsf{1}$ is basic, so there will be no more basic strings that begin with $\mathsf{1}$. But what about the basic strings beginning with $\mathsf{0}$? No sequence of $\mathsf{0}$'s is basic, and any string that begins with four or more $\mathsf{0}$'s will not be basic. It is easy to see that $\mathsf{000111}$ is basic. In fact, it is the only basic string of the form $\mathsf{000}u$. (The first $\mathsf{0}$ forces u to begin with $\mathsf{1}$, the second $\mathsf{0}$ forces u to continue with $\mathsf{1}$, and the third forces u to continue with $\mathsf{1}$. And, if $|u| > 3$, then the overall string would have a nonempty, proper prefix in B, and so wouldn't be basic.) Similarly, $\mathsf{00111}$ is the only basic string beginning with $\mathsf{001}$. But what about the basic strings beginning with $\mathsf{01}$? It's not hard to see that there are infinitely many such strings: $\mathsf{0111}$, $\mathsf{010111}$, $\mathsf{01010111}$, $\mathsf{0101010111}$, etc. Fortunately, there is a simple pattern here: we have all strings of the form $\mathsf{0}(\mathsf{10})^n\mathsf{111}$ for $n \in \mathbb{N}$.

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

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

But, using some of the equivalences we learned about above, we can turn this regular expression into

$$(\mathsf{1} + \mathsf{0}(\mathsf{0} + \mathsf{00} + (\mathsf{10})^*)\mathsf{111})^*,$$

which we take as our α. Now, we prove that $L(\alpha) = B$.

Let

$$X = \{\mathsf{0}\} \cup \{\mathsf{00}\} \cup \{\mathsf{10}\}^* \quad \text{and} \quad Y = \{\mathsf{1}\} \cup \{\mathsf{0}\}X\{\mathsf{111}\}.$$

Then, we have that

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

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

To begin with, we would like to use mathematical induction to prove that, for all $n \in \mathbb{N}$, $\{0\}\{10\}^n\{111\} \subseteq B$. But in order for the inductive step to succeed, we must prove something stronger. Let

$$C = \{\, w \in B \mid 01 \text{ is a prefix of } w \,\}.$$

Lemma 3.2.17
For all $n \in \mathbb{N}$, $\{0\}\{10\}^n\{111\} \subseteq C$.

Proof. We proceed by mathematical induction.

(Basis Step) Since 01 is a prefix of 0111, and $0111 \in B$, we have that $0111 \in C$. Hence $\{0\}\{10\}^0\{111\} = \{0\}\{\%\}\{111\} = \{0\}\{111\} = \{0111\} \subseteq C$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $\{0\}\{10\}^n\{111\} \subseteq C$. We must show that $\{0\}\{10\}^{n+1}\{111\} \subseteq C$. Since

$$\begin{aligned} \{0\}\{10\}^{n+1}\{111\} &= \{0\}\{10\}\{10\}^n\{111\} \\ &= \{01\}\{0\}\{10\}^n\{111\} \\ &\subseteq \{01\}C \qquad \text{(inductive hypothesis)}, \end{aligned}$$

it will suffice to show that $\{01\}C \subseteq C$. Suppose $w \in \{01\}C$. We must show that $w \in C$. We have that $w = 01x$ for some $x \in C$. Thus w begins with 01. It remains to show that $w \in B$. Since $x \in C$, we have that x begins with 01. Thus the first occurrence of 0 in $w = 01x$ is followed by $101 \in A$. Furthermore, any other occurrence of 0 in $w = 01x$ is within x, and so is followed by an element of A because $x \in C \subseteq B$. Thus $w \in B$.

□

Lemma 3.2.18
$Y \subseteq B$.

Proof. Suppose $w \in Y$. We must show that $w \in B$. If $w = 1$, then $w \in B$. Otherwise, we have that $w = 0x111$ for some $x \in X$. There are three cases to consider.

- Suppose $x = 0$. Then $w = 00111$ is in B.
- Suppose $x = 00$. Then $w = 000111$ is in B.
- Suppose $x \in \{10\}^*$. Then $x \in \{10\}^n$ for some $n \in \mathbb{N}$. By Lemma 3.2.17, we have that $w = 0x111 \in \{0\}\{10\}^n\{111\} \subseteq C \subseteq B$.

□

Lemma 3.2.19
$Y^* \subseteq B$.

Proof. It will suffice to show that, for all $n \in \mathbb{N}$, $Y^n \subseteq B$, and we proceed by mathematical induction.

(Basis Step) Since $\% \in B$, we have that $Y^0 = \{\%\} \subseteq B$.

(Inductive Step) Suppose $n \in \mathbb{N}$, and assume the inductive hypothesis: $Y^n \subseteq B$. Then $Y^{n+1} = YY^n \subseteq BB \subseteq B$, by Lemma 3.2.18 and the inductive hypothesis.

□

Lemma 3.2.20
$B \subseteq Y^*$.

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

$$\text{if } w \in B, \text{ then } w \in Y^*.$$

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

$$\text{if } x \in B, \text{ then } x \in Y^*.$$

We must show that

$$\text{if } w \in B, \text{ then } w \in Y^*.$$

Suppose $w \in B$. We must show that $w \in Y^*$. There are three main cases to consider.

- Suppose $w = \%$. Then $w \in \{\%\} = Y^0 \subseteq Y^*$.
- Suppose $w = 0x$ for some $x \in \{0,1\}^*$. Since $w \in B$, the first 0 of w must be followed by an element of A. Hence $x \neq \%$, so that there are two cases to consider.
  - 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 $y = 0z$ 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 $w = (000111)t \in YY^* \subseteq Y^*$.

* Suppose $y = 1z$ for some $z \in \{0,1\}^*$. Thus $w = 00y = 001z$. Since the first 0 in $001z = w$ is followed by an element of A, and the only element of A that begins with 01 is 011, we have that $z = 1u$ for some $u \in \{0,1\}^*$. Thus $w = 0011u$. Since the second 0 in $0011u = w$ is followed by an element of A, and 111 is the only element of A that begins with 11, we have that $u = 1v$ for some $v \in \{0,1\}^*$. Thus $w = 00111v$. Since $0 \in X$, we have that $00111 = (0)(0)(111) \in \{0\}X\{111\} \subseteq Y$. Because v is a suffix of w, it follows that $v \in B$. Thus the inductive hypothesis tells us that $v \in Y^*$. Hence $w = (00111)v \in YY^* \subseteq Y^*$.

– Suppose $x = 1y$ for some $y \in \{0,1\}^*$. Thus $w = 0x = 01y$. Since $w \in B$, we have that $y \neq \%$. Thus, there are two cases to consider.

* Suppose $y = 0z$ for some $z \in \{0,1\}^*$. Thus $w = 010z$. Let u be the longest prefix of z that is in $\{10\}^*$. (Since $\%$ is a prefix of z and is in $\{10\}^*$, it follows that u is well-defined.) Let $v \in \{0,1\}^*$ be such that $z = uv$. Thus $w = 010z = 010uv$.
  Suppose, toward a contradiction, that v begins with 10. Then $u10$ is a prefix of $z = uv$ that is longer than u. Furthermore $u10 \in \{10\}^*\{10\} \subseteq \{10\}^*$, contradicting the definition of u. Thus we have that v does not begin with 10.
  Next, we show that $010u$ ends with 010. Since $u \in \{10\}^*$, we have that $u \in \{10\}^n$ for some $n \in \mathbb{N}$. There are three cases to consider.
  - Suppose $n = 0$. Since $u \in \{10\}^0 = \{\%\}$, we have that $u = \%$. Thus $010u = 010$ ends with 010.
  - Suppose $n = 1$. Since $u \in \{10\}^1 = \{10\}$, we have that $u = 10$. Hence $010u = 01010$ ends with 010.
  - Suppose $n \geq 2$. Then $n - 2 \geq 0$, so that $u \in \{10\}^{(n-2)+2} = \{10\}^{n-2}\{10\}^2$. Hence u ends with 1010, showing that $010u$ ends with 010.

  Summarizing, we have that $w = 010uv$, $u \in \{10\}^*$, $010u$ ends with 010, and v does not begin with 10. Since the second-to-last 0 in $010u$ is followed in w by an element of A, and 101 is the only element of A that begins with 10, we have that $v = 1s$ for some $s \in \{0,1\}^*$. Thus $w = 010u1s$, and $010u1$ ends with 0101. Since the second-to-last symbol of $010u1$ is a 0, we have that

$s \neq \%$. Furthermore, s does not begin with $\mathsf{0}$, since, if it did, then $v = \mathsf{1}s$ would begin with $\mathsf{10}$. Thus we have that $s = \mathsf{1}t$ for some $t \in \{\mathsf{0},\mathsf{1}\}^*$. Hence $w = \mathsf{010}u\mathsf{11}t$. Since $\mathsf{010}u\mathsf{11}$ ends with $\mathsf{011}$, it follows that the last $\mathsf{0}$ in $\mathsf{010}u\mathsf{11}$ must be followed in w by an element of A. Because $\mathsf{111}$ is the only element of A that begins with $\mathsf{11}$, we have that $t = \mathsf{1}r$ for some $r \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{010}u\mathsf{111}r$. Since $(\mathsf{10})u \in \{\mathsf{10}\}\{\mathsf{10}\}^* \subseteq \{\mathsf{10}\}^* \subseteq X$, we have that $\mathsf{010}u\mathsf{111} = (\mathsf{0})((\mathsf{10})u)\mathsf{111} \in \{\mathsf{0}\}X\{\mathsf{111}\} \subseteq Y$. Since r is a suffix of w, it follows that $r \in B$. Thus, the inductive hypothesis tells us that $r \in Y^*$. Hence $w = (\mathsf{010}u\mathsf{111})r \in YY^* \subseteq Y^*$.

- Suppose $y = \mathsf{1}z$ for some $z \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{011}z$. Since the first $\mathsf{0}$ of w is followed by an element of A, and $\mathsf{111}$ is the only element of A that begins with $\mathsf{11}$, we have that $z = \mathsf{1}u$ for some $u \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $w = \mathsf{0111}u$. Since $\% \in \{\mathsf{10}\}^* \subseteq X$, we have that $\mathsf{0111} = (\mathsf{0})(\%)(\mathsf{111}) \in \{\mathsf{0}\}X\{\mathsf{111}\} \subseteq Y$. Because u is a suffix of w, it follows that $u \in B$. Thus, since u is a proper substring of w, the inductive hypothesis tells us that $u \in Y^*$. Hence $w = (\mathsf{0111})u \in YY^* \subseteq Y^*$.

- Suppose $w = \mathsf{1}x$ for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Since x is a suffix of w, we have that $x \in B$. Because x is a proper substring of w, the inductive hypothesis tells us that $x \in Y^*$. Thus $w = \mathsf{1}x \in YY^* \subseteq Y^*$.

□

By Lemmas 3.2.19 and 3.2.20, we have that $Y^* \subseteq B \subseteq Y^*$, so that $Y^* = B$. This completes our regular expression design and proof of correctness example.

Exercise 3.2.21
Let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathsf{010} \text{ is not a substring of } w \,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

Exercise 3.2.22
Define $\mathbf{diff} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{Z}$ as in Section 2.2, so that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

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

Thus $\mathbf{diff}\, \% = 0$, $\mathbf{diff}\, \mathsf{0} = -1$, $\mathbf{diff}\, \mathsf{1} = 1$, and for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\, x + \mathbf{diff}\, y$. Let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{diff}\, w = 3m, \text{ for some } m \in \mathbb{Z} \,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

Exercise 3.2.23
Define a function $\mathbf{diff} \in \{\mathsf{0},\mathsf{1}\}^* \to \mathbb{Z}$ by: for all $w \in \{\mathsf{0},\mathsf{1}\}^*$,

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

Thus $\mathbf{diff}\, w = 0$ iff w has twice as many $\mathsf{1}$'s as $\mathsf{0}$'s. Furthermore $\mathbf{diff}\, \% = 0$, $\mathbf{diff}\, \mathsf{0} = 1$, $\mathbf{diff}\, \mathsf{1} = -2$, and, for all $x, y \in \{\mathsf{0},\mathsf{1}\}^*$, $\mathbf{diff}(xy) = \mathbf{diff}\, x + \mathbf{diff}\, y$. Let $X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid \mathbf{diff}\, w = 0$ and, for all prefixes v of w, $0 \leq \mathbf{diff}\, v \leq 3\,\}$. Find a regular expression α such that $L(\alpha) = X$, and prove that your answer is correct.

3.2.3 Notes

Our approach in this section is somewhat more formal than is common, but is otherwise standard.

3.3 Simplification of Regular Expressions

In this section, we give three algorithms—of increasing power, but decreasing efficiency—for regular expression simplification. The first algorithm—weak simplification—is defined via a straightforward structural recursion, and is sufficient for many purposes. The remaining two algorithms—local simplification and global simplification—are based on a set of simplification rules that is still incomplete and evolving.

3.3.1 Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (\mathsf{00}^*\mathsf{11}^*)^*, \text{ and}$$
$$\beta = \% + \mathsf{0}(\mathsf{0} + \mathsf{11}^*\mathsf{0})^*\mathsf{11}^*.$$

Although the size of β (18) is strictly greater than the size of α (10), β has only one closure inside another closure, whereas α has two closures inside its outer closure, and thus there is a sense in which β is easier to understand than α.

The standard measure of the closure-related complexity of a regular expression is its *star-height*: the maximum number $n \in \mathbb{N}$ such that there is a path from the root of the regular expression to one of its leaves that passes through n closures. But α and β both have star-heights of 2. Furthermore, star-height isn't respected by the ways of forming regular expressions. E.g., if γ_1 has strictly smaller star-height than γ_2, we can't conclude that $\gamma_1\gamma'$ has strictly smaller star-height than $\gamma_2\gamma'$, as the star height of γ' may be greater than the star-height of γ_2.

So, we need a better measure of the closure-related complexity of regular expressions than star-height. Toward that end, let's define a *closure complexity* to be a nonempty list ns of natural numbers that is (not-necessarily strictly)

descending: for all $i \in [1 : |ns|-1]$, $ns\,i \geq ns(i+1)$. This is a way of representing nonempty multisets of natural numbers that makes it easy to define the usual ordering on multisets. We write **CC** for the set of all closure complexities. E.g., $[3,2,2,1]$ is a closure complexity, but $[3,2,3]$ and $[\,]$ are not. For all $n \in \mathbb{N}$, $[n]$ is a *singleton* closure complexity. The *union* of closure complexities ns and ms ($ns \cup ms$) is the closure complexity that results from putting ns @ ms in descending order, keeping any duplicate elements. (Here we are overloading the term union and the operation $\cup$, but the set-theoretic union isn't an operation on closure complexities, and so no confusion should result.) E.g., $[3,2,2,1] \cup [4,2,1,0] = [4,3,2,2,2,1,1,0]$. The *successor* $\overline{ns}$ of a closure complexity ns is the closure complexity formed by adding one to each element of ns, maintaining the order of the elements. E.g., $\overline{[3,2,2,1]} = [4,3,3,2]$.

It is easy to see that $\cup$ is commutative and associative on **CC**, and that the successor operation on **CC** preserves union:

Proposition 3.3.1

*(1) For all $ns, ms \in$ **CC**, $ns \cup ms = ms \cup ns$.*

*(2) For all $ns, ms, ls \in$ **CC**, $(ns \cup ms) \cup ls = ns \cup (ms \cup ls)$.*

*(3) For all $ns, ms \in$ **CC**, $\overline{ns \cup ms} = \overline{ns} \cup \overline{ms}$.*

Proposition 3.3.2

*(1) For all $ns, ms \in$ **CC**, $\overline{ns} = \overline{ms}$ iff $ns = ms$.*

*(2) For all $ns, ms, ls \in$ **CC**, $ns \cup ls = ms \cup ls$ iff $ns = ms$.*

We define a relation $<_{cc}$ on **CC** by: for all $ns, ms \in$ **CC**, $ns <_{cc} ms$ iff either:

- $ms = ns$ @ ls for some $ls \in$ **CC**; or
- there is an $i \in \mathbb{N} - \{0\}$ such that
  - $i \leq |ns|$ and $i \leq |ms|$,
  - for all $j \in [1 : i-1]$, $ns\,j = ms\,j$, and
  - $ns\,i < ms\,i$.

In other words, $ns <_{cc} ms$ iff either ms consists of the result of appending a nonempty list at the end of ns, or ns and ms agree up to some point, at which ns's value is strictly smaller than ms's value. E.g., $[2,2] <_{cc} [2,2,1]$ and $[2,1,1,0,0] <_{cc} [2,2,1]$. We define the relation $\leq_{cc}$ on **CC** by: for all $ns, ms \in$ **CC**, $ns \leq_{cc} ms$ iff $ns <_{cc} ms$ or $ns = ms$.

Proposition 3.3.3

*(1) For all $ns, ms \in$ **CC**, $\overline{ns} <_{cc} \overline{ms}$ iff $ns <_{cc} ms$.*

(2) For all $ns, ms, ls \in \mathbf{CC}$, $ns \cup ls <_{cc} ms \cup ls$ *iff* $ns <_{cc} ms$.

(3) For all $ns, ms \in \mathbf{CC}$, $ns <_{cc} ns \cup ms$.

Proposition 3.3.4
$<_{cc}$ *is a strict total ordering on* $\mathbf{CC}$.

Proposition 3.3.5
$<_{cc}$ *is a well-founded relation on* $\mathbf{CC}$.

Now we can define the closure complexity of a regular expression. Define the function $\mathbf{cc} \in \mathbf{Reg} \to \mathbf{CC}$ by structural recursion:

$$\begin{aligned}
\mathbf{cc}\,\% &= [0];\\
\mathbf{cc}\,\$ &= [0];\\
\mathbf{cc}\,a &= [0], \text{ for all } a \in \mathbf{Sym};\\
\mathbf{cc}(*(\alpha)) &= \overline{\mathbf{cc}\,\alpha}, \text{ for all } \alpha \in \mathbf{Reg};\\
\mathbf{cc}(@(\alpha,\beta)) &= \mathbf{cc}\,\alpha \cup \mathbf{cc}\,\beta, \text{ for all } \alpha,\beta \in \mathbf{Reg}; \text{ and}\\
\mathbf{cc}(+(\alpha,\beta)) &= \mathbf{cc}\,\alpha \cup \mathbf{cc}\,\beta, \text{ for all } \alpha,\beta \in \mathbf{Reg}.
\end{aligned}$$

We say that $\mathbf{cc}\,\alpha$ is *the closure complexity of* α. E.g.,

$$\begin{aligned}
\mathbf{cc}((12^*)^*) &= \overline{\mathbf{cc}(12^*)} = \overline{\mathbf{cc}\,1 \cup \mathbf{cc}(2^*)} = \overline{[0] \cup \overline{\mathbf{cc}\,2}}\\
&= \overline{[0] \cup \overline{[0]}} = \overline{[0] \cup [1]} = \overline{[1,0]} = [2,1].
\end{aligned}$$

In other words, the $\mathbf{cc}\,\alpha$ can be computed by first collecting together all the paths through α that terminate in leafs, then counting the numbers of closures visited when following each of these paths, and finally putting those sums in descending order.

Returning to our initial examples, we have that $\mathbf{cc}((00^*11^*)^*) = [2,2,1,1]$ and $\mathbf{cc}(\% + 0(0 + 11^*0)^*11^*) = [2,1,1,1,1,0,0,0]$. Since $[2,1,1,1,1,0,0,0] <_{cc} [2,2,1,1]$, the closure complexity of $\% + 0(0 + 11^*0)^*11^*$ is strictly smaller than the closure complexity of $(00^*11^*)^*$.

Proposition 3.3.6
For all $\alpha \in \mathbf{Reg}$, $|\mathbf{cc}\,\alpha| = \mathbf{numLeaves}\,\alpha$.

Proof. An easy induction on regular expressions. □

Exercise 3.3.7
Find regular expressions α and β such that $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ but $\mathbf{size}\,\alpha \neq \mathbf{size}\,\beta$.

In contrast to star-height, closure complexity is compatible with the ways of forming regular expressions. In fact, we can prove even stronger results.

Proposition 3.3.8

(1) For all $\alpha \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha^*) = \mathbf{cc}(\beta^*)$.

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha\gamma) = \mathbf{cc}(\beta\gamma)$.

(3) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\gamma\alpha) = \mathbf{cc}(\gamma\beta)$.

(4) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha + \gamma) = \mathbf{cc}(\beta + \gamma)$.

(5) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\gamma + \alpha) = \mathbf{cc}(\gamma + \beta)$.

Proof. Follows by Proposition 3.3.2. □

The following proposition says that if we replace a subtree of a regular expression by a regular expression with the same closure complexity, then the closure complexity of the resulting, whole regular expression will be unchanged.

Proposition 3.3.9

Suppose $\alpha, \beta, \beta' \in \mathbf{Reg}$, $\mathbf{cc}\,\beta = \mathbf{cc}\,\beta'$, *pat* $\in \mathbf{Path}$ *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* $\mathbf{cc}\,\alpha = \mathbf{cc}\,\alpha'$.

Proof. By induction on α using Proposition 3.3.8. □

Proposition 3.3.10

(1) For all $\alpha \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha^*) <_{cc} \mathbf{cc}(\beta^*)$.

(2) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha\gamma) <_{cc} \mathbf{cc}(\beta\gamma)$.

(3) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\gamma\alpha) <_{cc} \mathbf{cc}(\gamma\beta)$.

(4) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\alpha + \gamma) <_{cc} \mathbf{cc}(\beta + \gamma)$.

(5) For all $\alpha, \beta, \gamma \in \mathbf{Reg}$, $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$ *iff* $\mathbf{cc}(\gamma + \alpha) <_{cc} \mathbf{cc}(\gamma + \beta)$.

Proof. Follows by Proposition 3.3.3. □

The following proposition says that if we replace a subtree of a regular expression by a regular expression with strictly smaller closure complexity, that the resulting, whole regular expression will have strictly smaller closure complexity than the original regular expression.

Proposition 3.3.11

Suppose $\alpha, \beta, \beta' \in \mathbf{Reg}$, $\mathbf{cc}\,\beta' <_{cc} \mathbf{cc}\,\beta$, *pat* $\in \mathbf{Path}$ *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* $\mathbf{cc}\,\alpha' <_{cc} \mathbf{cc}\,\alpha$.

Proof. By induction on α, using Proposition 3.3.10. □

When judging the relative complexity of regular expressions α and β, we will first look at how their closure complexities are related. And, when their closure complexities are equal, we will look at how their sizes are related. To finish explaining how we will judge the relative complexity of regular expressions, we need three definitions.

The function

$$\mathbf{numConcats} \in \mathbf{Reg} \to \mathbb{N}$$

is defined by recursion:

$$\begin{aligned}
\mathbf{numConcats}\,\% &= 0;\\
\mathbf{numConcats}\,\$ &= 0;\\
\mathbf{numConcats}\,a &= 0, \text{ for all } a \in \mathbf{Sym};\\
\mathbf{numConcats}(\alpha^*) &= \mathbf{numConcats}\,\alpha, \text{ for all } \alpha \in \mathbf{Reg};\\
\mathbf{numConcats}(\alpha\beta) &= 1 + \mathbf{numConcats}\,\alpha + \mathbf{numConcats}\,\beta; \text{ and}\\
\mathbf{numConcats}(\alpha + \beta) &= \mathbf{numConcats}\,\alpha + \mathbf{numConcats}\,\beta.
\end{aligned}$$

Thus $\mathbf{numConcats}\,\alpha$ is the number of concatenations in α, i.e., the number of subtrees of α that are concatenations, where a given concatenation may occur (and will be counted) multiple times. E.g., $\mathbf{numConcats}(((\mathsf{01})^*(\mathsf{01}))^*) = 3$. The function

$$\mathbf{numSyms} \in \mathbf{Reg} \to \mathbb{N}$$

is defined by structural recursion:

$$\begin{aligned}
\mathbf{numSyms}\,\% &= 0;\\
\mathbf{numSyms}\,\$ &= 0;\\
\mathbf{numSyms}\,a &= 1, \text{ for all } a \in \mathbf{Sym};\\
\mathbf{numSyms}(\alpha^*) &= \mathbf{numSyms}\,\alpha, \text{ for all } \alpha \in \mathbf{Reg};\\
\mathbf{numSyms}(\alpha\beta) &= \mathbf{numSyms}\,\alpha + \mathbf{numSyms}\,\beta; \text{ and}\\
\mathbf{numSyms}(\alpha + \beta) &= \mathbf{numSyms}\,\alpha + \mathbf{numSyms}\,\beta.
\end{aligned}$$

Thus $\mathbf{numSyms}\,\alpha$ is the number of occurrences of symbols in α, where a given symbol may occur (and will be counted) more than once. E.g., $\mathbf{numSyms}((\mathsf{0}^*\mathsf{1}) + \mathsf{0}) = 3$.

Finally, we say that a regular expression α is *standardized* iff none of α's subtrees have any of the following forms:

- $(\beta_1 + \beta_2) + \beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping unions to the right);

- $\beta_1 + \beta_2$, where $\beta_1 > \beta_2$, or $\beta_1 + (\beta_2 + \beta_3)$, where $\beta_1 > \beta_2$ (it's pleasing if the regular expressions appear in order (recall that unions are greater than all other kinds of regular expressions));
- $(\beta_1\beta_2)\beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping concatenations to the right); and
- $\beta^*\beta$, $\beta^*(\beta\gamma)$, $(\beta_1\beta_2)^*\beta_1$ or $(\beta_1\beta_2)^*\beta_1\gamma$ (moving closures to the right makes a regular expression easier to understand/process from left-to-right).

Thus every subtree of a standardized regular expression will be standardized.

Returning to our assessment of regular expression complexity, suppose that α and β are regular expressions generating %. Then $(\alpha\beta)^*$ and $(\alpha + \beta)^*$ are equivalent, but will will prefer the latter over the former, because unions are generally more amenable to understanding and processing than concatenations. Consequently, when two regular expression have the same closure complexity and size, we will judge their relative complexity according to their numbers of concatenations.

Next, consider the regular expressions $\mathsf{0} + \mathsf{01}$ and $\mathsf{0}(\% + \mathsf{1})$. These regular expressions have the same closure complexity $[0, 0, 0]$, size (5) and number of concatenations (1). We would like to consider the latter to be simpler than the former, since in general we would like to prefer $\alpha(\%+\beta)$ over $\alpha+\alpha\beta$. And we can base this preference on the fact that the number of symbols of $\mathsf{0}(\% + \mathsf{1})$ (2) is one less than the number of symbols of $\mathsf{0} + \mathsf{01}$. When regular expressions have the same closure complexity, size and number of concatenations, the one with fewer symbols is likely to be easier to understand and process. Thus, when regular expressions have identical closure complexity, size and number of concatenations, we will use their relative numbers of symbols to judge their relative complexity.

Finally, when regular expressions have the same closure complexity, size, number of concatenations, and number of symbols, we will judge their relative complexity according to whether they are standardized, thinking that a standardized regular expression is simpler than one that is not standardized.

We define a relation $<_{\mathbf{simp}}$ on **Reg** by, for all $\alpha, \beta \in \mathbf{Reg}$, $\alpha <_{\mathbf{simp}} \beta$ iff:

- $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ but $\mathbf{size}\,\alpha < \mathbf{size}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ and $\mathbf{size}\,\alpha = \mathbf{size}\,\beta$, but $\mathbf{numConcats}\,\alpha < \mathbf{numConcats}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$, $\mathbf{size}\,\alpha = \mathbf{size}\,\beta$ and $\mathbf{numConcats}\,\alpha = \mathbf{numConcats}\,\beta$, but $\mathbf{numSyms}\,\alpha < \mathbf{numSyms}\,\beta$; or

- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$, $\mathbf{size}\,\alpha = \mathbf{size}\,\beta$, $\mathbf{numConcats}\,\alpha = \mathbf{numConcats}\,\beta$ and $\mathbf{numSyms}\,\alpha = \mathbf{numSyms}\,\beta$, but α is standardized and β is not standardized.

We read $\alpha <_{\mathbf{simp}} \beta$ as α is *simpler* (less *complex*) than β. We define a relation $\equiv_{\mathbf{simp}}$ on **Reg** by, for all $\alpha, \beta \in \mathbf{Reg}$, $\alpha \equiv_{\mathbf{simp}} \beta$ iff α and β have the same closure complexity, size, numbers of concatenations, numbers of symbols, and status of being (or not being) standardized. We read $\alpha \equiv_{\mathbf{simp}} \beta$ as α and β have the *same complexity*. Finally, we define a relation $\leq_{\mathbf{simp}}$ on **Reg** by, for all $\alpha, \beta \in \mathbf{Reg}$, $\alpha \leq_{\mathbf{simp}} \beta$ iff $\alpha <_{\mathbf{simp}} \beta$ or $\alpha \equiv_{\mathbf{simp}} \beta$. We read $\alpha \leq_{\mathbf{simp}} \beta$ as α is at least as simpler as (no more complex) than β

For example, the following regular expressions are equivalent and have the same complexity:

$$1(01 + 10) + (\% + 01)1 \quad \text{and} \quad 011 + 1(\% + 01 + 10).$$

Proposition 3.3.12

(1) $<_{\mathbf{simp}}$ *is transitive.*

(2) $\equiv_{\mathbf{simp}}$ *is reflexive on* **Reg**, *transitive and symmetric.*

(3) For all $\alpha, \beta \in \mathbf{Reg}$, *exactly one of the following holds:* $\alpha <_{\mathbf{simp}} \beta$, $\beta <_{\mathbf{simp}} \alpha$ *or* $\alpha \equiv_{\mathbf{simp}} \beta$.

(4) $\leq_{\mathbf{simp}}$ *is transitive, and, for all* $\alpha, \beta \in \mathbf{Reg}$, $\alpha \equiv_{\mathbf{simp}} \beta$ *iff* $\alpha \leq \beta$ *and* $\beta \leq \alpha$.

The Forlan module `Reg` defines the abstract type `cc` of closure complexities, along with these functions:

```
val ccToList  : cc -> int list
val singCC    : int -> cc
val unionCC   : cc * cc -> cc
val succCC    : cc -> cc
val cc        : reg -> cc
val compareCC : cc * cc -> order
```

The function `ccToList` is the identity function on closure complexities: all that changes is the type. `singCC n` returns the singleton closure complexity $[n]$, if n is nonnegative; otherwise it issues an error message. The functions `unionCC` and `succCC` implement the union and successor operations on closure complexities. The function `cc` corresponds to **cc**, and `compareCC` implements $<_{cc}$.

Here are some examples of how these functions can be used:

```
- val ns = Reg.succCC(Reg.unionCC(Reg.singCC 1, Reg.singCC 1));
val ns = - : Reg.cc
- Reg.ccToList ns;
val it = [2,2] : int list
```

```
- val ms = Reg.unionCC(ns, Reg.succCC ns);
val ms = - : Reg.cc
- Reg.ccToList ms;
val it = [3,3,2,2] : int list
- Reg.ccToList(Reg.cc(Reg.fromString "(00*11*)*"));
val it = [2,2,1,1] : int list
- Reg.ccToList(Reg.cc(Reg.fromString "% + 0(0 + 11*0)*11*"));
val it = [2,1,1,1,1,0,0,0] : int list
- Reg.compareCC
= (Reg.cc(Reg.fromString "(00*11*)*"),
=  Reg.cc(Reg.fromString "% + 0(0 + 11*0)*11*"));
val it = GREATER : order
- Reg.compareCC
= (Reg.cc(Reg.fromString "(00*11*)*"),
=  Reg.cc(Reg.fromString "(1*10*0)*"));
val it = EQUAL : order
```

The module `Reg` also includes these functions:

```
val numConcats             : reg -> int
val numSyms                : reg -> int
val standardized           : reg -> bool
val compareComplexity      : reg * reg -> order
val compareComplexityTotal : reg * reg -> order
```

The first two functions implement the functions with the same names. The function `standardized` tests whether a regular expression is standardized, and the function `compareComplexity` implements $\leq_{\mathbf{simp}}/\equiv_{\mathbf{simp}}$. Finally, `compareComplexityTotal` is like `compareComplexity`, but falls back on `Reg.compare` (our total ordering on regular expressions) to order regular expressions with the same complexity. Thus `compareComplexityTotal` is a total ordering.

Here are some examples of how these functions can be used:

```
- Reg.numConcats(Reg.fromString "(01)*(10)*");
val it = 3 : int
- Reg.numSyms(Reg.fromString "(01)*(10)*");
val it = 4 : int
- Reg.standardized(Reg.fromString "00*1");
val it = true : bool
- Reg.standardized(Reg.fromString "00*0");
val it = false : bool
- Reg.compareComplexity
= (Reg.fromString "(00*11*)*",
=  Reg.fromString "% + 0(0 + 11*0)*11*");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "0**1**",  Reg.fromString "(01)**");
val it = GREATER : order
```

```
- Reg.compareComplexity
= (Reg.fromString "(0*1*)*",  Reg.fromString "(0*+1*)*");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "0+01",  Reg.fromString "0(%+1)");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "(01)2",  Reg.fromString "012");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "1(01+10)+(%+01)1",
=  Reg.fromString "011+1(%+01+10)");
val it = EQUAL : order
```

3.3.2 Weak Simplification

In this subsection, we give our first simplification algorithm: weak simplification. We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the \$ is redundant);
- $\beta + \beta$ or $\beta + (\beta + \gamma)$ (the duplicate occurrence of β is redundant);
- $\%\beta$ or $\beta\%$ (the % is redundant);
- $\$\beta$ or $\beta\$$ (both are equivalent to \$); and
- $\%^*$ or $\$^*$ or $(\beta^*)^*$ (the first two can be replaced by %, and the extra closure can be omitted in the third case).

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

Weakly simplified regular expressions have some pleasing properties.

Proposition 3.3.13

*(1) For all $\alpha \in$ **Reg**, if α is weakly simplified and $L(\alpha) = \emptyset$, then $\alpha = \$$.*

*(2) For all $\alpha \in$ **Reg**, if α is weakly simplified and $L(\alpha) = \{\%\}$, then $\alpha = \%$.*

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

E.g., part (2) of the proposition says that, if α is weakly simplified and $L(\alpha)$ is the language whose only string is %, then α is %.

Proof. The three parts are proved in order, using induction on regular expressions. We will show the concatenation case of part (3). Suppose $\alpha, \beta \in$ **Reg** and assume the inductive hypothesis: for all $a \in$ **Sym**, if α is weakly simplified

and $L(\alpha) = \{a\}$, then $\alpha = a$, and for all $a \in \mathbf{Sym}$, if β is weakly simplified and $L(\beta) = \{a\}$, then $\beta = a$. Suppose $a \in \mathbf{Sym}$, and assume that $\alpha\beta$ is weakly simplified and $L(\alpha\beta) = \{a\}$. We must show that $\alpha\beta = a$. Because $\alpha\beta$ is weakly simplified, so are α and β.

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 (2) 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.

□

Proposition 3.3.14
For all $\alpha \in \mathbf{Reg}$, if α is weakly simplified, then $\mathbf{alphabet}(L(\alpha)) = \mathbf{alphabet}\,\alpha$.

Proof. By Proposition 3.1.5, it suffices to show that, for all $\alpha \in \mathbf{Reg}$, if α is weakly simplified, then $\mathbf{alphabet}\,\alpha \subseteq \mathbf{alphabet}(L(\alpha))$. And this follows by an easy induction on α, using Proposition 3.3.13(2). □

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

Proposition 3.3.15
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 regular expressions. □

Finally, we have that weakly simplified regular expressions with closures generate infinite languages:

Proposition 3.3.16
For all $\alpha \in \mathbf{Reg}$, if α is weakly simplified and α has one or more closures, then $L(\alpha)$ is infinite.

Proof. An easy induction on regular expressions. □

Next, we see how we can test whether a regular expression is weakly simplified via a simple structural recursion. Define $\mathbf{weaklySimplified} \in \mathbf{Reg} \to \mathbf{Bool}$ by structural recursion, as follows. Given a regular expression α, it proceeds as follows:

- Suppose α is %, \$ or a symbol. Then it returns **true**.

- Suppose α has the form β^*. Then it checks that:
  - β is weakly simplified (this is done using recursion); and
  - β is neither %, nor \$, nor a closure.
- Suppose α has the form $\alpha_1\,\alpha_2$. Then it checks that:
  - α_1 and α_2 are weakly simplified; and
  - α_1 is neither % nor \$ nor a concatenation; and
  - α_2 is neither % nor \$; and
  - α has none of the following forms: $\beta^*\beta$, $\beta^*(\beta\gamma)$, $(\beta_1\beta_2)^*\beta_1$ or $(\beta_1\beta_2)^*\beta_1\gamma$.
- Suppose α has the form $\alpha_1 + \alpha_2$. Then it checks that:
  - α_1 and α_2 are weakly simplified; and
  - α_1 is neither \$ nor a union; and
  - α_2 is not \$;
  - if α_2 has the form $\beta_1 + \beta_2$, then $\alpha_1 < \beta_1$; and
  - if α_2 is not a union, then $\alpha_1 < \alpha_2$.

Proposition 3.3.17
*For all $\alpha \in$ **Reg**, α is weakly simplified iff* **weaklySimplified** $\alpha =$ **true**.

Proof. By induction on regular expressions. □

In preparation for giving our weak simplification algorithm, we need to define some auxiliary functions. We say that a regular expression α is *almost weakly simplified* iff either:

- $w \in \{\%, \$\}$; or
- all elements of **concatsToList** α are weakly simplified, and are not %, \$ or concatenations.

For example, $0^*0(1+2)^*(1+2) = 0^*(0((1+2)^*(1+2)))$ is almost weakly simplified, even though it's not weakly simplified. On the other hand: $(\$ + 1)1$ isn't almost weakly simplified, because $\$ + 1$ isn't weakly simplified; 1% isn't weakly simplified, because of the location of %; and $(01)1$ isn't almost weakly simplified, because of the location of the concatenation 01.

Let

$$\mathbf{WS} = \{\, \alpha \in \mathbf{Reg} \mid \alpha \text{ is weakly simplified} \,\}, \text{ and}$$
$$\mathbf{AWS} = \{\, \alpha \in \mathbf{Reg} \mid \alpha \text{ is almost weakly simplified} \,\}.$$

We define a function **shiftClosuresRight** $\in$ **AWS** $\to$ **WS** by recursion. Given $\alpha \in$ **AWS**, **shiftClosuresRight** proceeds as follows. If α is not a concatenation, than it returns α. Otherwise, $\alpha = \alpha_1\alpha_2$ for some $\alpha_1, \alpha_2 \in$ **Reg**. Since α is almost weakly simplified, so is α_2. So it lets $\alpha_2' \in$ **WS** be the result of calling **shiftClosuresRight** on α_2.

- If $\alpha_1\alpha_2'$ has the form $\beta^*\beta$, for some $\beta \in$ **Reg**, then **shiftClosuresRight** returns
$$\textbf{shiftClosuresRight}(\textbf{rightConcat}(\beta, \beta^*)).$$
- Otherwise, if $\alpha_1\alpha_2'$ has the form $\beta^*\beta\gamma$, for some $\beta, \gamma \in$ **Reg**, then **shiftClosuresRight** returns
$$\textbf{shiftClosuresRight}(\beta\beta^*\gamma)).$$
- Otherwise, if $\alpha_1\alpha_2'$ has the form $(\beta_1\beta_2)^*\beta_1$, for some $\beta_1, \beta_2 \in$ **Reg**, then **shiftClosuresRight** returns
$$\textbf{shiftClosuresRight}(\beta_1(\textbf{rightConcat}(\beta_2, \beta_1))^*).$$
- Otherwise, if $\alpha_1\alpha_2'$ has the form $(\beta_1\beta_2)^*\beta_1\gamma$, for some $\beta_1, \beta_2, \gamma \in$ **Reg**, then **shiftClosuresRight** returns
$$\textbf{shiftClosuresRight}(\beta_1(\textbf{rightConcat}(\beta_2, \beta_1))^*\gamma).$$
- Otherwise, **shiftClosuresRight** returns $\alpha_1\alpha_2'$.

(The work needed to justify the kind of well-founded recursion used in **shiftClosuresRight**'s definition will be added in a subsequent revision.)

Proposition 3.3.18
*For all $\alpha \in$ **AWS**, **shiftClosuresRight** α is equivalent to α and has the same closure complexity, size, number of concatenations and number of symbols as α.*

Define a function **deepClosure** $\in$ **WS** $\to$ **WS** as follows. For all $\alpha \in$ **WS**:

$$\begin{aligned}
\textbf{deepClosure}\,\% &= \%,\\
\textbf{deepClosure}\,\$ &= \%,\\
\textbf{deepClosure}\,(*(\alpha)) &= \alpha^*, \text{ and}\\
\textbf{deepClosure}\,\alpha &= \alpha^*, \text{ if } \alpha \notin \{\%, \$\} \text{ and } \alpha \text{ is not a closure.}
\end{aligned}$$

Lemma 3.3.19
*For all $\alpha \in$ **WS**, **deepClosure** α is equivalent to α^*, has the same alphabet as α^*, has a closure complexity that is no bigger than that of α^*, has a size that is no bigger than that of α^*, has the same number of concatenations as α^*, and has the same number of symbols α^*.*

Define a function **deepConcat** $\in$ **WS** $\times$ **WS** $\to$ **WS** as follows. For all $\alpha, \beta \in$ **WS**:

$$\begin{aligned}
\mathbf{deepConcat}(\alpha, \$) &= \$, \\
\mathbf{deepConcat}(\$, \alpha) &= \$, \text{ if } \alpha \neq \$, \\
\mathbf{deepConcat}(\alpha, \%) &= \alpha, \text{ if } \alpha \neq \$, \\
\mathbf{deepConcat}(\%, \alpha) &= \alpha, \text{ if } \alpha \notin \{\$, \%\}, \text{ and} \\
\mathbf{deepConcat}(\alpha, \beta) &= \mathbf{shiftClosuresRight}(\mathbf{rightConcat}(\alpha, \beta)), \\
&\qquad \text{if } \alpha, \beta \notin \{\$, \%\}.
\end{aligned}$$

To see that the last clause of this definition is proper, suppose that $\alpha, \beta \in$ **WS** $- \{\%, \$\}$. Thus all the elements of **concatsToList** α and **concatsToList** β are weakly simplified, and are not %, \$ or concatenations. Hence

$$\mathbf{concatsToList}(\mathbf{rightConcat}(\alpha, \beta)) = \mathbf{concatsToList}\,\alpha \mathbin{@} \mathbf{concatsToList}\,\beta$$

also has this property, showing that **rightConcat**(α, β) is almost weakly simplified, which is what **shiftClosuresRight** needs to deliver a weakly simplified result.

Lemma 3.3.20
*For all $\alpha, \beta \in$ **WS**, **deepConcat**(α, β) is equivalent to $\alpha\beta$, has an alphabet that is a subset of the alphabet of $\alpha\beta$, has a closure complexity that is no bigger than that of $\alpha\beta$, has a size that is no bigger than that of $\alpha\beta$, has no more concatenations than $\alpha\beta$, and has no more symbols than $\alpha\beta$.*

Define a function **deepUnion** $\in$ **WS** $\times$ **WS** $\to$ **WS** as follows. For all $\alpha, \beta \in$ **WS**:

$$\begin{aligned}
\mathbf{deepUnion}(\alpha, \$) &= \alpha, \\
\mathbf{deepUnion}(\$, \alpha) &= \alpha, \text{ if } \alpha \neq \$, \text{ and} \\
\mathbf{deepUnion}(\alpha, \beta) &= \mathbf{sortUnions}(\mathbf{rightUnion}(\alpha, \beta)), \text{ if } \alpha \neq \$ \text{ and } \beta \neq \$.
\end{aligned}$$

To see that the last clause of this definition is proper, suppose $\alpha, \beta \in$ **WS** $-$ $\{\$\}$. Then all the elements of **unionsToList**(**rightUnion**(α, β)) will be weakly simplified, and won't be \$ or unions. Consequently, **sortUnions** will deliver a weakly simplified result.

Lemma 3.3.21
*For all $\alpha, \beta \in$ **WS**, **deepUnion**(α, β) is equivalent to $\alpha + \beta$, has an alphabet that is a subset of the alphabet of $\alpha + \beta$, has a closure complexity that is no bigger than that of $\alpha + \beta$, has a size that is no bigger than that of $\alpha + \beta$, has no more concatenations than $\alpha + \beta$, and has no more symbols than $\alpha + \beta$.*

Now, we can define our weak simplification function/algorithm. Define **weaklySimplify** $\in$ **Reg** $\to$ **WS** by structural recursion:

- **weaklySimplify** $\% = \%$;
- **weaklySimplify** $\$ = \$$;
- **weaklySimplify** $a = a$, for all $a \in$ **Sym**;
- **weaklySimplify**$(*(\alpha))$

  $$= \textbf{deepClosure}(\textbf{weaklySimplify}\,\alpha),$$

  for all $\alpha \in$ **Reg**;
- **weaklySimplify**$(@(\alpha, \beta))$

  $$= \textbf{deepConcat}(\textbf{weaklySimplify}\,\alpha, \textbf{weaklySimplify}\,\beta),$$

  for all $\alpha, \beta \in$ **Reg**; and
- **weaklySimplify**$(+(\alpha, \beta))$

  $$= \textbf{deepUnion}(\textbf{weaklySimplify}\,\alpha, \textbf{weaklySimplify}\,\beta),$$

  for all $\alpha, \beta \in$ **Reg**.

Proposition 3.3.22
For all $\alpha \in$ **Reg***:*

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

(2) **alphabet**(**weaklySimplify**$(\alpha)) \subseteq$ **alphabet** α*;*

(3) **cc**(**weaklySimplify** $\alpha) \leq_{cc}$ **cc** α*;*

(4) **size**(**weaklySimplify** $\alpha) \leq$ **size** α*;*

(5) **numSyms**(**weaklySimplify** $\alpha) \leq$ **numSyms** α*; and*

(6) **numConcats**(**weaklySimplify** $\alpha) \leq$ **numConcats** α.

Proof. By induction on regular expressions. □

Exercise 3.3.23
Prove Proposition 3.3.22.

Proposition 3.3.24
For all $\alpha \in$ **Reg**, *if* α *is weakly simplified, then* **weaklySimplify**$(\alpha) = \alpha$.

Proof. By induction on regular expressions. □

Using our weak 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, and thus we can output a message saying that $L(\alpha)$ is infinite.

The Forlan module `Reg` defines the following functions relating to weak simplification:

```
val weaklySimplified : reg -> bool
val weaklySimplify   : reg -> reg
val toStrSet         : reg -> str set
```

The function `weaklySimplified` tests whether its argument is weakly simplified, and `weaklySimplify` implements **weaklySimplify**. Finally, the function `toStrSet` implements our algorithm for calculating the language generated by a regular expression, if that language is finite, and for announcing the this language is infinite, otherwise.

Here are some examples of how these functions can be used:

```
- val reg = Reg.input "";
@ (% + $0)(% + 00*0 + 0**)*
@ .
val reg = - : reg
- Reg.output("", Reg.weaklySimplify reg);
(% + 0* + 000*)*
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.weaklySimplify reg');
(% + 1)2(% + 3)(% + 4)
val it = () : unit
- Reg.output
= ("",
=  Reg.weaklySimplify(Reg.fromString "(00*11*)*"));
(00*11*)*
val it = () : unit
```

3.3.3 Local and Global Simplification

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

$$\mathbf{hasEmp} \in \mathbf{Reg} \to \mathbf{Bool}$$

by recursion:

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

Proposition 3.3.25
For all $\alpha \in \mathbf{Reg}$, $\% \in L(\alpha)$ *iff* $\mathbf{hasEmp}\,\alpha = \mathbf{true}$.

Proof. By induction on regular expressions. □

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

$$\mathbf{hasSym} \in \mathbf{Sym} \times \mathbf{Reg} \to \mathbf{Bool}$$

by recursion:

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

Proposition 3.3.26
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 regular expressions, using Proposition 3.3.25. □

Finally, we define a function/algorithm

$$\textbf{obviousSubset} \in \textbf{Reg} \times \textbf{Reg} \to \{\textbf{true}, \textbf{false}\}$$

meeting the following specification: for all $\alpha, \beta \in \textbf{Reg}$,

$$\text{if } \textbf{obviousSubset}(\alpha, \beta) = \textbf{true}, \text{ then } L(\alpha) \subseteq L(\beta).$$

I.e., this function is a *conservative approximation to subset testing*. 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 3.13, we will learn of a less efficient algorithm that will provide a complete test for $L(\alpha) \subseteq L(\beta)$.

Given $\alpha, \beta \in \textbf{Reg}$, $\textbf{obviousSubset}(\alpha, \beta)$ proceeds as follows. First, it lets $\alpha' = \textbf{weaklySimplify}\,\alpha$ and $\beta' = \textbf{weaklySimplify}\,\beta$. Then it returns $\textbf{obviSub}(\alpha', \beta')$, where

$$\textbf{obviSub} \in \textbf{WS} \times \textbf{WS} \to \textbf{Bool}$$

is the function defined below.

obviSub is defined by well-founded recursion on the sum of the sizes of its arguments. If $\alpha = \beta$, then it returns **true**; otherwise, it considers the possible forms of α.

- Suppose $\alpha = \%$. It returns $\textbf{hasEmp}\,\beta$.
- Suppose $\alpha = \$$. It returns **true**.
- Suppose $\alpha = a$, for some $a \in \textbf{Sym}$. It returns $\textbf{hasSym}(a, \beta)$.
- Suppose $\alpha = \alpha_1{}^*$, for some $\alpha_1 \in \textbf{Reg}$. Here it looks at the form of β.
  - Suppose $\beta = \%$. It returns **false**. (Because α will be weakly simplified, and so α won't generate $\{\%\}$.)
  - Suppose $\beta = \$$. It returns **false**.
  - Suppose $\beta = a$, for some $a \in \textbf{Sym}$. It returns **false**.
  - Suppose β is a closure. It returns $\textbf{obviSub}(\alpha_1, \beta)$.
  - Suppose $\beta = \beta_1\beta_2$, for some $\beta_1, \beta_2 \in \textbf{Reg}$. If $\textbf{hasEmp}\,\beta_1 = \textbf{true}$ and $\textbf{obviSub}(\alpha, \beta_2)$, then it returns **true**. Otherwise, if $\textbf{hasEmp}\,\beta_2 = \textbf{true}$ and $\textbf{obviSub}(\alpha, \beta_1)$, then it returns **true**. Otherwise, it returns **false** (even though the answer sometimes should be **true**).
  - Suppose $\beta = \beta_1 + \beta_2$, for some $\beta_1, \beta_2 \in \textbf{Reg}$. It returns

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

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

- Suppose $\alpha = \alpha_1\alpha_2$, for some $\alpha_1, \alpha_2 \in \mathbf{Reg}$. Here it looks at the form of β.
  - Suppose $\beta = \%$. It returns **false**. (Because α is weakly simplified, α won't generate $\{\%\}$.)
  - Suppose $\beta = \$$. It returns **false**. (Because α is weakly simplified, α won't generate $\emptyset$.)
  - Suppose $\beta = a$, for some $a \in \mathbf{Sym}$. It returns **false**. (Because α is weakly simplified, α won't generate $\{a\}$.)
  - Suppose $\beta = {\beta_1}^*$, for some $\beta_1 \in \mathbf{Reg}$. It returns

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

    (even though this returns **false** too often).
  - Suppose $\beta = \beta_1\beta_2$, for some $\beta_1, \beta_2 \in \mathbf{Reg}$. If $\mathbf{obviSub}(\alpha_1, \beta_1) = \mathbf{true}$ and $\mathbf{obviSub}(\alpha_2, \beta_2) = \mathbf{true}$, then it returns **true**. Otherwise, if $\mathbf{hasEmp}\,\beta_1 = \mathbf{true}$ and $\mathbf{obviSub}(\alpha, \beta_2) = \mathbf{true}$, then it returns **true**. Otherwise, if $\mathbf{hasEmp}\,\beta_2 = \mathbf{true}$ and $\mathbf{obviSub}(\alpha, \beta_1) = \mathbf{true}$, then it returns **true**. Otherwise, if β_1 is a closure but β_2 is not a closure, then it returns

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

    (even though this returns **false** too often). Otherwise, if β_2 is a closure but β_1 is not a closure, then it returns

    $$\mathbf{obviSub}(\alpha_1, \beta)\ \mathbf{and}\ \mathbf{obviSub}(\alpha_2, \beta_2)$$

    (even though this returns **false** too often). Otherwise, if β_1 and β_2 are closures, then it returns

    $$(\mathbf{obviSub}(\alpha_1, \beta_1)\ \mathbf{and}\ \mathbf{obviSub}(\alpha_2, \beta))$$
    $$\mathbf{or}$$
    $$(\mathbf{obviSub}(\alpha_1, \beta)\ \mathbf{and}\ \mathbf{obviSub}(\alpha_2, \beta_2))$$

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

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

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

- Suppose $\alpha = \alpha_1 + \alpha_2$. It returns

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

We say that α is *obviously a subset of* β iff $\textbf{obviousSubset}(\alpha, \beta) = \textbf{true}$. On the positive side, we have that, e.g., $\textbf{obviousSubset}(0^*011^*1, 0^*1^*) = \textbf{true}$. On the other hand, $\textbf{obviousSubset}((01)^*, (\% + 0)(10)^*(\% + 1)) = \textbf{false}$, even though $L((01)^*) \subseteq L((\% + 0)(10)^*(\% + 1))$.

Proposition 3.3.27
For all $\alpha, \beta \in \textbf{Reg}$, *if* $\textbf{obviousSubset}(\alpha, \beta) = \textbf{true}$, *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 $\alpha, \beta \in \textbf{Reg}$, if $\textbf{obviSub}(\alpha, \beta) = \textbf{true}$, then $L(\alpha) \subseteq L(\beta)$. The result then follows by Proposition 3.3.22. □

The Forlan module `Reg` provides the following functions corresponding to the auxiliary functions **hasEmp**, **hasSym** and **obviousSubset**:

```
val hasEmp        : reg -> bool
val hasSym        : sym * reg -> bool
val obviousSubset : reg * reg -> bool
```

Here are some examples of how they can be used:

```
- Reg.hasEmp(Reg.fromString "0*1*");
val it = true : bool
- Reg.hasEmp(Reg.fromString "01*");
val it = false : bool
- Reg.hasSym(Sym.fromString "0", Reg.fromString "0*1*");
val it = true : bool
- Reg.hasSym(Sym.fromString "1", Reg.fromString "0*1*");
val it = true : bool
- Reg.hasSym(Sym.fromString "0", Reg.fromString "0*$");
val it = false : bool
- Reg.obviousSubset
= (Reg.fromString "(0 + 1)*",
=  Reg.fromString "0*(0 + 1)*1*");
val it = true : bool
- Reg.obviousSubset
= (Reg.fromString "0*(0 + 1)*1*",
=  Reg.fromString "(0 + 1)*");
val it = true : bool
- Reg.obviousSubset
= (Reg.fromString "0*011*1",
=  Reg.fromString "0*1*");
val it = true : bool
- Reg.obviousSubset
```

```
= (Reg.fromString "(01 + 011)1*",
=  Reg.fromString "01*");
val it = true : bool
- Reg.obviousSubset
= (Reg.fromString "(01)*",
=  Reg.fromString "(% + 0)(10)*(% + 1)");
val it = false : bool
```

Our local and global simplification algorithms make use of simplification rules, which may be applied to arbitrary subtrees of regular expressions. There are three kinds of rules: structural rules, distributive rules and reduction rules.

There are nine *structural rules*, which preserve the alphabet, closure complexity, size, number of concatenations and number of symbols 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) $(\alpha\beta)^*\alpha \to \alpha(\beta\alpha)^*$.

For even small regular expressions, there may be a very large number of ways to reorganize them using the structural rules. E.g., suppose $\alpha_1, \ldots, \alpha_n$ are distinct regular expressions, for $n \geq 1$. There are $n!$ ways of ordering the α_i. And there are $(2n)!/(n!)(n+1)!$ (these are the Catalan numbers) binary trees with exactly n leaves. Consequently, using structural rules (1), (2) and (5) (withing making changes inside the α_i), we can reorganize $\alpha_1 + \cdots + \alpha_n$ into $(n!)(2n)!/(n!)(n+1)!$ regular expressions. For $n = 16$, this is about $7 * 10^{20}$.

There are two *distributive rules*, which preserve the alphabet of a regular expression:

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

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

Finally, there are 26 *reduction rules*, some of which make use of a conservative approximation *sub* to subset testing. When $\alpha \to \beta$ because of a reduction rule, we have that $\mathbf{alphabet}\,\beta \subseteq \mathbf{alphabet}\,\alpha$ and β **simp** α, where **simp** is the well-founded relation on **Reg** that is defined below.

We define the relation **simp** on **Reg** by: for all $\alpha, \beta \in \mathbf{Reg}$, α **simp** β iff either:

- $\mathbf{cc}\,\alpha <_{cc} \mathbf{cc}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$, but $\mathbf{size}\,\alpha < \mathbf{size}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ and $\mathbf{size}\,\alpha = \mathbf{size}\,\beta$, but $\mathbf{numConcats}\,\alpha < \mathbf{numConcats}\,\beta$; or
- $\mathbf{cc}\,\alpha = \mathbf{cc}\,\beta$ and $\mathbf{size}\,\alpha = \mathbf{size}\,\beta$, and $\mathbf{numConcats}\,\alpha = \mathbf{numConcats}\,\beta$, but $\mathbf{numSyms}\,\alpha < \mathbf{numSyms}\,\beta$.

Note that this is almost the same definition as that of $<_{\mathbf{simp}}$—the difference being that **simp** doesn't have the final step involving standardization.

Proposition 3.3.28
simp *is a well-founded relation on* **Reg**.

Proof. Follows by Propositions 3.3.5, 1.2.11 and 1.2.10, plus the fact that $<$ is well-founded on $\mathbb{N}$ (Proposition 1.2.5). □

The following proposition says that if we replace a subtree of a regular expression by a regular expression that is a **simp**-predecessor of that subtree, that the resulting, whole regular expression will be a **simp**-predecessor of the original, whole regular expression.

Proposition 3.3.29
Suppose $\alpha, \beta, \beta' \in \mathbf{Reg}$, β' **simp** β, *pat* $\in \mathbf{Path}$ *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* α' **simp** α.

Our reduction rules follow. In the rules, we abbreviate $\mathbf{hasEmp}\,\alpha = \mathbf{true}$ and $sub(\alpha, \beta) = \mathbf{true}$ to $\mathbf{hasEmp}\,\alpha$ and $sub(\alpha, \beta)$, respectively. Most of the rules strictly decrease a regular expression's closure complexity and size. The exceptions are labeled "cc" (for when the closure complexity strictly decreases, but the size strictly increases), "concatenations" (for when the closure complexity and size are preserved, but the number of concatenations strictly decreases) or "symbols" (for when the closure complexity and size normally strictly decrease, but occasionally they and the number of concatenations stay they same, but the number of symbols strictly decreases).

(1) If $sub(\alpha, \beta)$, then $\alpha + \beta \to \beta$.

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

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

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

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

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

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

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

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

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

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

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

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

(14) (cc) If $\mathbf{not}(\mathbf{hasEmp}\,\alpha)$ and $\mathbf{cc}\,\alpha \cup \overline{\mathbf{cc}\,\beta} <_{cc} \overline{\overline{\mathbf{cc}\,\beta}}$, then $(\alpha\beta^*)^* \to \% + \alpha(\alpha+\beta)^*$.

(15) (cc) If $\mathbf{not}(\mathbf{hasEmp}\,\beta)$ and $\overline{\mathbf{cc}\,\alpha} \cup \mathbf{cc}\,\beta <_{cc} \overline{\overline{\mathbf{cc}\,\alpha}}$, then $(\alpha^*\beta)^* \to \% + (\alpha+\beta)^*\beta$.

(16) (cc) If $\mathbf{not}(\mathbf{hasEmp}\,\alpha)$ or $\mathbf{not}(\mathbf{hasEmp}\,\gamma)$, and $\mathbf{cc}\,\alpha \cup \overline{\mathbf{cc}\,\beta} \cup \mathbf{cc}\,\gamma <_{cc} \overline{\overline{\mathbf{cc}, \beta}}$, then $(\alpha\beta^*\gamma)^* \to \% + \alpha(\beta + \gamma\alpha)^*\gamma$.

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

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

(19) (symbols) If $\alpha \notin \{\%, \$\}$ and $sub(\alpha^n, \beta)$, then $\alpha^{n+1}\alpha^* + \beta \to \alpha^n\alpha^* + \beta$.

(20) If $n \geq 2$, $l \geq 0$ and $2n - 1 < m_1 < \cdots < m_l$, then $(\alpha^n + \alpha^{n+1} + \cdots + \alpha^{2n-1} + \alpha^{m_1} + \cdots + \alpha^{m_l})^* \to \% + \alpha^n\alpha^*$.

(21) (symbols) If $\alpha \notin \{\%, \$\}$, then $\alpha + \alpha\beta \to \alpha(\% + \beta)$.

(22) (symbols) If $\alpha \notin \{\%, \$\}$, then $\alpha + \beta\alpha \to (\% + \beta)\alpha$.

(23) $\alpha^*(\% + \beta(\alpha + \beta)^*) \to (\alpha + \beta)^*$.

(24) $(\% + (\alpha + \beta)^*\alpha)\beta^* \to (\alpha + \beta)^*$.

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

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

In rules (14)–(16), the preconditions involving **cc** are necessary and sufficient conditions for the right-hand side to have strictly smaller closure complexity than the left-hand side.

Consider, e.g., reduction rule (4). 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 $\alpha\beta^* \approx \beta^*$, $\mathbf{alphabet}(\beta^*) \subseteq \mathbf{alphabet}(\alpha\beta^*)$ and $\beta^*\ \mathbf{simp}\ \alpha\beta^*$. The alphabet of β^* is clearly a subset of that 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 $w = \%w \in AB^*$.

And, to see that $\beta^* <_{cc} \alpha\beta^*$, it will suffice to show that $\mathbf{cc}(\beta^*) <_{cc} \mathbf{cc}(\alpha\beta^*)$. And we have that

$$\mathbf{cc}(\beta^*) = \overline{\mathbf{cc}\,\beta} <_{cc} \mathbf{cc}\,\alpha \cup \overline{\mathbf{cc}\,\beta} = \mathbf{cc}(\alpha\beta^*).$$

Because the structural rules preserve the size and alphabet of regular expressions, if we start with a regular expression α, there are only finitely many regular expressions that we can transform α into using structural rules (we can apply one of the rules to some subtree of α, giving us β_1, apply a rule to one of the subtrees of β_2, giving us β_2, etc.).

Suppose *sub* is a conservative approximation to subset testing. We say that a regular expression α is *locally simplified with respect to sub*: iff

- α is weakly simplified, and
- α can't be transformed by our structural rules into a regular expression to which one of our reduction rules applies.

The *local simplification of* a regular expression α *with respect to* a conservative approximation to subset testing *sub* proceeds as follows. It calls its main function with the weak simplification, β of α. The closure complexity, size, number of concatenations, and number of symbols of β are no bigger than those of α, and $\mathbf{alphabet}\,\beta \subseteq \mathbf{alphabet}\,\alpha$.

The main function is defined by well-founded recursion **simp**. It works as follows, when called with a weakly simplified argument, α.

- It generates the set X of all regular expressions $\mathbf{weaklySimplify}\,\gamma$, such that α can be reorganized using the structural rules into a regular expression β, which can be transformed by a single application of one of our reduction rules into γ.

- If X is empty, then it returns α.
- Otherwise, it calls itself recursively on the simplest element, γ of X (when X doesn't have a unique simplest element, the smallest of the simplest elements—in our total ordering on regular expressions—is selected). Because
  - the structural rules preserve closure complexity, size, number of concatenations, and number of symbols,
  - the reduction rules produce **simp**-predecessors, and
  - and weak simplification doesn't increase closure complexity, size, numbers of concatenations, or numbers of symbols,

  we have that γ **simp** α, so that this recursive call is legal. Furthermore, weak simplification, and all of the rules, either preserve or decrease (via $\subseteq$) the alphabet of regular expressions. Thus **alphabet** $\gamma \subseteq$ **alphabet** α.

The algorithm is referred to as "local", because at each recursive call of its main function, γ is chosen using the best local knowledge. This strategy is reasonably efficient, but there is no guarantee that another local choice wouldn't result in a simpler global answer.

We define a function/algorithm

$$\textbf{locallySimplify} \in (\textbf{Reg} \times \textbf{Reg} \to \textbf{Bool}) \to \textbf{Reg} \to \textbf{Reg}$$

by: for all conservative approximations to subset testing *sub*, and $\alpha \in$ **Reg**, **locallySimplify** *sub* α is the result of running our local simplification algorithm on α, using *sub* as the conservative approximation to subset testing.

Theorem 3.3.30
For all conservative approximations to subset testing sub, and $\alpha \in$ **Reg***:*

- **locallySimplify** *sub* α *is locally simplified with respect to sub;*
- **locallySimplify** *sub* α *is equivalent to* α*;*
- **alphabet**(**locallySimplify** *sub* α) $\subseteq$ **alphabet** α*; and*
- **locallySimplify** *sub* $\alpha \leq_{\textbf{simp}} \alpha$.

The Forlan module `Reg` provides the following functions relating to local simplification:

```
val locallySimplified    :
      (reg * reg -> bool) -> reg -> bool
val locallySimplify      :
      int option * (reg * reg -> bool) -> reg -> bool * reg
val locallySimplifyTrace :
      int option * (reg * reg -> bool) -> reg -> bool * reg
```

The function `locallySimplified` takes in a conservative approximation to subset testing *sub* and returns a function that tests whether a regular expression is *sub*-locally simplified. The function `locallySimplifyTrace` implements **locallySimplify**. It emits tracing messages explaining its operation, takes in an extra argument of type `int option`, and produces an extra result of type `bool`. If this extra argument is `NONE`, then it runs as does **locallySimplify**, and its boolean result is always **true**. But if it is `SOME` n, for $n \geq 1$, then at each recursive call of the algorithm's function, no more than n ways of reorganizing the function's argument will be considered, and the boolean part of the result will be **false** iff, in the final recursive call, n was not sufficient to explore all structural reorganizations, so that the regular expression returned may not be locally simplified with respect to *sub*. The function `locallySimplify` works identically, except it doesn't issue tracing messages.

Here are some examples of how these functions can be used.

```
- val locSimped = Reg.locallySimplified Reg.obviousSubset;
val locSimped = fn : reg -> bool
- locSimped(Reg.fromString "(1 + 00*1)*00*");
val it = false : bool
- locSimped(Reg.fromString "(0 + 1)*0");
val it = true : bool
- fun locSimp nOpt =
=       Reg.locallySimplify(nOpt, Reg.obviousSubset);
val locSimp = fn : int option -> reg -> bool * reg
- locSimp NONE (Reg.fromString "% + 0*0(0 + 1)* + 1*1(0 + 1)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*
val it = () : unit
- locSimp NONE (Reg.fromString "% + 1*0(0 + 1)* + 0*1(0 + 1)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*
val it = () : unit
- locSimp NONE (Reg.fromString "(1 + 00*1)*00*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*0
val it = () : unit
- Reg.locallySimplifyTrace
= (NONE, Reg.obviousSubset)
= (Reg.fromString "1*(01*01*)*");
considered all 10 structural reorganizations of 1*(01*01*)*
1*(01*01*)* transformed by structural rule 4 at position [2, 1] to
1*((01*)01*)* transformed by structural rule 4 at position [2, 1]
to 1*(((01*)0)1*)* transformed by reduction rule 14 at position
[2] to 1*(% + ((01*)0)((01*)0 + 1)*) weakly simplifies to
1*(% + 01*0(1 + 01*0)*)
```

```
considered all 40 structural reorganizations of
1*(% + 01*0(1 + 01*0)*)
1*(% + 01*0(1 + 01*0)*) transformed by structural rule 4 at
position [2, 2, 2] to 1*(% + 0(1*0)(1 + 01*0)*) transformed by
structural rule 4 at position [2, 2] to 1*(% + (01*0)(1 + 01*0)*)
transformed by reduction rule 23 at position [] to (1 + 01*0)*
considered all 4 structural reorganizations of (1 + 01*0)*
(1 + 01*0)* is locally simplified
val it = (true,-) : bool * reg
```

For even fairly small regular expressions, running through all the structural reorganizations can take prohibitively long. So, one often has to bound the number of such reorganizations, as in:

```
- 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.equal(Reg.weaklySimplify reg, reg);
val it = true : bool
- val (b', reg') = locSimp (SOME 10) reg;
val b' = false : bool
val reg' = - : reg
- Reg.output("", reg');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
- val (b'', reg'') = locSimp (SOME 1000) reg';
val b'' = true : bool
val reg'' = - : reg
- Reg.output("", reg'');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
```

Note that, in this transcript, `reg'` turns out to be locally simplified, despite the fact that `b'` is `false`.

Our global simplification algorithm comes in two variants, a non-distributive one, which doesn't use the distributive rules, and a distributive one, which does. Given a boolean b, a conservative approximation to subset testing *sub*, and a regular expression α, we say that α is *globally simplified with respect to* b and *sub* iff no strictly simpler regular expression can be found by an arbitrary number of applications of weak simplification, structural rules, reduction rules and—if $b = \mathbf{true}$—distributive rules.

The *global simplification of* a regular expression α *with respect to* a boolean b and conservative approximation to subset testing *sub* consists of generating the set X of all regular expressions β that can formed from α by an arbitrary

number of applications of weak simplification, the structural rules, reduction rules, and—in the case of the distributive variant—the distributive ones. The simplest element of X is then selected (when there isn't a unique simplest element, the smallest of the simplest elements—in our total ordering on regular expressions—is selected). (A proof that the generation of X terminates even in the distributive case is not yet complete.)

Of course, this algorithm is much less efficient than the local one, but by revisiting choices, it is capable of producing simpler answers.

We define a function/algorithm

$$\mathbf{globallySimplify} \in \mathbf{Bool} \times (\mathbf{Reg} \times \mathbf{Reg} \to \mathbf{Bool}) \to \mathbf{Reg} \to \mathbf{Reg}$$

by: for all $b \in \mathbf{Bool}$, conservative approximation to subset testing *sub*, and $\alpha \in \mathbf{Reg}$, $\mathbf{globallySimplify}\,(b, sub)\,\alpha$ is the result of running our global simplification algorithm on α, including the distributive rules iff $b = \mathbf{true}$, and using *sub* as our conservative approximation to subset testing.

Theorem 3.3.31
For all $b \in \mathbf{Bool}$, conservative approximations to subset testing sub, and $\alpha \in \mathbf{Reg}$:

- $\mathbf{globallySimplify}\,(b, sub)\,\alpha$ *is globally simplified with respect to b and sub;*
- $\mathbf{globallySimplify}\,(b, sub)\,\alpha$ *is equivalent to α;*
- $\mathbf{alphabet}(\mathbf{globallySimplify}\,(b, sub)\,\alpha) \subseteq \mathbf{alphabet}\,\alpha$*; and*
- $\mathbf{globallySimplify}\,(b, sub)\,\alpha \leq_{\mathbf{simp}} \alpha$.

The Forlan module `Reg` provides the following functions relating to global simplification:

```
val globallySimplified    :
      bool * (reg * reg -> bool) -> reg -> bool
val globallySimplifyTrace :
      int option * bool * (reg * reg -> bool) -> reg -> bool * reg
val globallySimplify      :
      int option * bool * (reg * reg -> bool) -> reg -> bool * reg
```

The function `globallySimplified` takes in a boolean b and a conservative approximation to subset testing *sub*, and returns a function that tests whether a regular expression is globally simplified with respect to b and *sub*. The function `globallySimplifyTrace` implements **globallySimplify**. It emits tracing messages explaining its operation, and takes in an extra argument of type `int option`, and produces an extra result of type `bool`. If this argument is `NONE`, then it runs as does **globallySimplify**, and the boolean result is always

true. But if it is `SOME` n, for $n \geq 1$, then at most n elements of the set X are generated, before picking the simplest one, and the boolean result is **false** if this n isn't enough to generate all of X. The function `globallySimplify` works identically, except it doesn't issue tracing messages.

For even quite small regular expressions, `globallySimplified` will fail to run to completion in an acceptable time-frame, and one will have to bound the size of the set X in order for `globallySimplify` and `globallySimplifyTrace` to run to completion in an acceptable time-frame.

Here are some examples of how these functions can be used.

```
- Reg.globallySimplifyTrace
= (NONE, false, Reg.obviousSubset)
= (Reg.fromString "(00*1)*");
considering candidates with explanations of length 0
simplest result now: (00*1)*
considering candidates with explanations of length 1
simplest result now: (00*1)* transformed by reduction rule 16 at
position [] to % + 0(0 + 10)*1
considering candidates with explanations of length 2
simplest result now: (00*1)* transformed by reduction rule 16 at
position [] to % + 0(0 + 10)*1 transformed by reduction rule 22 at
position [2, 2, 1, 1] to % + 0((% + 1)0)*1
considering candidates with explanations of length 3
considering candidates with explanations of length 4
considering candidates with explanations of length 5
considering candidates with explanations of length 6
considering candidates with explanations of length 7
search completed after considering 36 candidates with maximum size
12
(00*1)* transformed by reduction rule 16 at position [] to
% + 0(0 + 10)*1 transformed by reduction rule 22 at position
[2, 2, 1, 1] to % + 0((% + 1)0)*1 is globally simplified
val it = (true,-) : bool * reg
- locSimp NONE (Reg.fromString "(00*11*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 00*1(% + (0 + 1)*1)
val it = () : unit
- fun globSimp(nOpt, b) =
=       Reg.globallySimplify(nOpt, b, Reg.obviousSubset);
val globSimp = fn : int option * bool -> reg -> bool * reg
- globSimp (NONE, false) (Reg.fromString "(00*11*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0 + 1)*1
val it = () : unit
```

Finally, here are two examples showing how using the distributive rules can make a difference:

```
- globSimp (NONE, false) (Reg.fromString "% + 0*(0 + 1)");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0*(0 + 1)
val it = () : unit
- Reg.globallySimplifyTrace
= (NONE, true, Reg.obviousSubset)
= (Reg.fromString "% + 0*(0 + 1)");
considering candidates with explanations of length 0
simplest result now: % + 0*(0 + 1)
considering candidates with explanations of length 1
considering candidates with explanations of length 2
considering candidates with explanations of length 3
considering candidates with explanations of length 4
simplest result now: % + 0*(0 + 1) transformed by distributive
rule 1 at position [2] to % + 0*0 + 0*1 transformed by structural
rule 2 at position [] to (% + 0*0) + 0*1 transformed by reduction
rule 26 at position [1] to 0* + 0*1 transformed by reduction rule
21 at position [] to 0*(% + 1)
considering candidates with explanations of length 5
considering candidates with explanations of length 6
considering candidates with explanations of length 7
considering candidates with explanations of length 8
considering candidates with explanations of length 9
considering candidates with explanations of length 10
considering candidates with explanations of length 11
search completed after considering 76 candidates with maximum size
11
% + 0*(0 + 1) transformed by distributive rule 1 at position [2]
to % + 0*0 + 0*1 transformed by structural rule 2 at position []
to (% + 0*0) + 0*1 transformed by reduction rule 26 at position
[1] to 0* + 0*1 transformed by reduction rule 21 at position [] to
0*(% + 1) is globally simplified
val it = (true,-) : bool * reg
- globSimp (NONE, false) (Reg.fromString "(0(0(0 + 1))*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0(% + 0 + 1))*
val it = () : unit
- globSimp (NONE, true) (Reg.fromString "(0(0(0 + 1))*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0(% + 1))*
val it = () : unit
```

3.3.4 Notes

Although books on formal language theory usually study various regular expression equivalences, we have gone much further, giving three at least partly novel algorithms for regular expression simplification. Although many of the simplification and structural rules used in the simplification algorithms are well-known, some were invented, as was the concept of closure complexity.

3.4 Finite Automata and Labeled Paths

In this section, we: say what finite automata (FA) are, and show 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.

3.4.1 Finite Automata

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, and we often write (q, x, r) as

$$q \overset{x}{\to} r$$

  or $q, x \to r$).

We order transitions first by their left-hand sides, then by their middles, and then by their right-hand sides, using our total orderings on symbols and strings. This gives us a total ordering on transitions.

We often abbreviate Q_M, s_M, A_M and T_M to Q, s, A and T, when it's clear which FA we are working with. 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.

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

- $Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$;
- $s_M = \mathsf{A}$;

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

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 $p, x \to q$, then the machine *may* read x from its input and switch to state q. If $p, y \to r$ is also a transition, and the remaining input begins with y, then consuming y and switching to state r will also be possible, etc. The case when $x = \%$, i.e., when we have a %-*transition*, is interesting: a state switch can happen without reading anything.

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.

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
```

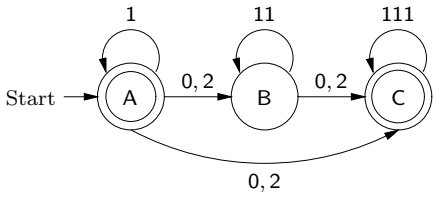
Remember that it's possible to read input from a file, and to write output to a file. During printing, Forlan merges transitions into transition families whenever possible.

Suppose that our example FA is in the file `3.4-fa`. We can input this FA into Forlan, and then output it to the standard output, as follows:

```
- val fa = FA.input "3.4-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 $p, x \to q$ 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.

Here is how our FA M can be described graphically:



The Java program JForlan, can be used to view and edit finite automata. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

We define a function **alphabet** $\in$ **FA** $\to$ **Alp** by: for all $M \in$ **FA**, **alphabet** M is $\{ a \in$ **Sym** $\mid$ there are q, x, r such that $q, x \to r \in T_M$ and $a \in$ **alphabet** $x \}$. I.e., **alphabet** M is all of the symbols appearing in the strings of M's transitions. We say that **alphabet** M is *the alphabet of* M. For example, the alphabet of our example FA M is $\{0, 1, 2\}$.

We say that an FA M is a *sub-FA* of an FA N iff:

- $Q_M \subseteq Q_N$;
- $s_M = s_N$;
- $A_M \subseteq A_N$; and
- $T_M \subseteq T_N$.

Thus $M = N$ iff M is a sub-FA of N and N is a sub-FA of M.

The Forlan module `FA` contains the functions

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

The function `equal` tests whether two FAs are equal, i.e., whether they have the same states, start states, accepting states and transitions. The functions `numStates` and `numTransitions` return the numbers of states and transitions, respectively, of an FA. The function `alphabet` returns the alphabet of an FA. And the function `sub` tests whether a first FA is a sub-FA of a second FA.

For example, we can continue out Forlan session as follows:

```
- val fa' = FA.input "";
@ {states} A, B, C
@ {start state} A
@ {accepting states} C
@ {transitions}
@ A, 0 -> B; A, 2 -> B; A, 0 -> C; A, 2 -> C;
@ B, 0 -> C; B, 2 -> C
@ .
val fa' = - : fa
- FA.equal(fa', fa);
val it = false : bool
- FA.sub(fa', fa);
val it = true : bool
- FA.sub(fa, fa');
val it = false : bool
- FA.numStates fa;
val it = 3 : int
- FA.numTransitions fa;
val it = 9 : int
- SymSet.output("", FA.alphabet fa);
0, 1, 2
val it = () : unit
```

3.4.2 Labeled Paths and FA Meaning

We will formally explain when strings are accepted by finite automata using the notion of a labeled path. A *labeled path* consists of a pair (xs, q), where $xs \in \mathbf{List}(\mathbf{Sym} \times \mathbf{Str})$ and $q \in \mathbf{Sym}$, and the set $\mathbf{LP}$ of labeled paths is $\mathbf{List}(\mathbf{Sym} \times \mathbf{Str}) \times \mathbf{Sym}$. Clearly, $\mathbf{LP}$ is countably infinite. We typically write $([(q_1, x_1), (q_2, x_2) \ldots, (q_n, x_n)], q_n + 1) \in \mathbf{LP}$ as:

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots q_n \overset{x_n}{\Rightarrow} q_{n+1}$$

or

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

This path describes a way of getting from state q_1 to state q_{n+1} in some unspecified machine, by reading the strings $x_1, \ldots, x_n$ from the machine's input. We start out in state q_1, make use of the transition $q_1, x_1 \to q_2$ to read x_1 from the input and switch to state q_2, etc.

Let $lp = (xs, q) \in \mathbf{LP}$. We say that:

- the *start state* of lp ($\mathbf{startState}\, lp$) is the left-hand side of the first element of xs, if xs is nonempty, and is q, if xs is empty;
- the *end state* of lp ($\mathbf{endState}\, lp$) is q;
- the *length* of lp ($|lp|$) is $|xs|$; and
- the *label* of lp ($\mathbf{label}\, lp$) is the result of concatenating the right-hand sides of xs (%, if xs is empty).

This defines functions $\mathbf{startState} \in \mathbf{LP} \to \mathbf{Sym}$, $\mathbf{endState} \in \mathbf{LP} \to \mathbf{Sym}$ and $\mathbf{label} \in \mathbf{LP} \to \mathbf{Str}$. For example $\mathsf{A} = ([\,], \mathsf{A})$ is a labeled path whose start and end states are both A, whose length is 0, and whose label is %. And

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

is a labeled path whose start state is A, end state is C, length is 3, and label is $(0)(11)(2) = 0112$.

We can join compatible paths together. Let $\mathbf{Join} = \{\, (lp_1, lp_2) \in \mathbf{LP} \times \mathbf{LP} \mid \mathbf{endState}\, lp_1 = \mathbf{startState}\, lp_2 \,\}$, and define $\mathbf{join} \in \mathbf{Join} \to \mathbf{LP}$ by: for all $xs, ys \in \mathbf{List}(\mathbf{Sym} \times \mathbf{Str})$ and $q, r \in \mathbf{Sym}$, if $((xs, q), (ys, r)) \in \mathbf{Join}$, then $\mathbf{join}((xs, q), (ys, r)) = (xs \mathbin{@} ys, r)$. E.g., if lp_1 and lp_2 are defined by

$$lp_1 = \mathsf{A} \overset{0}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B}, \quad \text{and} \quad lp_2 = \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \overset{2}{\Rightarrow} \mathsf{C},$$

then $\mathbf{join}(lp_1, lp_2)$ is

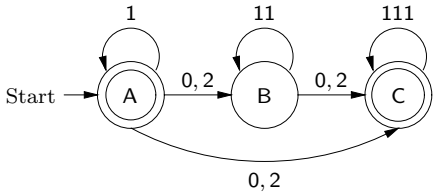
$$\mathsf{A} \overset{0}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \overset{11}{\Rightarrow} \mathsf{B} \overset{2}{\Rightarrow} \mathsf{C}.$$

A labeled path $(xs, q) \in \mathbf{LP}$ is *valid for* an FA M iff

- for all $i \in [1 : |xs| - 1]$, $\sharp 1(xs\, i), \sharp 2(xs\, i) \to \sharp 1(xs(i+1))$;
- if xs is nonempty, then $\sharp 1(xs\, |xs|), \sharp 2(xs\, |xs|) \to q$; and
- $q \in Q_M$.

(The last of these conditions is redundant whenever xs is nonempty.)

Recall our example FA M:



The labeled path $\mathsf{A} = ([\,], \mathsf{A})$ is valid for M, since $A \in Q_M$. And

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

is valid for M, since $\mathsf{A} \overset{0}{\to} \mathsf{B}$, $\mathsf{B} \overset{11}{\to} \mathsf{B}$ and $\mathsf{B} \overset{2}{\to} \mathsf{C}$ are in T_M (and $\mathsf{C} \in Q_M$). But the labeled path

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

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

A string w is *accepted by* a finite automaton M iff there is a labeled path lp such that

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

For example, 0112 is accepted by M because of the labeled path

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

since this labeled path is valid for M, is labeled by $\mathsf{0112} = (\mathsf{0})(\mathsf{11})(\mathsf{2})$, has a start state ($\mathsf{A}$) that is M's start state, and has an end state (C) that is one of M's accepting states.

Clearly, if w is accepted by M, then $\mathbf{alphabet}\, w \subseteq \mathbf{alphabet}\, M$. Thus $\{\, w \in \mathbf{Str} \mid w \text{ is accepted by } M \,\} \subseteq (\mathbf{alphabet}\, M)^*$, so we may define the *language accepted by* a finite automaton M ($L(M)$) to be

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

Furthermore:

Proposition 3.4.1
Suppose M is a finite automaton. Then $\mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}\, M$.

In other words, 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.

Going back to our example, 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. But 21112 and 2211 are not accepted by M.

Suppose that M is a sub-FA of N. Then any labeled path that is valid for M will also be valid for N. Furthermore, we have that:

Proposition 3.4.2
If M is a sub-FA of N, then $L(M) \subseteq L(N)$.

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
```

The function `equal` tests whether two labeled paths are equal. The functions `startState`, `endState`, `label` and `length` return the start state, end state, label and length, respectively, of a labeled path. And the function `join` joins two compatible paths, and issues an error message when given paths that are incompatible.

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.

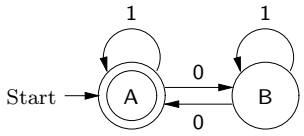
Here are some examples of labeled path and FA processing (`fa` is still our example FA):

```
- 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
```

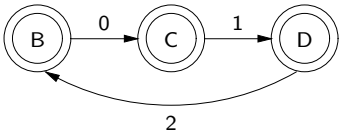
```
- Str.output("", LP.label lp');
%
val it = () : unit
- checkLP lp';
val it = () : unit
- val lp'' = LP.input "";
@ A, % => A, 1 => B
@ .
val lp'' = - : lp
- checkLP lp'';
invalid transition: "A, % -> A"

uncaught exception Error
- val lp''' = LP.input "";
@ B, 2 => C, 34 => D
@ .
val lp''' = - : lp
- LP.output("", LP.join(lp'', lp'''));
A, % => A, 1 => B, 2 => C, 34 => D
val it = () : unit
- LP.output("", LP.join(lp''', lp''));
incompatible labeled paths

uncaught exception Error
```

3.4.3 Design of Finite Automata

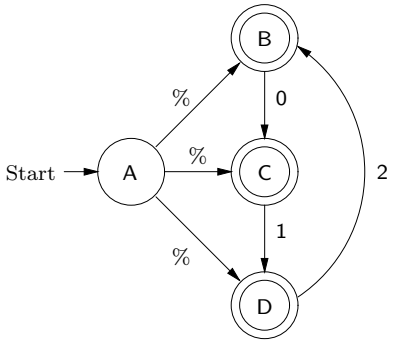
In this subsection, we give two examples of finite automata design. First, let's find a finite automaton that accepts the set of all strings of 0's and 1's with an even number of 0's. Thus, we should be looking for an FA whose alphabet is $\{0,1\}^*$. It seems reasonable that our machine have two states: an accepting 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. The above considerations lead us to the FA



For the second example, let's find an FA that accepts the language $X = \{\, w \in \{0,1,2\}^* \mid \text{for all substrings } x \text{ of } w, \text{ if } |x| = 2, \text{ then } x \in \{01, 12, 20\}\,\}$. We have that 0, 01, 012, 0120, etc., are in X, and so are 1, 12, 120, 1201, etc., and 2, 20, 201, 2012, etc. On the other hand, no string containing 00, 02, 11, 10, 22 or 20 is in X. The above observations suggest that part of our machine should look like:



But how should the machine get started? The simplest approach is to make use of %-transitions from the start state, giving us the FA



Exercise 3.4.3
Let $X = \{\, w \in \{0,1\}^* \mid 010 \text{ is not a substring of } w \,\}$. Find a finite automaton M such that $L(M) = X$.

Exercise 3.4.4
Let

$A = \{001, 011, 101, 111\}$, and
$B = \{\, w \in \{0,1\}^* \mid \text{for all } x, y \in \{0,1\}^*, \text{ if } w = x0y, \text{then there is a } z \in A$
such that z is a prefix of y $\}$.

Find a finite automaton M such that $L(M) = B$. Hint: see the second example of Subsection 3.2.2.

3.4.4 Notes

Finite automata are normally defined via transition functions, δ, which is simple to do for deterministric finite automata, but increasingly complicated as one adds degrees of nonterminininism. Furthermore, this approach means that a deterministic finite automaton (DFA) is not a nondeterministic finite automaton (NFA), and that an NFA is not an ϵ-NFA, because the transition function of a DFA is not one for an NFA, and the transition function for an NFA is not one of an ϵ-NFA. And formalizing our FAs, whose transition labels can be strings of length greater than one, is very messy if done via transition functions, which probably accounts for why such machines are not normally considered. Furthermore, in the standard approach, to say when strings are accepted by finite

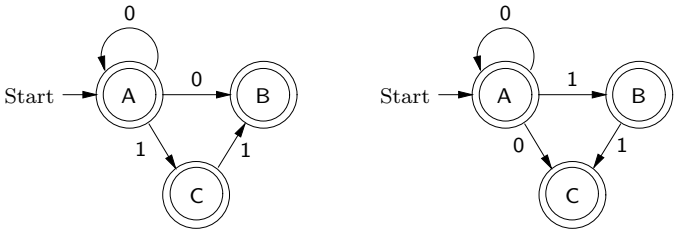
automata, one must first extend the different kinds of transition functions to work on strings.

In contrast, our approach is very simple. Instead of transition functions, we work with finite sets of transitions, enabling us to define the deterministic finite automata, nondeterministic finite automata, and nondeterministic finite automata with %-moves (which we call empty-string finite automata) as restrictions on finite automata. Furthermore, using labeled paths to say when—and how—strings are accepted by finite automata is simple, natural and diagrammatic. It's the analogue of using parse trees to say when—and how—strings are generated by grammars.

3.5 Isomorphism of Finite Automata

3.5.1 Definition and Algorithm

Let M and N be the finite automata



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:

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

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

Proposition 3.5.1
The relation **iso** *is reflexive on* **FA**, *symmetric and transitive.*

Proof. If M is an FA, then $\mathbf{id}_M$ is an isomorphism from M to M.

If M, N are FAs, and h is a isomorphism from M to N, then h^{-1} 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 $g \circ f$ an isomorphism from M_1 to M_3. □

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

Proposition 3.5.2
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

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

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

$$lp' = h\,q_1 \overset{x_1}{\Rightarrow} h\,q_2 \overset{x_2}{\Rightarrow} \cdots h\,q_n \overset{x_n}{\Rightarrow} h\,q_{n+1}.$$

Then the label of lp' is w, lp' is valid for N, $h\,q_1 = h\,s_M = s_N$ and $h\,q_{n+1} \in A_N$, showing that $w \in L(N)$. □

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 3.5.3
Suppose M and N are isomorphic FAs. Then $M \approx N$.

Proof. Since M **iso** N, we have that N **iso** M, by Proposition 3.5.1. Thus, by Proposition 3.5.2, we have that $L(M) \subseteq L(N) \subseteq L(M)$. Hence $L(M) = L(N)$, i.e., $M \approx N$. □

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

Proposition 3.5.4
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 3.5.5
Suppose that h is a bijection from Q_M to Q_N. Then

$$\{\, h\, q \overset{x}{\to} h\, r \mid q \overset{x}{\to} r \in T_M \,\} = T_N$$

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

$$\{\, p \in Q_M \mid q \overset{x}{\to} p \in T_M \,\}$$

to

$$\{\, p \in Q_N \mid r \overset{x}{\to} p \in T_N \,\}.$$

If any of the following conditions are true, then the algorithm reports 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 $s_N \notin A_N$; and
- $s_N \in A_N$, but $s_M \notin A_M$.

Otherwise, it calls its main function, **findIso**, which is defined by well-founded recursion.

The function **findIso** is called with an argument $(f, [C_1, \ldots, C_n])$, where:

- f is a bijection from a subset of Q_M to a subset of Q_N, and

- the C_i are *constraints* of the form (X, Y), where $X \subseteq Q_M$, $Y \subseteq Q_N$ and $|X| = |Y|$.

It returns an element of **Option** X, where X is the set of bijections from Q_M to Q_N. **none** is returned to indicate failure, and **some** h is returned when it has produced the bijection h. We say that a bijection *satisfies* a constraint (X, Y) iff it has a subset that is a bijection from X to Y.

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, $p \notin X$, $q \notin Y$ 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 well-founded *termination relation* in which argument $(f, [C_1, \ldots, C_n])$ is less than argument $(f', [C'_1, \ldots, C'_m])$ 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 $C'_1, \ldots, C'_m$.

When **findIso** is called with argument $(f, [C_1, \ldots, C_n])$, the following property, which we call (†), will hold: for all bijections h from a subset of Q_M to a subset of Q_N, if $h \supseteq f$ and h satisfies all of the C_i's, then:

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

Thus, if **findIso** is called with a bijection f and an empty list of constraints, it will follow, by Lemma 3.5.5, that f is an isomorphism from M to N.

Initially, the algorithm calls **findIso** with the *initial argument*:

$$(\emptyset, [(\{s_M\}, \{s_N\}), (U, V), (X, Y)]),$$

where $U = A_M - \{s_M\}$, $V = A_N - \{s_N\}$, $X = (Q_M - A_M) - \{s_M\}$ and $Y = (Q_N - A_N) - \{s_N\}$. Clearly the initial argument satisfies (†).

If **findIso** is called with argument $(f, [\,])$, then it returns **some** f.

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

Otherwise, if **findIso** is called with argument $(f, [(\{q\}, \{r\}), C_2, \ldots, C_n])$, then it proceeds as follows:

- If $(q, r) \in f$, then it calls itself recursively with argument $(f, [C_2, \ldots, C_n])$ 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 $q \in$ **domain** f or $r \in$ **range** f, then **findIso** returns **none**.
- 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 $A_x = \{\, p \in Q_M \mid q, x \to p \in T_M \,\}$ and $B_x = \{\, p \in Q_N \mid r, x \to p \in T_N \,\}$. If $|A_x| \neq |B_x|$, then it returns **none**. Otherwise, it adds the constraint (A_x, B_x) to our list of new constraints. When all such strings have been exhausted, it calls itself recursively with argument $(f \cup \{(q, r)\}, [C'_1, \ldots, C'_m, C_2, \ldots, C_n])$ and returns what this recursive call returns. (The size of the bijection has been increased by one.)

Otherwise, **findIso** has been called with argument $(f, [(A, A'), C_2, \ldots, C_n])$, where $|A| > 1$, and it proceeds as follows. It picks the smallest symbol $q \in A$, and lets $B = A - \{q\}$. Then, it works its way through the elements of A'. Given $r \in A'$, it lets $B' = A' - \{r\}$. Then, it tries calling itself recursively with argument $(f, [(\{q\}, \{r\}), (B, B'), C_2, \ldots, C_n])$. (The size of the bijection has been preserved, but the sum of the sizes of the weights of the constraints has gone down by $2 \cdot 3^{|B_1|} - 3 \geq 3$.) If this call returns a result of form **some** h, then it returns this to its caller. But if the call returns **none**, it tries the next element of A'. If it exhausts the elements of A', then it returns **none**.

Lemma 3.5.6
If **findIso** *is called with an argument* $(f, [C_1, \ldots, C_n])$ *satisfying property (†), then it returns* **none***, if there is no isomorphism from* M *to* N *that is a superset of* f *and satisfies the constaints* $C_1, \ldots, C_n$*, and returns* **some** h *where* h *is such an isomorphism, otherwise.*

Proof. By well-founded induction on our termination relation. I.e., when proving the result for $(f, [C_1, \ldots, C_n])$, we may assume that the result holds for all arguments $(f', [C'_1, \ldots, C'_m])$ that are strictly smaller in our termination ordering. □

Theorem 3.5.7
If **findIso** *is called with its initial argument, then it returns* **none**, *if there is no isomorphism from M to N, and returns* **some** h *where h is an isomorphism from M to N, otherwise.*

Proof. Follows easily from Lemma 3.5.6. □

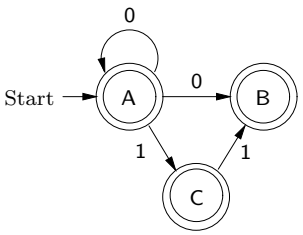
3.5.2 Isomorphism Finding/Checking in Forlan

The Forlan module `FA` also defines the functions

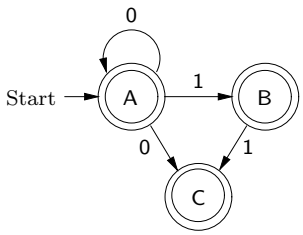
```
val isomorphism             : fa * fa * sym_rel -> bool
val findIsomorphism         : fa * fa -> sym_rel
val isomorphic              : fa * fa -> bool
val renameStates            : fa * sym_rel -> fa
val renameStatesCanonically : fa -> fa
```

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 there isn't 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 `fa1` and `fa2` have been bound to our example our example finite automata M and N:



(M)



(N)

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
```

3.5.3 Notes

Books on formal language theory rarely formalize the isomorphism of finite automata, although most note or prove that the minimization of deterministic finite automata yields a result that is unique up to the renaming of states. Our algorithm for trying to find an isomorphism between finite automata will be unsurprising to those familiar with graph algorithms.

3.6 Checking Acceptance and Finding Accepting Paths

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

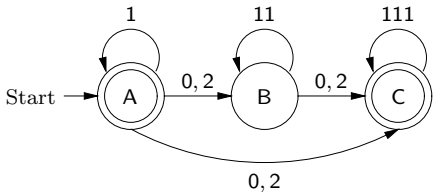
3.6.1 Processing a String from a Set of States

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; and
- 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 $p \in P$, $(p, w) \in X$; and

(2) for all $q, r \in Q_M$ and $x, y \in \mathbf{Str}$, if $(q, xy) \in X$ and $q, x \to r \in T_M$, 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 $\mathsf{A}, \mathsf{2} \to \mathsf{B}$;
- $(\mathsf{C}, \mathsf{111})$, because of $(\mathsf{A}, \mathsf{2111})$ and the transition $\mathsf{A}, \mathsf{2} \to \mathsf{C}$ (now, we're done with $(\mathsf{A}, \mathsf{2111})$);
- $(\mathsf{B}, \mathsf{1})$, because of $(\mathsf{B}, \mathsf{111})$ and the transition $\mathsf{B}, \mathsf{11} \to \mathsf{B}$ (now, we're done with $(\mathsf{B}, \mathsf{111})$);
- $(\mathsf{C}, \%)$, because of $(\mathsf{C}, \mathsf{111})$ and the transition $\mathsf{C}, \mathsf{111} \to \mathsf{C}$ (now, we're done with $(\mathsf{C}, \mathsf{111})$); and
- 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 3.6.1
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 $\mathbf{pre}\, y = x_1 x_2 \cdots x_{n-1}$, $q_1 \in P$ and $q_n = q$. Since $q_1 \in P$ and $w = (\mathbf{pre}\, y)y = x_1 x_2 \cdots x_{n-1} y$, we have that $(q_1, x_1 x_2 \cdots x_{n-1} y) = (q_1, w) \in X$, by (1). But $(q_1, x_1, q_2) \in T_M$, and thus $(q_2, x_2 \cdots x_{n-1} y) \in X$, by (2). Continuing on in this way (we could do this by mathematical induction), we finally get that $(q, y) = (q_n, y) \in X$. □

Lemma 3.6.2
For all $q \in Q_M$, $(q, \%) \in X$ iff $q \in \Delta_M(P, w)$.

Proof. Suppose $(q, \%) \in X$. Lemma 3.6.1 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 3.6.1 tells us that $(q, \%) \in X$. □

By Lemma 3.6.2, 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.

3.6.2 Checking String Acceptance and Finding Accepting Paths

Proposition 3.6.3
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 $w \in L(M)$. □

According to Proposition 3.6.3, 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 accepted            : fa -> str -> bool
```

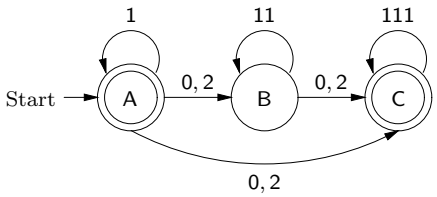
The function `processStr` takes in a finite automaton M, and returns a function that takes in a pair (P, w) and returns $\Delta_M(P, w)$. And, the function `accepted` takes in a finite automaton M, and returns a function that checks whether a string x is accepted by M.

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 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
- 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
```

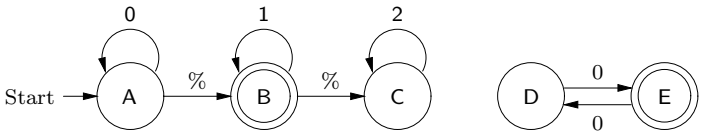
3.6.3 Notes

The material in this section is original. Our definition of the meaning of FAs via labeled paths allows us not simply to test *whether* an FA accepts a string w, but to ask for evidence—in the form of a labeled path—for *why* FA accepts w.

3.7 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 in* M 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 in* M 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 in* M iff q is not live in M; and
- *useful in* M iff q is both reachable and live in M.

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:

- $s_M \in X$; and
- 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.

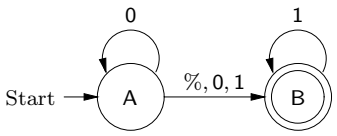
Similarly, there is a simple algorithm for generating the set of live states of a finite automaton M. We generate the least subset Y of Q_M such that:

- $A_M \subseteq Y$; and
- 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 are 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.

Thus, we can generate the set of useful states of an FA by generating the set of reachable states, generating the set of live states, and intersecting those sets of states.

Now, suppose N is the FA



Here, the transitions $(\mathsf{A}, \mathsf{0}, \mathsf{B})$ and $(\mathsf{A}, \mathsf{1}, \mathsf{B})$ are redundant, in the sense that if N' is the result of removing these transitions from N, we still have that $\mathsf{B} \in \Delta_{N'}(\{\mathsf{A}\}, \mathsf{0})$ and $\mathsf{B} \in \Delta_{N'}(\{\mathsf{A}\}, \mathsf{1})$.

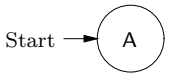
Given an FA M and a finite subset U of $\{\, (q, x, r) \mid q, r \in Q_M \text{ and } x \in \mathbf{Str} \,\}$, we write M/U for the FA that is identical to M except that its set of transitions is U. If M is an FA and $(p, x, q) \in T_M$, we say that:

- (p, x, q) *is redundant in* M iff $q \in \Delta_N(\{p\}, x)$, where $N = M/(T_M - \{(p, x, q)\})$; and
- (p, x, q) *is irredundant in* M iff (p, x, q) is not redundant in M.

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

- every state of M is useful, and every transition of M is irredundant; or
- $|Q_M| = 1$ and $A_M = T_M = \emptyset$.

Thus the FA



is simplified, even though its start state is not live, and is thus not useful.

Proposition 3.7.1
If M is a simplified finite automaton, then $\mathbf{alphabet}\, M = \mathbf{alphabet}(L(M))$.

Proof. We always have that $\mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}\, M$. But, because M is simplified, we also have that $\mathbf{alphabet}\, M \subseteq \mathbf{alphabet}(L(M))$, 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. This is because given any transition (p, x, q) of a simplified finite automaton, p is reachable and q is live. □

To give our simplification algorithm for finite automta, we need an auxiliary function for removing redundant transitions from an FA. Given an FA M, $p, q \in Q_M$ and $x \in \mathbf{Str}$, we say that (p, x, q) *is implicit in* M iff $q \in \Delta_M(\{p\}, x)$.

Given an FA M, we define a function $\mathbf{remRedun}_M \in \mathcal{P}\,T_M \times \mathcal{P}\,T_M \to \mathcal{P}\,T_M$ (we often drop the M when it's clear from the context) by well-founded recursion on the size of its second argument. For $U, V \subseteq T_M$, $\mathbf{remRedun}(U, V)$ proceeds as follows:

- If $V = \emptyset$, then it returns U.
- Otherwise, let v be the greatest element of V, and $V' = V - \{v\}$. If v is implicit in $M/(U \cup V')$, then **remRedun** returns the result of evaluating $\mathbf{remRedun}(U, V')$. Otherwise, it returns the result of evaluating $\mathbf{remRedun}(U \cup \{v\}, V')$.

In general, there are multiple—incompatible—ways of removing redundant transitions from an FA. **remRedun** is defined so as to favor removing transitions that are larger in our total ordering on transitions.

Proposition 3.7.2
Suppose M is a finite automaton. For all $U, V \subseteq T_M$, if all the elements of U are irredundant in $M/(U \cup V)$, and, for all $p, q \in Q_M$ and $x \in \mathbf{Str}$, (p, x, q) is implicit in M iff (p, x, q) is implicit in $M/(U \cup V)$, then all the elements of $\mathbf{remRedun}(U, V)$ are irredundant in $M/\mathbf{remRedun}(U, V)$, and, for all $p, q \in Q_M$ and $x \in \mathbf{Str}$, (p, x, q) is implicit in M iff (p, x, q) is implicit in $M/\mathbf{remRedun}(U, V)$.

Proof. By well-founded induction on the size of the second argument to **remRedun**. □

Now we can give an algorithm for simplifying finite automata. We define a function $\mathbf{simplify} \in \mathbf{FA} \to \mathbf{FA}$ by: $\mathbf{simplify}\,M$ is the finite automaton N produced by the following process.

- First, the useful states are M are determined.
- If s_M is not useful in M, the N is defined by:
  - $Q_N = \{s_M\}$;
  - $s_N = s_M$;
  - $A_N = \emptyset$; and
  - $T_N = \emptyset$.
- And, if s_M is useful in M, then N is defined by:
  - $Q_N = \{\, q \in Q_M \mid q \text{ is useful in } M \,\}$;

- $s_N = s_M$;
- $A_N = A_M \cap Q_N = \{\, q \in A_M \mid q \in Q_N \,\}$; and
- $T_N = \mathbf{remRedun}(\emptyset, \{\, (q, x, r) \in T_M \mid q, r \in Q_N \,\})$.

Proposition 3.7.3
Suppose M is a finite automaton. Then:

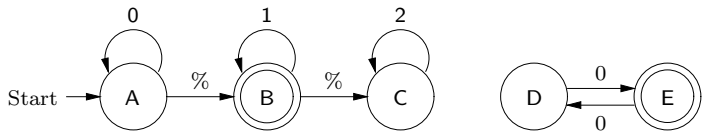
(1) **simplify** M *is simplified;*

(2) **simplify** $M \approx M$*; and*

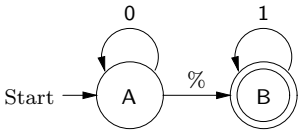
(3) $\mathbf{alphabet}(\mathbf{simplify}\, M) = \mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}\, M$.

Proof. Follows easily using Propositions 3.7.1 and 3.7.2. □

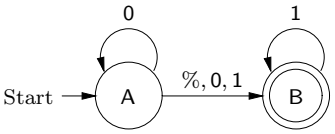
If M is the finite automaton



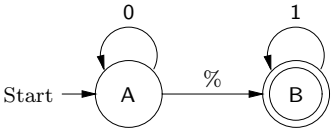
then **simplify** M is the finite automaton



And if N is the finite automaton



then **simplify** N is the finite automaton



Our simplification function/algorithm **simplify** gives us an algorithm for testing whether an FA is simplified: we apply **simplify** to it, and check that the resulting FA is equal to the original one.

Our simplification algorithm gives us an algorithm for testing whether the language accepted by an FA M is empty. We first simplify M, calling the result

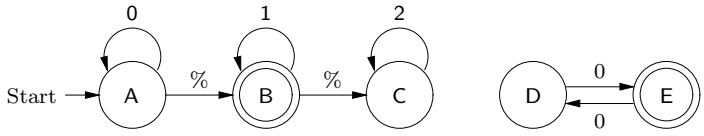
N. We then test whether $A_N = \emptyset$. If the answer is "yes", clearly $L(M) = L(N) = \emptyset$. And if the answer is "no", then s_N is useful, and so N (and thus M) accepts at least one string.

The Forlan module `FA` includes the following functions relating to the simplification of finite automata:

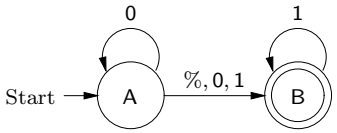
```
val simplify   : fa -> fa
val simplified : fa -> bool
```

The function `simplify` corresponds to **simplify**, and `simplified` tests whether an FA is simplified.

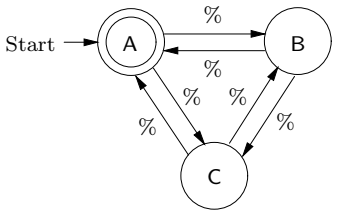
In the following, suppose `fa1` is the finite automaton



`fa2` is the finite automaton



and `fa3` is the finita automaton



Here are some example uses of `simplify` and `simplified`:

```
- FA.simplified fa1;
val it = false : bool
- val fa1' = FA.simplify fa1;
val fa1' = - : fa
- FA.output("", fa1');
{states} A, B {start state} A {accepting states} B
{transitions} A, % -> B; A, 0 -> A; B, 1 -> B
val it = () : unit
- FA.simplified fa1';
val it = true : bool
- val fa2' = FA.simplify fa2;
val fa2' = - : fa
- FA.output("", fa2');
```

```
{states} A, B {start state} A {accepting states} B
{transitions} A, % -> B; A, 0 -> A; B, 1 -> B
val it = () : unit
- val fa3' = FA.simplify fa3;
val fa3' = - : fa
- FA.output("", fa3');
{states} A, B, C {start state} A {accepting states} A
{transitions} A, % -> B | C; B, % -> A; C, % -> A
val it = () : unit
```

Thus the simplification of `fa3` resulted in the removal of the %-transitions between B and C.

Exercise 3.7.4
In the simplification of `fa3`, if transitions had been considered for removal due to being redundant in other orders, what FAs could have resulted.

3.7.1 Notes

The removal of useless states is analogous to the standard approach to ridding grammars of useless variables. The idea of removing redundant transitions, though, seems to be novel.

3.8 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 by defining an indexed family of languages, Λ.

3.8.1 Definition of Λ

Proposition 3.8.1
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 $w \in \mathbf{Str}$, if $q, w \to r \in T_M$, then $r \in \Delta_M(\{q\}, w)$.

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

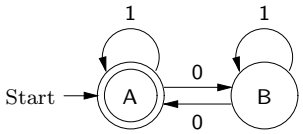
Suppose M is a finite automaton and $q \in Q_M$. Then we define

$$\Lambda_{M,q} = \{\, w \in \mathbf{Str} \mid q \in \Delta_M(\{s_M\}, w) \,\}.$$

In other words, $\Lambda_{M,q}$ is the labels of all of the valid labeled paths for M that start at state s_M and end at q, i.e., it's all strings that can take us to state

q when processed by M. Clearly, $\Lambda_{M,q} \subseteq (\textbf{alphabet}\, M)^*$, for all FAs M and $q \in Q_M$. If it's clear which FA we are talking about, we sometimes abbreviate $\Lambda_{M,q}$ to Λ_q.

Let our example FA, M, be



Then:

- $01101 \in \Lambda_{\mathsf{A}}$, because of the labeled path

$$\mathsf{A} \overset{0}{\Rightarrow} \mathsf{B} \overset{1}{\Rightarrow} \mathsf{B} \overset{1}{\Rightarrow} \mathsf{B} \overset{0}{\Rightarrow} \mathsf{A} \overset{1}{\Rightarrow} \mathsf{A},$$

- $01100 \in \Lambda_{\mathsf{B}}$, because of the labeled path

$$\mathsf{A} \overset{0}{\Rightarrow} \mathsf{B} \overset{1}{\Rightarrow} \mathsf{B} \overset{1}{\Rightarrow} \mathsf{B} \overset{0}{\Rightarrow} \mathsf{A} \overset{0}{\Rightarrow} \mathsf{B}.$$

Proposition 3.8.2
Suppose M is an FA. Then $L(M) = \bigcup\{\, \Lambda_{M,q} \mid q \in A_M \,\}$, i.e., for all w, $w \in L(M)$ iff $w \in \Lambda_{M,q}$ for some $q \in A_M$.

Proof.

(only if) Suppose $w \in L(M)$. By Proposition 3.5.3, we have that $\Delta_M(s_M, w) \cap A_M \neq \emptyset$, so that there is a $q \in A_M$ such that $q \in \Delta_M(s_M, w)$. Thus $w \in \Lambda_{M,q}$.

(if) Suppose $w \in \Lambda_{M,q}$ for some $q \in A_M$. Thus $q \in \Delta_M(s_M, w)$ and $q \in A_M$, so that $\Delta_M(s_M, w) \cap A_M \neq \emptyset$. Hence $w \in L(M)$, by Proposition 3.5.3.

□

Proposition 3.8.3
Suppose M is a finite automaton.

(1) $\% \in \Lambda_{M,s_M}$.

(2) For all $q, r \in Q_M$ and $w, x \in$ **Str**. *If $w \in \Lambda_{M,q}$ and $q, x \to r \in T_M$, then $wx \in \Lambda_{M,r}$.*

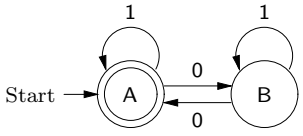
Proof.

(1) By Proposition 3.8.1(1), we have that $s_M \in \Delta(\{s_M\}, \%)$, so that $\% \in \Lambda_{M,s_M}$.

(2) Suppose $q, r \in Q_M$, $w, x \in \mathbf{Str}$, $w \in \Lambda_{M,q}$ and $q, x \to r \in T_M$. Thus $q \in \Delta(\{s_M\}, w)$. Because $q, x \to r \in T_M$, Proposition 3.8.1(2) tells us that $r \in \Delta(\{q\}, x)$. Hence by Proposition 3.8.1(3), we have that $r \in \Delta(\{s_M\}, wx)$, so that $wx \in \Lambda_{M,r}$.

□

Our main example will be the FA, M:



Let

$$X = \{\, w \in \{0,1\}^* \mid w \text{ has an even number of } 0\text{'s} \,\}, \text{ and}$$
$$Y = \{\, w \in \{0,1\}^* \mid w \text{ has an odd number of } 0\text{'s} \,\}.$$

We want to prove that $L(M) = X$. Because $A_M = \{\mathsf{A}\}$, Proposition 3.8.2 tells us that $L(M) = \Lambda_{M,\mathsf{A}}$. Thus it will suffice to show that $\Lambda_{M,\mathsf{A}} = X$. But our approach will also involve showing $\Lambda_{M,\mathsf{B}} = Y$. We would cope with more states analogously, having one language per state.

3.8.2 Proving that Enough is Accepted

First, we study techniques for showing that everything we want an automaton to accept is really accepted.

Since $X, Y \subseteq \{0,1\}^*$, to prove that $X \subseteq \Lambda_{M,\mathsf{A}}$ and $Y \subseteq \Lambda_{M,\mathsf{B}}$, it will suffice to use strong string induction to show that, for all $w \in \{0,1\}^*$:

(A) if $w \in X$, then $w \in \Lambda_{M,\mathsf{A}}$; and

(B) if $w \in Y$, then $w \in \Lambda_{M,\mathsf{B}}$.

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

(A) if $x \in X$, then $x \in \Lambda_{\mathsf{A}}$; and

(B) if $x \in Y$, then $x \in \Lambda_{\mathsf{B}}$.

We must prove that:

(A) if $w \in X$, then $w \in \Lambda_{\mathsf{A}}$; and

(B) if $w \in Y$, then $w \in \Lambda_{\mathsf{B}}$.

There are two parts to show.

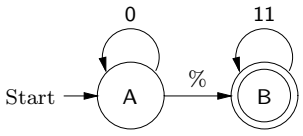
(A) Suppose $w \in X$, so that w has an even number of $\mathsf{0}$'s. We must show that $w \in \Lambda_{\mathsf{A}}$. There are three cases to consider.

- Suppose $w = \%$. By Proposition 3.8.3(1), we have that $w = \% \in \Lambda_{\mathsf{A}}$.
- Suppose $w = x\mathsf{0}$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Thus x has an odd number of $\mathsf{0}$'s, so that $x \in Y$. Because x is a proper substring of w, part (B) of the inductive hypothesis tells us that $x \in \Lambda_{\mathsf{B}}$. Furthermore, $\mathsf{B},\mathsf{0} \to \mathsf{A} \in T$, so that $w = x\mathsf{0} \in \Lambda_{\mathsf{A}}$, by Proposition 3.8.3(2).
- Suppose $w = x\mathsf{1}$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Thus x has an even number of $\mathsf{0}$'s, so that $x \in X$. Because x is a proper substring of w, part (A) of the inductive hypothesis tells us that $x \in \Lambda_{\mathsf{A}}$. Furthermore, $\mathsf{A},\mathsf{1} \to \mathsf{A} \in T$, so that $w = x\mathsf{1} \in \Lambda_{\mathsf{A}}$, by Proposition 3.8.3(2).

(B) Suppose $w \in Y$, so that w has an odd number of $\mathsf{0}$'s. We must show that $w \in \Lambda_{\mathsf{B}}$. There are three cases to consider.

- Suppose $w = \%$. But the number of $\mathsf{0}$'s in $\%$ is 0, which is even—contradiction. Thus $w \in \Lambda_{\mathsf{B}}$.
- Suppose $w = x\mathsf{0}$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Thus x has an even number of $\mathsf{0}$'s, so that $x \in X$. Because x is a proper substring of w, part (A) of the inductive hypothesis tells us that $x \in \Lambda_{\mathsf{A}}$. Furthermore, $\mathsf{A},\mathsf{0} \to \mathsf{B} \in T$, so that $w = x\mathsf{0} \in \Lambda_{\mathsf{B}}$, by Proposition 3.8.3(2).
- Suppose $w = x\mathsf{1}$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Thus x has an odd number of $\mathsf{0}$'s, so that $x \in Y$. Because x is a proper substring of w, part (B) of the inductive hypothesis tells us that $x \in \Lambda_{\mathsf{B}}$. Furthermore, $\mathsf{B},\mathsf{1} \to \mathsf{B} \in T$, so that $w = x\mathsf{1} \in \Lambda_{\mathsf{B}}$, by Proposition 3.8.3(2).

Let N be the finite automaton



Here we hope that $\Lambda_{N,\mathsf{A}} = \{\mathsf{0}\}^*$ and $L(N) = \Lambda_{N,\mathsf{B}} = \{\mathsf{0}\}^*\{\mathsf{11}\}^*$, but if we try to prove that

$$\{\mathsf{0}\}^* \subseteq \Lambda_{N,\mathsf{A}}, \text{ and}$$
$$\{\mathsf{0}\}^*\{\mathsf{11}\}^* \subseteq \Lambda_{N,\mathsf{B}}$$

using our standard technique, there is a complication related to the %-transition.

We use strong string induction to show that, for all $w \in \{\mathsf{0},\mathsf{1}\}^*$:

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

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

In part (B), we assume that $w \in \{0\}^*\{11\}^*$, so that $w = 0^n(11)^m$ for some $n, m \in \mathbb{N}$. We must show that $w \in \Lambda_B$. We consider two cases: $m = 0$ and $m \geq 1$. The second of these is straightforward, so let's focus on the first. Then $w = 0^n \in \{0\}^*$. We want to use part (A) of the inductive hypothesis to conclude that $0^n \in \Lambda_A$, but there is a problem: 0^n is not a proper substring of $0^n = w$.

So, we must consider two subcases, when $n = 0$ and $n \geq 1$. In the first subcase, because $\% \in \Lambda_A$ and $A, \% \to B \in T$, we have that $w = \% = \%\% \in \Lambda_B$.

In the second subcase, we have that $w = 0^{n-1}0$. By part (A) of the inductive hypothesis, we have that $0^{n-1} \in \Lambda_A$. Thus, because $A, 0 \to A \in T$ and $A, \% \to B \in T$, we can conclude $w = 0^n = 0^{n-1}0\% \in \Lambda_B$.

Because there are no transitions from B back to A, we could first prove that, for all $w \in \{0, 1\}^*$,

(A) if $w \in \{0\}^*$, then $w \in \Lambda_A$,

and then use (A) to prove that for all $w \in \{0, 1\}^*$,

(B) if $w \in \{0\}^*\{11\}^*$, then $w \in \Lambda_B$.

This works whenever one part of a machine has transitions to another part, but there are no transitions from that second part back to the first part, i.e., when the two parts are not mutually recursive.

In the case of N, we could use mathematical induction instead of strong string induction:

(A) for all $n \in \mathbb{N}$, $0^n \in \Lambda_A$, and

(B) for all $n, m \in \mathbb{N}$, $0^n(11)^m \in \Lambda_B$ (do induction on m, fixing n).

3.8.3 Proving that Everything Accepted is Wanted

It's tempting to try to prove that everything accepted by a finite automaton is wanted using strong string induction, with implications like

(A) if $w \in \Lambda_A$, then $w \in X$.

Unfortunately, this doesn't work when a finite automaton contains %-transitions. Instead, we do such proofs using a new induction principle that we call induction on Λ.

Theorem 3.8.4 (Principle of Induction on Λ)
Suppose M is a finite automaton, and $P_q(w)$ is a property of a $w \in \Lambda_{M,q}$, for all $q \in Q_M$. If

- *$P_{s_M}(\%)$ and*
- *for all $q, r \in Q_M$, $x \in$ **Str** and $w \in \Lambda_{M,q}$,*
  if $q, x \to r \in T_M$ and (†) $P_q(w)$, then $P_r(wx)$,

then

$$\textit{for all } q \in Q_M,\ \textit{for all } w \in \Lambda_{M,q}, P_q(w).$$

We refer to (†) as the inductive hypothesis.

Proof. It suffices to show that, for all $lp \in \mathbf{LP}$, for all $q \in Q_M$, if lp is valid for M, $\mathbf{startState}\, lp = s_M$ and $\mathbf{endState}\, lp = q$, then $P_q(\mathbf{label}\, lp)$. We prove this by well-founded induction on the length of lp. □

In the case of our example FA, M, we can let $P_{\mathsf{A}}(w)$ and $P_{\mathsf{B}}(w)$ be $w \in X$ and $w \in Y$, respectively, where, as before,

$$X = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ has an even number of } \mathsf{0}\text{'s} \,\}, \text{ and}$$
$$Y = \{\, w \in \{\mathsf{0},\mathsf{1}\}^* \mid w \text{ has an odd number of } \mathsf{0}\text{'s} \,\}.$$

Then the principle of induction on Λ tells us that

(A) for all $w \in \Lambda_{\mathsf{A}}$, $w \in X$, and

(B) for all $w \in \Lambda_{\mathsf{B}}$, $w \in Y$,

follows from showing

(empty string) $\% \in X$;

($\mathsf{A}, \mathsf{0} \to \mathsf{B}$) for all $w \in \Lambda_{\mathsf{A}}$, if (†) $w \in X$, then $w\mathsf{0} \in Y$;

($\mathsf{A}, \mathsf{1} \to \mathsf{A}$) for all $w \in \Lambda_{\mathsf{A}}$, if (†) $w \in X$, then $w\mathsf{1} \in X$;

($\mathsf{B}, \mathsf{0} \to \mathsf{A}$) for all $w \in \Lambda_{\mathsf{B}}$, if (†) $w \in Y$, then $w\mathsf{0} \in X$; and

($\mathsf{B}, \mathsf{1} \to \mathsf{B}$) for all $w \in \Lambda_{\mathsf{B}}$, if (†) $w \in Y$, then $w\mathsf{1} \in Y$.

We refer to (†) as the inductive hypothesis.

In fact, when setting this proof up, instead of explicitly mentioning P_{A} and P_{B}, we can simply say that we are proving

(A) for all $w \in \Lambda_{\mathsf{A}}$, $w \in X$, and

(B) for all $w \in \Lambda_{\mathsf{B}}$, $w \in Y$,

by induction on Λ.

There are five steps to show.

(empty string) Because $\% \in \{\mathsf{0},\mathsf{1}\}^*$ and $\%$ has no $\mathsf{0}$'s, we have that $\% \in X$.

($\mathsf{A}, \mathsf{0} \to \mathsf{B}$) Suppose $w \in \Lambda_{\mathsf{A}}$, and assume the inductive hypothesis: $w \in X$. Hence $w \in \{\mathsf{0},\mathsf{1}\}^*$ and w has an even number of $\mathsf{0}$'s. Thus $w\mathsf{0} \in \{\mathsf{0},\mathsf{1}\}^*$ and $w\mathsf{0}$ has an odd number of $\mathsf{0}$'s, so that $w\mathsf{0} \in Y$.

($\mathsf{A}, 1 \to \mathsf{A}$) Suppose $w \in \Lambda_{\mathsf{A}}$, and assume the inductive hypothesis: $w \in X$. Then $w1 \in X$.

($\mathsf{B}, 0 \to \mathsf{A}$) Suppose $w \in \Lambda_{\mathsf{B}}$, and assume the inductive hypothesis: $w \in Y$. Then $w0 \in X$.

($\mathsf{B}, 1 \to \mathsf{B}$) Suppose $w \in \Lambda_{\mathsf{B}}$, and assume the inductive hypothesis: $w \in Y$. Then $w1 \in Y$.

Because of

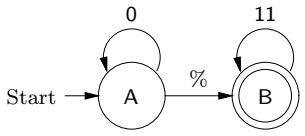
(A) for all $w \in \Lambda_{\mathsf{A}}$, $w \in X$, and

(B) for all $w \in \Lambda_{\mathsf{B}}$, $w \in Y$,

we have that $\Lambda_{\mathsf{A}} \subseteq X$ and $\Lambda_{\mathsf{B}} \subseteq Y$.

Because $X \subseteq \Lambda_{\mathsf{A}}$ and $Y \subseteq \Lambda_{\mathsf{B}}$, we can conclude that $L(M) = \Lambda_{\mathsf{A}} = X$ and $\Lambda_{\mathsf{B}} = Y$.

Consider our second example, N, again:



We can use induction on Λ to prove that

(A) for all $w \in \Lambda_{\mathsf{A}}$, $w \in \{0\}^*$; and

(B) for all $w \in \Lambda_{\mathsf{B}}$, $w \in \{0\}^*\{11\}^*$.

Thus $\Lambda_{\mathsf{A}} \subseteq \{0\}^*$ and $\Lambda_{\mathsf{B}} \subseteq \{0\}^*\{11\}^*$. Because $\{0\}^* \subseteq \Lambda_{\mathsf{A}}$ and $\{0\}^*\{11\}^* \subseteq \Lambda_{\mathsf{B}}$, we can conclude that $\Lambda_{\mathsf{A}} = \{0\}^*$ and $L(N) = \Lambda_{\mathsf{B}} = \{0\}^*\{11\}^*$.

3.8.4 Notes

Books on formal language theory typically give short shrift to the proof of correctness of finite automata, carrying out one or two correctness proofs using induction on the length of strings. In contrast, we have introduced and applied elegant techniques for proving the correctness of FAs. Of particular note is our principle of induction on Λ.

3.9 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); and
- 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 exponentially bigger than the smallest FA accepting the language.

3.9.1 Definition of EFAs

An *empty-string finite automaton* (EFA) M is a finite automaton such that

$$T_M \subseteq \{\, q, x \to r \mid q, r \in \mathbf{Sym} \text{ and } x \in \mathbf{Str} \text{ and } |x| \leq 1 \,\}.$$

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, $\mathsf{A}, \% \to \mathsf{B}$ and $\mathsf{A}, \mathsf{1} \to \mathsf{B}$ are legal EFA transitions, but $\mathsf{A}, \mathsf{11} \to \mathsf{B}$ is not legal. We write **EFA** for the set of all empty-string finite automata. Thus $\mathbf{EFA} \subsetneq \mathbf{FA}$.

The following proposition obviously holds.

Proposition 3.9.1
Suppose M is an EFA.

- *For all $N \in \mathbf{FA}$, 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.*

3.9.2 Converting FAs to EFAs

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

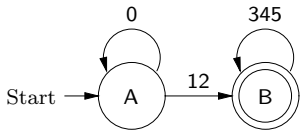
$$p, b_1 b_2 \cdots b_n \to r,$$

where $n \geq 2$ and the b_i are symbols, then we replace this transition with the n transitions

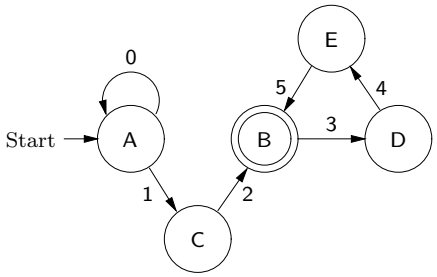
$$p \stackrel{b_1}{\to} q_1, q_1 \stackrel{b_2}{\to} q_2, \ \ldots, q_{n-1} \stackrel{b_n}{\to} r,$$

where $q_1, \ \ldots, q_{n-1}$ are $n-1$ new, non-accepting, states.

For example, we can convert the FA



into the EFA



We have to be careful how we choose our new states. The symbols we choose can't be states of the original machine, and we can't choose the same symbol twice. Instead of making a series of random choices, we will use structured symbols in such a way that one will be able to look at a resulting EFA and tell what the original FA was.

First, the algorithm renames each old state q to $\langle 1, q \rangle$. Then it can replace a transition

$$p \stackrel{b_1 b_2 \cdots b_n}{\to} r,$$

where $n \geq 2$ and the b_i are symbols, with the transitions

$$\langle 1, p \rangle \stackrel{b_1}{\to} \langle 2, \langle p, b_1, b_2 \cdots b_n, r \rangle \rangle,$$

$$\langle 2, \langle p, b_1, b_2 \cdots b_n, r \rangle \rangle \stackrel{b_2}{\to} \langle 2, \langle p, b_1 b_2, b_3 \cdots b_n, r \rangle \rangle,$$

$$\ldots,$$

$$\langle 2, \langle p, b_1 b_2 \cdots b_{n-1}, b_n, r \rangle \rangle \stackrel{b_n}{\to}, \langle 1, r \rangle.$$

We define a function **faToEFA** $\in$ **FA** $\rightarrow$ **EFA** that converts FAs into EFAs by saying that **faToEFA** M is the result of running the above algorithm on input M.

Theorem 3.9.2
For all $M \in$ **FA***:*

- **faToEFA** $M \approx M$*; and*
- **alphabet**(**faToEFA** M) = **alphabet** M.

3.9.3 Processing EFAs in Forlan

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:

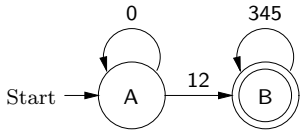
```
val faToEFA : fa -> efa
```

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 equal                 : efa * efa -> bool
val alphabet              : efa -> sym set
val checkLP               : efa -> lp -> unit
val validLP               : efa -> lp -> 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 accepted                : efa -> str -> bool
val findLP                  : efa -> sym set * str * sym set -> lp
val findAcceptingLP         : efa -> str -> lp
val simplified              : efa -> bool
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
```

3.9.4 Notes

The algorithm for converting FAs to EFAs is obvious, but our use of structured state names so as to make the resulting EFAs self-documenting is novel.

3.10 Nondeterministic Finite Automata

In this section, we study the second of our more restricted kinds of finite automata: nondeterministic finite automata.

3.10.1 Definition of NFAs

A *nondeterministic finite automaton* (NFA) M is a finite automaton such that

$$T_M \subseteq \{\, q, x \to r \mid q, r \in \mathbf{Sym} \text{ and } x \in \mathbf{Str} \text{ and } |x| = 1 \,\}.$$

In other words, an FA is an NFA iff every string of every transition of the FA has a single symbol. For example, $\mathsf{A}, 1 \to \mathsf{B}$ is a legal NFA transition, but $\mathsf{A}, \% \to \mathsf{B}$ and $\mathsf{A}, 11 \to \mathsf{B}$ are not legal. We write **NFA** for the set of all nondeterministic finite automata. Thus $\mathbf{NFA} \subsetneq \mathbf{EFA} \subsetneq \mathbf{FA}$.

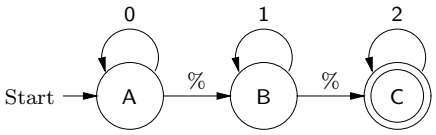
The following proposition obviously holds.

Proposition 3.10.1
Suppose M is an NFA.

- *For all $N \in \mathbf{FA}$, 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.*

3.10.2 Converting EFAs to NFAs

Suppose M is the EFA



To convert M into an equivalent NFA, we will have to:

- replace the transitions $\mathsf{A}, \% \to \mathsf{B}$ and $\mathsf{B}, \% \to \mathsf{C}$ with legal transitions (for example, because of the valid labeled path

$$\mathsf{A} \overset{\%}{\Rightarrow} \mathsf{B} \overset{1}{\Rightarrow} \mathsf{B} \overset{\%}{\Rightarrow} \mathsf{C},$$

  we will add the transition $\mathsf{A}, 1 \to \mathsf{C}$);

- make (at least) A be an accepting state (so that % is accepted by the NFA).

Before defining a 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

- $P \subseteq X$; and
- for all $q, r \in Q_M$, if $q \in X$ and $q, \% \to r \in T_M$, then $r \in X$.

We sometimes abbreviate $\mathbf{emptyClose}_M\, P$ to $\mathbf{emptyClose}\, P$, when M is clear from the context. 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 $\mathsf{A}, \% \to \mathsf{B} \in T_M$;
- $\mathsf{C} \in X$, since $\mathsf{B} \in X$ and $\mathsf{B}, \% \to \mathsf{C} \in T_M$.

Thus $\mathbf{emptyClose}\, P = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

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

- $P \subseteq X$; and
- for all $q, r \in Q_M$, if $r \in X$ and $q, \% \to r \in T_M$, then $q \in X$.

We sometimes drop the M from $\mathbf{emptyCloseBackwards}_M$, when it's clear from the context. 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 $\mathsf{B}, \% \to \mathsf{C} \in T_M$;
- $\mathsf{A} \in X$, since $\mathsf{B} \in X$ and $\mathsf{A}, \% \to \mathsf{B} \in T_M$.

Thus **emptyCloseBackwards** $P = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

Proposition 3.10.2
Suppose M is a finite automaton. For all $P \subseteq Q_M$,

$$\textbf{emptyClose}_M\, P = \Delta_M(P, \%).$$

In other words, $\textbf{emptyClose}_M\, P$ is all of the states that can be reached from elements of P by sequences of %-transitions.

Proposition 3.10.3
Suppose M is a finite automaton. For all $P \subseteq Q_M$,

$$\textbf{emptyCloseBackwards}_M\, P = \{\, q \in Q_M \mid \Delta_M(\{q\}, \%) \cap P \neq \emptyset \,\}.$$

In other words, $\textbf{emptyCloseBackwards}_M\, P$ is all of the states from which it is possible to reach elements of P by sequences of %-transitions.

We define a function/algorithm **efaToNFA** $\in$ **EFA** $\to$ **NFA** that converts EFAs into NFAs by saying that **efaToNFA** M is the NFA N such that:

- $Q_N = Q_M$;
- $s_N = s_M$;
- $A_N =$ **emptyCloseBackwards** A_M; and
- T_N is the set of all transitions $q', a \to r'$ such that $q', r' \in Q_M$, $a \in$ **Sym**, and there are $q, r \in Q_M$ such that:
  - $q, a \to r \in T_M$;
  - $q' \in$ **emptyCloseBackwards** $\{q\}$; and
  - $r' \in$ **emptyClose** $\{r\}$.

To compute the set T_N, we process each transition $q, x \to r$ of M as follows. If $x = \%$, then we generate no transitions. Otherwise, our transition is $q, a \to r$ 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

$$\{\, q', a \to r' \mid q' \in X \text{ and } r' \in Y \,\}$$

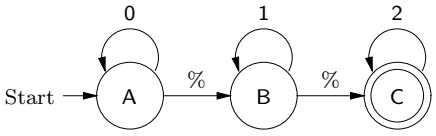
to T_N.

Because the algorithm defines A_N to be **emptyCloseBackwards** A_M, it could let T_N be the set of all transitions $q', a \to r$ such that $q', r \in Q_M$, $a \in$ **Sym**, and there is a $q \in Q_M$ such that:

- $q, a \to r \in T_M$; and
- $q' \in$ **emptyCloseBackwards** $\{q\}$.

This would mean that N would have fewer transitions. However, for esthetic reasons, we'll stick with the symmetric definition of T_N.

Let M be our example EFA



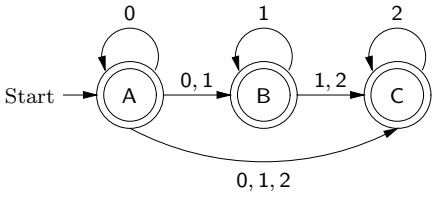
and let $N =$ **efaToNFA** M. Then

- $Q_N = Q_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$;
- $s_N = s_M = \mathsf{A}$;
- $A_N =$ **emptyCloseBackwards** $A_M =$ **emptyCloseBackwards** $\{\mathsf{C}\} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

Now, let's work out what T_N is, by processing each of M's transitions.

- From the transitions $\mathsf{A}, \% \to \mathsf{B}$ and $\mathsf{B}, \% \to \mathsf{C}$, we get no elements of T_N.
- Consider the transition $\mathsf{A}, 0 \to \mathsf{A}$. Since **emptyCloseBackwards** $\{\mathsf{A}\} = \{\mathsf{A}\}$ and **emptyClose** $\{\mathsf{A}\} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, we add $\mathsf{A}, 0 \to \mathsf{A}$, $\mathsf{A}, 0 \to \mathsf{B}$ and $\mathsf{A}, 0 \to \mathsf{C}$ to T_N.
- Consider the transition $\mathsf{B}, 1 \to \mathsf{B}$. Since **emptyCloseBackwards** $\{\mathsf{B}\} = \{\mathsf{A}, \mathsf{B}\}$ and **emptyClose** $\{\mathsf{B}\} = \{\mathsf{B}, \mathsf{C}\}$, we add $\mathsf{A}, 1 \to \mathsf{B}$, $\mathsf{A}, 1 \to \mathsf{C}$, $\mathsf{B}, 1 \to \mathsf{B}$ and $\mathsf{B}, 1 \to \mathsf{C}$ to T_N.
- Consider the transition $\mathsf{C}, 2 \to \mathsf{C}$. Since **emptyCloseBackwards** $\{\mathsf{C}\} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and **emptyClose** $\{\mathsf{C}\} = \{\mathsf{C}\}$, we add $\mathsf{A}, 2 \to \mathsf{C}$, $\mathsf{B}, 2 \to \mathsf{C}$ and $\mathsf{C}, 2 \to \mathsf{C}$ to T_N.

Thus our NFA N is



Theorem 3.10.4
For all $M \in$ **EFA***:*

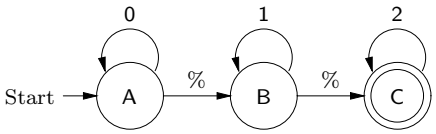
- **efaToNFA** $M \approx M$*; and*
- **alphabet**(**efaToNFA** M) = **alphabet** M.

3.10.3 Converting EFAs to NFAs, and Processing NFAs in Forlan

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
```

For example, if `fa` is bound to the finite automaton



then we can compute the empty-closure of {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 textttprojFromEFA are available in the top-level environment as textttinjNFAToFA, `injNFAToEFA`, `projFAToNFA` and textttprojEFAToNFA, 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
```

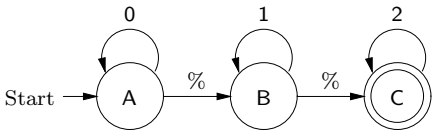
Most of the functions for processing FAs that were introduced in previous sections are inherited by `NFA`:

```
val output                  : string * nfa -> unit
val numStates               : nfa -> int
val numTransitions          : nfa -> int
val alphabet                : nfa -> sym set
val equal                   : nfa * nfa -> bool
val checkLP                 : nfa -> lp -> unit
val validLP                 : nfa -> lp -> 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 accepted                : nfa -> str -> bool
val findLP                  : nfa -> sym set * str * sym set -> lp
val findAcceptingLP         : nfa -> str -> lp
val simplified              : nfa -> bool
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
```

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
```

3.10.4 Notes

Because we have defined the meaning of finite automata via labeled paths instead of transition functions, our EFA to NFA conversion algorithm is easy to understand and prove correct.

3.11 Deterministic Finite Automata

In this section, we study the third of our more restricted kinds of finite automata: deterministic finite automata.

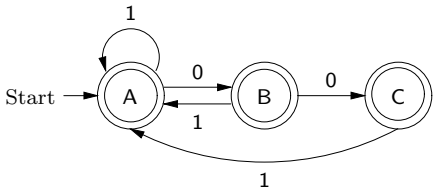
3.11.1 Definition of DFAs

A *deterministic finite automaton* (DFA) M is a finite automaton such that:

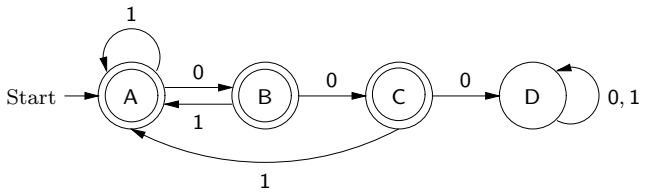
- $T_M \subseteq \{\, q, x \to r \mid q, r \in \textbf{Sym}$ and $x \in \textbf{Str}$ and $|x| = 1 \,\}$; and
- for all $q \in Q_M$ and $a \in \textbf{alphabet}\, M$, there is a unique $r \in Q_M$ such that $q, a \to r \in T_M$.

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 **DFA** $\subsetneq$ **NFA** $\subsetneq$ **EFA** $\subsetneq$ **FA**.

Let M be the finite automaton



It turns out that $L(M) = \{\, w \in \{0,1\}^* \mid 000$ is not a substring of $w \,\}$. Although M is an NFA, it's not a DFA, since $0 \in \textbf{alphabet}\, M$ but there is no transition of the form $C, 0 \to r$. However, 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 3.11.1
Suppose M is a DFA.

- *For all $N \in$* **FA**, *if M* **iso** *N, then N is a DFA.*
- *For all bijections f from Q_M to some set of symbols,* **renameStates**(M, f) *is a DFA.*
- **renameStatesCanonically** *M is a DFA.*

Now, we prove a proposition that doesn't hold for arbitary NFAs.

Proposition 3.11.2
*Suppose M is a DFA. For all $q \in Q_M$ and $w \in ($***alphabet** $M)^*$, $|\Delta_M(\{q\}, w)| = 1$.

Proof. An easy left string induction on w. □

Suppose M is a DFA. Because of Proposition 3.11.2, we can define *the transition function δ_M for M*, $\delta_M \in Q_M \times ($**alphabet** $M)^* \to Q_M$, by:

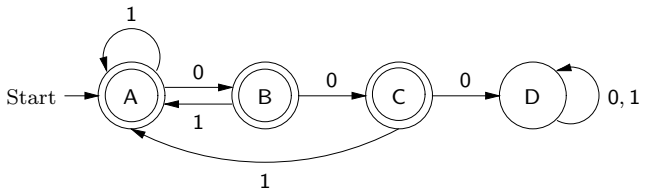
$$\delta_M(q, w) = \text{the unique } r \in Q_M \text{ such that } r \in \Delta_M(\{q\}, w).$$

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 $w \in ($**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 $\delta(\mathsf{A}, \%) = \mathsf{A}$, $\delta(\mathsf{A}, \mathsf{0100}) = \mathsf{C}$ and $\delta(\mathsf{B}, \mathsf{000100}) = \mathsf{D}$.

Having defined the δ function, we can study its properties.

Proposition 3.11.3
Suppose M is a DFA.

(1) For all $q \in Q_M$, $\delta_M(q, \%) = q$.

*(2) For all $q \in Q_M$ and $a \in$ **alphabet** M, $\delta_M(q, a) =$ the unique $r \in Q_M$ such that $q, a \to r \in T_M$.*

(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 proposition, we have that, for all $q, r \in Q_M$ and $a \in$ **alphabet** M,

$$\delta_M(q, a) = r \qquad \text{iff} \qquad q, a \to r \in T_M.$$

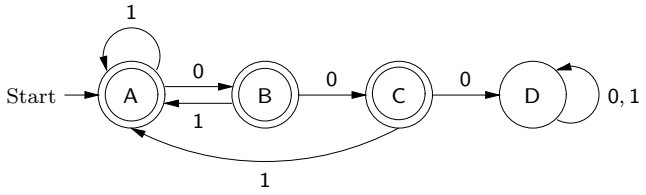
Now we can use the δ function to explain when a string is accepted by an FA.

Proposition 3.11.4
Suppose M is a DFA. $L(M) = \{\, w \in (\textbf{alphabet}\, M)^* \mid \delta_M(s_M, w) \in A_M \,\}$.

Proof. To see that the left-hand side is a subset of the right-hand side, suppose $w \in L(M)$. Then $w \in (\textbf{alphabet}\, M)^*$ and there is a $q \in A_M$ such that $q \in \Delta_M(\{s_M\}, w)$. Thus $\delta_M(s_M, w) = q \in A_M$.

To see that the right-hand side is a subset of the left-hand side, suppose $w \in (\textbf{alphabet}\, M)^*$ 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 $\mathsf{0100}$ is accepted by M, we need to determine whether $\delta(\mathsf{A}, \mathsf{0100}) \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

For instance, we have that:

$$\begin{aligned}\delta(\mathsf{A}, 0100) &= \delta(\delta(\mathsf{A}, 0), 100)\\ &= \delta(\mathsf{B}, 100)\\ &= \delta(\delta(\mathsf{B}, 1), 00)\\ &= \delta(\mathsf{A}, 00)\\ &= \delta(\delta(\mathsf{A}, 0), 0)\\ &= \delta(\mathsf{B}, 0)\\ &= \mathsf{C}\\ &\in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.\end{aligned}$$

Thus 0100 is accepted by M.

3.11.2 Proving the Correctness of DFAs

Since every DFA is an FA, we could prove the correctness of DFAs using the techniques that we have already studied. But it turns out that giving a separate proof that enough is accepted by a DFA is unnecessary—it will follow from the proof that everything accepted is wanted.

Proposition 3.11.5
Suppose M is a DFA. Then, for all $w \in (\textbf{alphabet}\, M)^*$ and $q \in Q_M$,

$$w \in \Lambda_{M,q} \qquad \textit{iff} \qquad \delta_M(s_M, w) = q.$$

Proof. Suppose $w \in (\textbf{alphabet}\, M)^*$ and $q \in Q_M$.

(only if) Suppose $w \in \Lambda_q$. Then $q \in \Delta(\{s\}, w)$. Thus $\delta(s, w) = q$.

(if) Suppose $\delta(s, w) = q$. Then $q \in \Delta(\{s\}, w)$, so that $w \in \Lambda_q$.

□

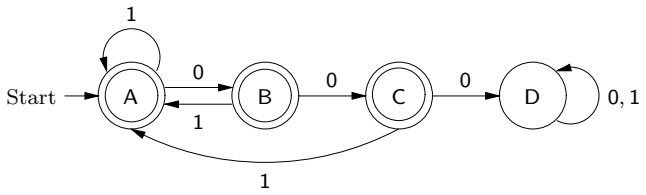
We already know that, if M is an FA, then $L(M) = \bigcup\{\, \Lambda_q \mid q \in A_M \,\}$.

Proposition 3.11.6
Suppose M is a DFA.

(1) $(\textbf{alphabet}\, M)^* = \bigcup\{\, \Lambda_q \mid q \in Q_M \,\}$.

(2) For all $q, r \in Q_M$, if $q \neq r$, then $\Lambda_q \cap \Lambda_r = \emptyset$.

Suppose M is the DFA



and let $X = \{ w \in \{0,1\}^* \mid 000 \text{ is not a substring of } w \}$. We will show that $L(M) = X$. Note that, for all $w \in \{0,1\}^*$:

- $w \in X$ iff 000 is not a substring of w.
- $w \notin X$ iff 000 is a substring of w.

First, we use induction on Λ, to prove that:

(A) for all $w \in \Lambda_A$, $w \in X$ and 0 is not a suffix of w;

(B) for all $w \in \Lambda_B$, $w \in X$ and 0, but not 00, is a suffix of w;

(C) for all $w \in \Lambda_C$, $w \in X$ and 00 is a suffix of w; and

(D) for all $w \in \Lambda_D$, $w \notin X$.

There are nine steps (1 + the number of transitions) to show.

(empty string) We must show that $\% \in X$ and 0 is not a suffix of %. This follows since % has no 0's.

(A, 0 → B) Suppose $w \in \Lambda_A$, and assume the inductive hypothesis: $w \in X$ and 0 is not a suffix of w. We must show that $w0 \in X$ and 0, but not 00, is a suffix of $w0$. Because $w \in X$ and 0 is not a suffix of w, we have that $w0 \in X$. Clearly, 0 is a suffix of $w0$. And, since 0 is not a suffix of w, we have that 00 is not a suffix of $w0$.

(A, 1 → A) Suppose $w \in \Lambda_A$, and assume the inductive hypothesis: $w \in X$ and 0 is not a suffix of w. We must show that $w1 \in X$ and 0 is not a suffix of $w1$. Since $w \in X$, we have that $w1 \in X$. And, 0 is not a suffix of $w1$.

(B, 0 → C) Suppose $w \in \Lambda_B$, and assume the inductive hypothesis: $w \in X$ and 0, but not 00, is a suffix of w. We must show that $w0 \in X$ and 00 is a suffix of $w0$. Because $w \in X$ and 00 is not suffix of w, we have that $w0 \in X$. And since 0 is a suffix of w, it follows that 00 is a suffix of $w0$.

(B, 1 → A) Suppose $w \in \Lambda_B$, and assume the inductive hypothesis: $w \in X$ and 0, but not 00, is a suffix of w. We must show that $w1 \in X$ and 0 is not a suffix of $w1$. Because $w \in X$, we have that $w1 \in X$. And, 0 is not a suffix of $w1$.

($\mathsf{C}, 0 \to \mathsf{D}$) Suppose $w \in \Lambda_{\mathsf{C}}$, and assume the inductive hypothesis: $w \in X$ and $\mathsf{00}$ is a suffix of w. We must show that $w\mathsf{0} \notin X$. Because $\mathsf{00}$ is a suffix of w, we have that $\mathsf{000}$ is a suffix of $w\mathsf{0}$. Thus $w\mathsf{0} \notin X$.

($\mathsf{C}, 1 \to \mathsf{A}$) Suppose $w \in \Lambda_{\mathsf{C}}$, and assume the inductive hypothesis: $w \in X$ and $\mathsf{00}$ is a suffix of w. We must show that $w\mathsf{1} \in X$ and $\mathsf{0}$ is not a suffix of $w\mathsf{1}$. Because $w \in X$, we have that $w\mathsf{1} \in X$. And, $\mathsf{0}$ is not a suffix of $w\mathsf{1}$.

($\mathsf{D}, 0 \to \mathsf{D}$) Suppose $w \in \Lambda_{\mathsf{D}}$, and assume the inductive hypothesis: $w \notin X$. We must show that $w\mathsf{0} \notin X$. Because $w \notin X$, we have that $w\mathsf{0} \notin X$.

($\mathsf{D}, 1 \to \mathsf{D}$) Suppose $w \in \Lambda_{\mathsf{D}}$, and assume the inductive hypothesis: $w \notin X$. We must show that $w\mathsf{1} \notin X$. Because $w \notin X$, we have that $w\mathsf{1} \notin X$.

Now, we use the result of our induction on Λ to show that $L(M) = X$.

($L(M) \subseteq X$) Suppose $w \in L(M)$. Because $A_M = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, we have that $w \in L(M) = \Lambda_{\mathsf{A}} \cup \Lambda_{\mathsf{B}} \cup \Lambda_{\mathsf{C}}$. Thus, by parts (A)–(C), we have that $w \in X$.

($X \subseteq L(M)$) Suppose $w \in X$. Since $X \subseteq \{0,1\}^*$, we have that $w \in \{0,1\}^*$. Suppose, toward a contradiction, that $w \notin L(M)$. Because $w \notin L(M) = \Lambda_{\mathsf{A}} \cup \Lambda_{\mathsf{B}} \cup \Lambda_{\mathsf{C}}$ and $w \in \{0,1\}^* = (\mathbf{alphabet}\, M)^* = \Lambda_{\mathsf{A}} \cup \Lambda_{\mathsf{B}} \cup \Lambda_{\mathsf{C}} \cup \Lambda_{\mathsf{D}}$, we must have that $w \in \Lambda_{\mathsf{D}}$. But then part (D) tells us that $w \notin X$—contradiction. Thus $w \in L(M)$.

For the above approach to work, when proving $L(M) = X$ for a DFA M and language X, we simply need that:

- the property associated with each accepting state implies being in X; and
- the property associated with each non-accepting state implies not being in X.

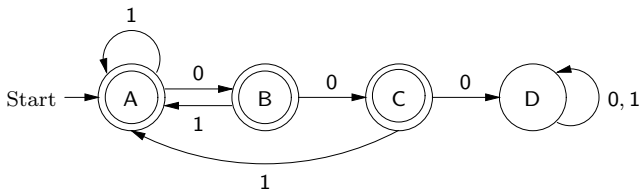
Exercise 3.11.7
Define $\mathbf{diff} \in \{0,1\}^* \to \mathbb{Z}$ by: for all $w \in \{0,1\}^*$,

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

Define **AllPrefixGood** to be the language $\{\, w \in \{0,1\}^* \mid \text{for all prefixes } v \text{ of } w, |\mathbf{diff}\, v| \leq 2 \,\}$. Find a DFA **allPrefixGoodDFA** such that $L(\mathbf{allPrefixGoodDFA}) = \mathbf{AllPrefixGood}$, and prove that your solution is correct.

3.11.3 Simplification of DFAs

Let M be our example DFA



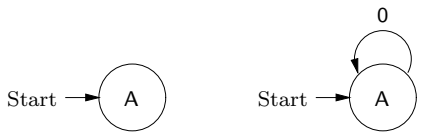
Then M is not simplified, since the state D is dead. But if we get rid of D, then we won't have a DFA anymore. Thus, we will need:

- a notion of when a DFA is simplified that is more liberal than our standard notion; and
- 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, both of the following DFAs, which accept $\emptyset$, are deterministically simplified:



(M_1) (M_2)

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,
- $N \approx M$, and
- **alphabet** $N =$ **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.

The algorithm begins by letting the FA M' be **simplify** M, i.e., the result of running our simplification algorithm for FAs on M. M' will have the following properties.

- M' is simplified.
- $M' \approx M$.
- $\textbf{alphabet}\, M' = \textbf{alphabet}(L(M')) = \textbf{alphabet}(L(M))$.
- For all $q \in Q_{M'}$ and $a \in \textbf{alphabet}\, M'$, there is at most one $r \in Q_{M'}$ such that $q, a \to r \in T_{M'}$. This property holds since M is a DFA and M' was formed by removing states and transitions from M.

Let $\Sigma' = \textbf{alphabet}\, M' \cup \Sigma = \textbf{alphabet}(L(M)) \cup \Sigma$. If M' is a DFA and $\textbf{alphabet}\, M' = \Sigma'$, then the algorithm returns M'. Otherwise, it must turn M' into a DFA whose alphabet is Σ'. We have that

- $\textbf{alphabet}\, M' \subseteq \Sigma'$; and
- for all $q \in Q_{M'}$ and $a \in \Sigma'$, there is at most one $r \in Q_{M'}$ such that $q, a \to r \in T_{M'}$.

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 the algorithm can return the DFA N defined by:

- $Q_N = Q_{M'} = \{s_{M'}\}$;
- $s_N = s_{M'}$;
- $A_N = A_{M'} = \emptyset$; and
- $T_N = \{\, s_{M'}, a \to s_{M'} \mid a \in \Sigma' \,\}$.

Alternatively, M' has at least one accepting state. Thus, M' has no dead states. So it can return the DFA N defined by:

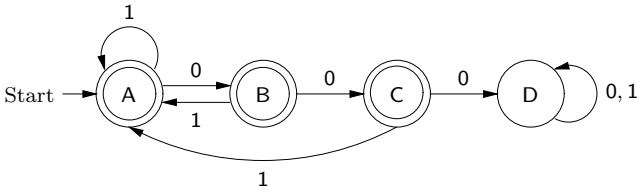
- $Q_N = Q_{M'} \cup \{\langle \mathsf{dead} \rangle\}$ (we put enough brackets around $\langle \mathsf{dead} \rangle$ so that it's not in $Q_{M'}$);
- $s_N = s_{M'}$;
- $A_N = A_{M'}$; and
- $T_N = T_{M'} \cup T'$, where T' is the set of all transitions $q, a \to \langle \mathsf{dead} \rangle$ such that either
  - $q \in Q_{M'}$ and $a \in \Sigma'$, but there is no $r \in Q_{M'}$ such that $q, a \to r \in T_{M'}$; or
  - $q = \langle \mathsf{dead} \rangle$ and $a \in \Sigma'$.

We define a function **determSimplify** $\in$ **DFA** $\times$ **Alp** $\to$ **DFA** by: **determSimplify**(M, Σ) is the result of running the above algorithm on M and Σ.

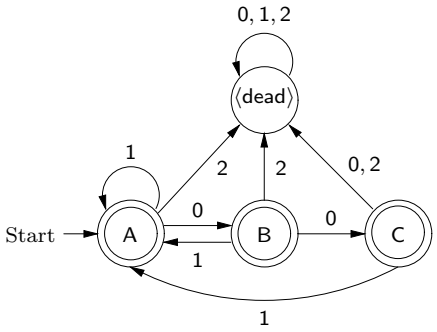
Theorem 3.11.8
For all $M \in$ **DFA** *and* $\Sigma \in$ **Alp***:*

- **determSimplify**(M, Σ) *is deterministically simplified;*
- **determSimplify**$(M, \Sigma) \approx M$*; and*
- **alphabet**(**determSimplify**(M, Σ)) = **alphabet**$(L(M)) \cup \Sigma$.

For example, suppose M is the DFA

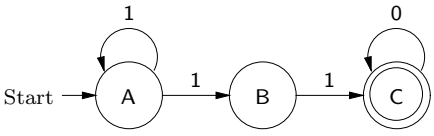


Then **determSimplify**$(M, \{2\})$ is the DFA



3.11.4 Converting NFAs to DFAs

Suppose M is the NFA



Our algorithm for converting NFAs to DFAs will convert M into a DFA N whose states represent the elements of the set

$$\{\, \Delta_M(\{\mathsf{A}\}, w) \mid w \in \{0,1\}^* \,\}.$$

For example, one the states of N will be $\langle \mathsf{A}, \mathsf{B} \rangle$, which represents $\{\mathsf{A}, \mathsf{B}\} = \Delta_M(\{\mathsf{A}\}, 1)$. This is the state that our DFA will be in after processing 1 from the start state.

Before describing our conversion algorithm, we first state a proposition concerning the Δ function for NFAs, and say how we will represent finite sets of symbols as symbols.

Proposition 3.11.9
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,*

$$\Delta_M(P, a) = \{\, r \in Q_M \mid p, a \to r \in T_M, \text{ for some } p \in P \,\}.$$

(3) For all $P \subseteq Q_M$ and $x, y \in (\textbf{alphabet}\, M)^*$,

$$\Delta_M(P, xy) = \Delta_M(\Delta_M(P, x), y).$$

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 total 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:

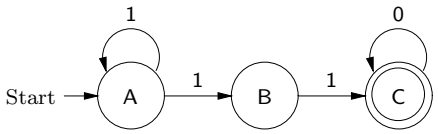
- $\{s_M\} \in X$; and
- for all $P \in X$ and $a \in$ **alphabet** M, $\Delta_M(P, a) \in X$.

Thus $|X| \leq 2^{|Q_M|}$. Then we define the DFA N as follows:

- $Q_N = \{\, \overline{P} \mid P \in X \,\}$;
- $s_N = \overline{\{s_M\}} = \langle s_M \rangle$;
- $A_N = \{\, \overline{P} \mid P \in X \text{ and } P \cap A_M \neq \emptyset \,\}$; and
- $T_N = \{\, (\overline{P}, a, \overline{\Delta_M(P, a)}) \mid P \in X \text{ and } a \in \textbf{alphabet}\, M \,\}$.

Then N is a DFA with alphabet **alphabet** M and, for all $P \in X$ and $a \in$ **alphabet** M, $\delta_N(\overline{P}, a) = \overline{\Delta_M(P, a)}$.

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 $\mathsf{A} \notin A_M$.

- Since $\{\mathsf{A}\} \in X$, and $\Delta(\{\mathsf{A}\}, \mathsf{0}) = \emptyset$, we add $\emptyset$ to X, $\langle\rangle$ to Q_N and $\langle \mathsf{A} \rangle, \mathsf{0} \to \langle\rangle$ 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 $\langle \mathsf{A} \rangle, \mathsf{1} \to \langle \mathsf{A}, \mathsf{B} \rangle$ 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 $\langle\rangle, \mathsf{0} \to \langle\rangle$ 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 $\langle\rangle, \mathsf{1} \to \langle\rangle$ 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 $\langle \mathsf{A}, \mathsf{B} \rangle, \mathsf{0} \to \langle\rangle$ 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 $\langle \mathsf{A}, \mathsf{B} \rangle, \mathsf{1} \to \langle \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle$ 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.

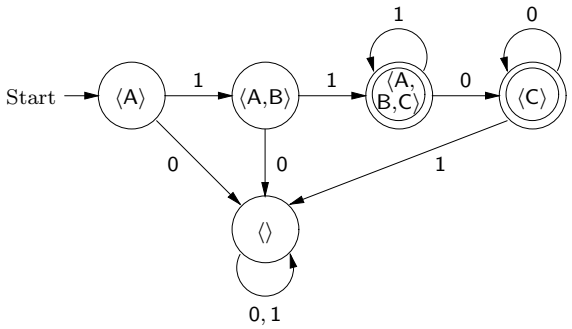
- Since $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\} \in X$ and $\Delta(\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}, \mathsf{0}) = \emptyset \cup \emptyset \cup \{\mathsf{C}\} = \{\mathsf{C}\}$, we add $\{\mathsf{C}\}$ to X, $\langle \mathsf{C} \rangle$ to Q_N and $\langle \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \mathsf{0} \to \langle \mathsf{C} \rangle$ 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 $\langle \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \mathsf{1} \to \langle \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle$ 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 $\langle \mathsf{C} \rangle, \mathsf{0} \to \langle \mathsf{C} \rangle$ 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 $\langle \mathsf{C} \rangle, \mathsf{1} \to \langle\rangle$ 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 3.11.10
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

$$\begin{aligned}
\delta_N(s_N, wa) &= \delta_N(\delta_N(s_N, w), a) \\
&= \delta_N(\overline{\Delta_M(\{s_M\}, w)}, a) \qquad \text{(ind. hyp.)} \\
&= \overline{\Delta_M(\Delta_M(\{s_M\}, w), a)} \\
&= \overline{\Delta_M(\{s_M\}, wa)}.
\end{aligned}$$

□

Lemma 3.11.11
$L(N) = L(M)$.

Proof.

($L(M) \subseteq L(N)$) Suppose $w \in L(M)$, so that $w \in (\textbf{alphabet}\, M)^* = (\textbf{alphabet}\, N)^*$ and $\Delta_M(\{s_M\}, w) \cap A_M \neq \emptyset$. By Lemma 3.11.10, 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 3.11.10, 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 $w \in L(M)$.

□

We define a function $\mathbf{nfaToDFA} \in \mathbf{NFA} \to \mathbf{DFA}$ by: $\mathbf{nfaToDFA}\, M$ is the result of running the preceding algorithm with input M.

Theorem 3.11.12
For all $M \in \mathbf{NFA}$:

- $\mathbf{nfaToDFA}\, M \approx M$; *and*
- $\mathbf{alphabet}(\mathbf{nfaToDFA}\, M) = \mathbf{alphabet}\, M$.

3.11.5 Processing DFAs in Forlan

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 injToEFA    : dfa -> efa
val injToNFA    : dfa -> nfa
val projFromFA  : fa -> dfa
val projFromEFA : efa -> dfa
val projFromNFA : nfa -> dfa
```

These functions are available in the top-level environment with the names `injDFAToFA`, `injDFAToEFA`, `injDFAToNFA`, `projFAToDFA`, `projEFAToDFA` and `projNFAToDFA`.

The module `DFA` also defines the functions:

```
val input            : string -> dfa
val determProcStr    : dfa -> sym * str -> sym
val determAccepted   : dfa -> str -> bool
val determSimplified : dfa -> bool
val determSimplify   : dfa * sym set -> dfa
val fromNFA          : nfa -> dfa
```

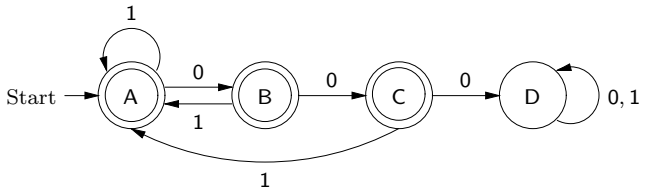
The function `input` is used to input a DFA. The function `determProcStr` is used to compute $\delta_M(q, w)$ for a DFA M, using the properties of δ_M. The function `determAccepted` uses `determProcStr` to check whether a string is accepted by a DFA. The function `determSimplified` tests whether a DFA is deterministically simplified, and 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 checkLP                 : dfa -> lp -> unit
val validLP                 : dfa -> lp -> 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 accepted                : dfa -> str -> bool
val findLP                  : dfa -> sym set * str * sym set -> lp
val findAcceptingLP         : dfa -> str -> lp
```

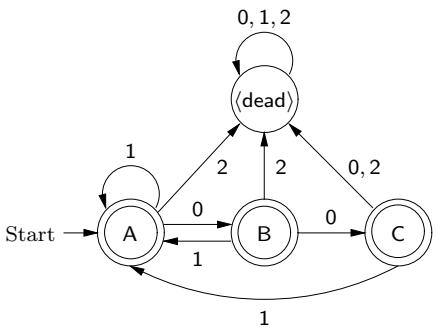
Suppose `dfa` is the DFA



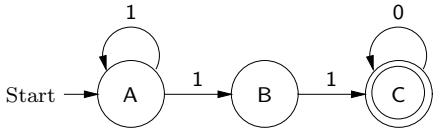
We can turn `dfa` into an equivalent deterministically simplified DFA whose alphabet is the union of the alphabet of the language of `dfa` and $\{2\}$, i.e., whose alphabet is $\{0, 1, 2\}$, as follows:

```
- 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



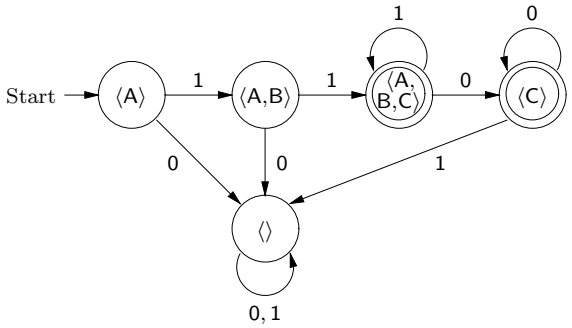
Suppose that `nfa` is the NFA



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

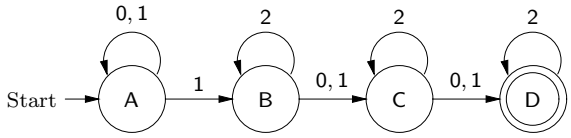


And we can see why `nfa` and `dfa` accept `111100`, as follows:

```
- LP.output
= ("",
=  NFA.findAcceptingLP nfa (Str.fromString "111100"));
A, 1 => A, 1 => A, 1 => B, 1 => C, 0 => C, 0 => C
```

```
val it = () : unit
- LP.output
= ("",
=  DFA.findAcceptingLP dfa (Str.fromString "111100"));
<A>, 1 => <A,B>, 1 => <A,B,C>, 1 => <A,B,C>, 1 => <A,B,C>, 0 =>
<C>, 0 => <C>
val it = () : unit
```

Finally, we see an example in which an NFA with 4 states is converted to a DFA with $2^4 = 16$ states; there is a state of the DFA corresponding to every element of the power set of the set of states of the NFA. Suppose `nfa'` is the NFA



Then we can convert `nfa'` into a DFA, as follows:

```
- val dfa' = nfaToDFA nfa';
val dfa' = - : dfa
- DFA.numStates dfa';
val it = 16 : int
- DFA.output("", dfa');
{states}
<>, <A>, <B>, <C>, <D>, <A,B>, <A,C>, <A,D>, <B,C>, <B,D>, <C,D>,
<A,B,C>, <A,B,D>, <A,C,D>, <B,C,D>, <A,B,C,D>
{start state} <A>
{accepting states}
<D>, <A,D>, <B,D>, <C,D>, <A,B,D>, <A,C,D>, <B,C,D>, <A,B,C,D>
{transitions}
<>, 0 -> <>; <>, 1 -> <>; <>, 2 -> <>; <A>, 0 -> <A>;
<A>, 1 -> <A,B>; <A>, 2 -> <>; <B>, 0 -> <C>; <B>, 1 -> <C>;
<B>, 2 -> <B>; <C>, 0 -> <D>; <C>, 1 -> <D>; <C>, 2 -> <C>;
<D>, 0 -> <>; <D>, 1 -> <>; <D>, 2 -> <D>; <A,B>, 0 -> <A,C>;
<A,B>, 1 -> <A,B,C>; <A,B>, 2 -> <B>; <A,C>, 0 -> <A,D>;
<A,C>, 1 -> <A,B,D>; <A,C>, 2 -> <C>; <A,D>, 0 -> <A>;
<A,D>, 1 -> <A,B>; <A,D>, 2 -> <D>; <B,C>, 0 -> <C,D>;
<B,C>, 1 -> <C,D>; <B,C>, 2 -> <B,C>; <B,D>, 0 -> <C>;
<B,D>, 1 -> <C>; <B,D>, 2 -> <B,D>; <C,D>, 0 -> <D>;
<C,D>, 1 -> <D>; <C,D>, 2 -> <C,D>; <A,B,C>, 0 -> <A,C,D>;
<A,B,C>, 1 -> <A,B,C,D>; <A,B,C>, 2 -> <B,C>; <A,B,D>, 0 -> <A,C>;
<A,B,D>, 1 -> <A,B,C>; <A,B,D>, 2 -> <B,D>; <A,C,D>, 0 -> <A,D>;
<A,C,D>, 1 -> <A,B,D>; <A,C,D>, 2 -> <C,D>; <B,C,D>, 0 -> <C,D>;
<B,C,D>, 1 -> <C,D>; <B,C,D>, 2 -> <B,C,D>;
<A,B,C,D>, 0 -> <A,C,D>; <A,B,C,D>, 1 -> <A,B,C,D>;
<A,B,C,D>, 2 -> <B,C,D>
val it = () : unit
```

In Section 3.13, we will use Forlan to show that there is no DFA with fewer than 16 states that accepts the language accepted by `nfa'` and `dfa'`.

3.11.6 Notes

In contrast to the standard approach, the transition function δ for a DFA M is not part of the definition of M, but is derived from the definition. Our approach to proving the correctness of DFAs, using induction on Λ plus proof by contradiction, is novel, simple and elegant. The material on deterministic simplification is original, but straightforward. And the algorithm for converting NFAs to DFAs is standard.

3.12 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; and
- 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; and
- an operation on DFAs corresponding to set difference.

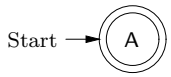
As a result, we will have that the set **RegLan** of regular languages is closed under union, concatenation, closure, intersection and set difference. I.e., we will have that, if $L, L_1, L_2 \in \mathbf{RegLan}$, then $L_1 \cup L_2$, $L_1 L_2$, L^*, $L_1 \cap L_2$ and $L_1 - L_2$ are in **RegLan**.

We will also show several additional closure properties of regular languages, in addition to giving the corresponding operations on regular expressions and automata.

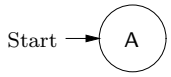
3.12.1 Converting Regular Expressions to FAs

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 *canonical finite automaton for* %,

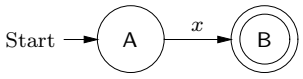


And we write **emptySet** for *canonical finite automaton for* $\emptyset$,



Thus, we have that $L(\mathbf{emptyStr}) = \{\%\}$ and $L(\mathbf{emptySet}) = \emptyset$. Furthermore both **emptyStr** and **emptySet** are DFAs, so that they are also NFAs and EFAs. Thus, we also refer to **emptyStr** as the *canonical DFA/NFA/EFA/FA for* %, and **emptySet** as the *canonical DFA/NFA/EFA/FA for* $\emptyset$.

Next, we define a function $\mathbf{strToFA} \in \mathbf{Str} \to \mathbf{FA}$ by: $\mathbf{strToFA}\, x$ is the *canonical finite automaton for* x,



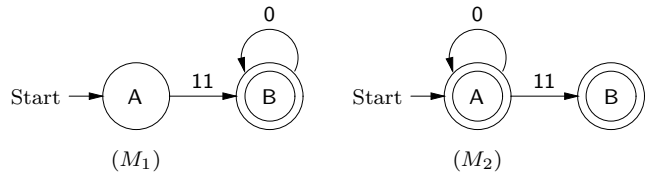
Thus, for all $x \in \mathbf{Str}$, $L(\mathbf{strToFA}\, x) = \{x\}$. It is also convenient to define a function $\mathbf{symToNFA} \in \mathbf{Sym} \to \mathbf{NFA}$ by: $\mathbf{symToNFA}\, a = \mathbf{strToFA}\, a$. Then, for all $a \in \mathbf{Sym}$, $L(\mathbf{symToNFA}\, a) = \{a\}$. Of course, **symToNFA** is also an element of $\mathbf{Sym} \to \mathbf{EFA}$ and $\mathbf{Sym} \to \mathbf{FA}$, and we say that $\mathbf{symToNFA}\, a$ is the *canonical NFA/EFA/FA for* a.

Next, we define a function/algorithm $\mathbf{union} \in \mathbf{FA} \times \mathbf{FA} \to \mathbf{FA}$ such that $L(\mathbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \mathbf{FA}$. If $M_1, M_2 \in \mathbf{FA}$, then $\mathbf{union}(M_1, M_2)$, *the union of* M_1 *and* M_2, is the FA N such that:

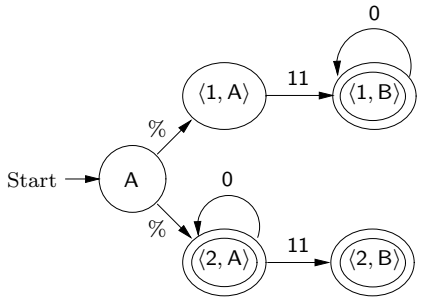
- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\, \langle 1, q\rangle \mid q \in A_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

$$
\begin{aligned}
&\{\mathsf{A}, \% \to \langle 1, s_{M_1}\rangle\} \\
\cup\, &\{\mathsf{A}, \% \to \langle 2, s_{M_2}\rangle\} \\
\cup\, &\{\, \langle 1, q\rangle, a \to \langle 1, r\rangle \mid q, a \to r \in T_{M_1} \,\} \\
\cup\, &\{\, \langle 2, q\rangle, a \to \langle 2, r\rangle \mid q, a \to r \in T_{M_2} \,\}.
\end{aligned}
$$

For example, if M_1 and M_2 are the FAs



then **union**(M_1, M_2) is the FA



Proposition 3.12.1
For all $M_1, M_2 \in$ **FA**:

- $L(\mathbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$; *and*
- $\mathbf{alphabet}(\mathbf{union}(M_1, M_2)) = \mathbf{alphabet}\, M_1 \cup \mathbf{alphabet}\, M_2$.

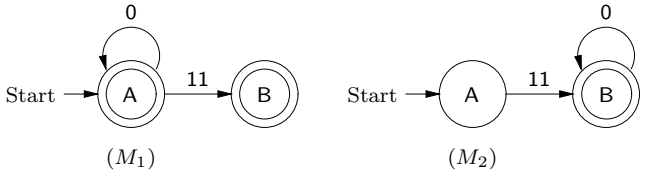
Proposition 3.12.2
For all $M_1, M_2 \in$ **EFA**, $\mathbf{union}(M_1, M_2) \in$ **EFA**.

Next, we define a function/algorithm $\mathbf{concat} \in \mathbf{FA} \times \mathbf{FA} \to \mathbf{FA}$ such that $L(\mathbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in$ **FA**. If $M_1, M_2 \in$ **FA**, then $\mathbf{concat}(M_1, M_2)$, *the concatentation of* M_1 *and* M_2, is the FA N such that:

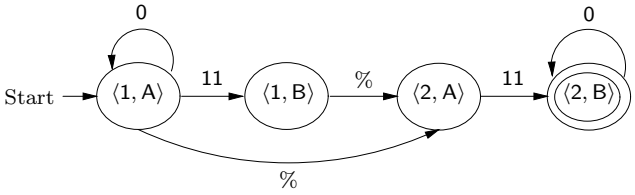
- $Q_N = \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \langle 1, s_{M_1}\rangle$;
- $A_N = \{\, \langle 2, q\rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

$$\begin{aligned} &\{\, \langle 1, q\rangle, \% \to \langle 2, s_{M_2}\rangle \mid q \in A_{M_1} \,\} \\ \cup\, &\{\, \langle 1, q\rangle, a \to \langle 1, r\rangle \mid q, a \to r \in T_{M_1} \,\} \\ \cup\, &\{\, \langle 2, q\rangle, a \to \langle 2, r\rangle \mid q, a \to r \in T_{M_2} \,\}. \end{aligned}$$

For example, if M_1 and M_2 are the FAs



then **concat**(M_1, M_2) is the FA



Proposition 3.12.3
For all $M_1, M_2 \in$ **FA***:*

- $L(\mathbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$*; and*
- **alphabet**(**concat**(M_1, M_2)) = **alphabet** $M_1 \cup$ **alphabet** M_2.

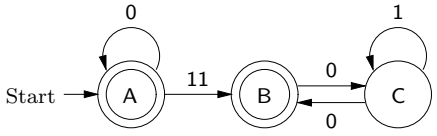
Proposition 3.12.4
For all $M_1, M_2 \in$ **EFA**, **concat**$(M_1, M_2) \in$ **EFA**.

Next, we define a function/algorithm **closure** $\in$ **FA** $\to$ **FA** such that $L($**closure** $M) = L(M)^*$, for all $M \in$ **FA**. If $M \in$ **FA**, then **closure** M, *the closure of* M, is the FA N such that:

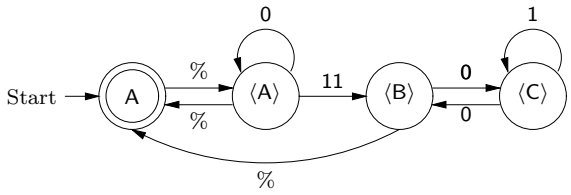
- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\mathsf{A}\}$; and
- $T_N =$

$$\begin{aligned} &\{\mathsf{A}, \% \to \langle s_M \rangle\} \\ \cup\, &\{\, \langle q \rangle, \% \to \mathsf{A} \mid q \in A_M \,\} \\ \cup\, &\{\, \langle q \rangle, a \to \langle r \rangle \mid q, a \to r \in T_M \,\}. \end{aligned}$$

For example, if M is the FA



then **closure** M is the FA



Proposition 3.12.5
For all $M \in$ **FA**,

- $L(\mathbf{closure}\, M) = L(M)^*$; *and*
- $\mathbf{alphabet}(\mathbf{closure}\, M) = \mathbf{alphabet}\, M$.

Proposition 3.12.6
For all $M \in$ **EFA**, $\mathbf{closure}\, M \in$ **EFA**.

We define a function/algorithm $\mathbf{regToFA} \in \mathbf{Reg} \to \mathbf{FA}$ by well-founded recursion on the height of regular expressions, as follows. The goal is for $L(\mathbf{regToFA}\, \alpha)$ to be equal to $L(\alpha)$, for all regular expressions α.

- $\mathbf{regToFA}\, \% = \mathbf{emptyStr}$;
- $\mathbf{regToFA}\, \$ = \mathbf{emptySet}$;
- for all $\alpha \in \mathbf{Reg}$, $\mathbf{regToFA}(\alpha^*) = \mathbf{closure}(\mathbf{regToFA}\, \alpha)$;
- for all $\alpha, \beta \in \mathbf{Reg}$, $\mathbf{regToFA}(\alpha + \beta) = \mathbf{union}(\mathbf{regToFA}\, \alpha, \mathbf{regToFA}\, \beta)$;
- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \mathbf{Sym}$, $\mathbf{regToFA}(a_1 \cdots a_n) = \mathbf{strToFA}(a_1 \cdots a_n)$;
- 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 doesn't have the form $b\,\beta$ for some $b \in \mathbf{Sym}$ and $\beta \in \mathbf{Reg}$, then $\mathbf{regToFA}(a_1 \cdots a_n\, \alpha) = \mathbf{concat}(\mathbf{strToFA}(a_1 \cdots a_n), \mathbf{regToFA}\, \alpha)$; and
- for all $\alpha, \beta \in \mathbf{Reg}$, if α doesn't consist of a single symbol, then $\mathbf{regToFA}(\alpha\beta) = \mathbf{concat}(\mathbf{regToFA}\, \alpha, \mathbf{regToFA}\, \beta)$.

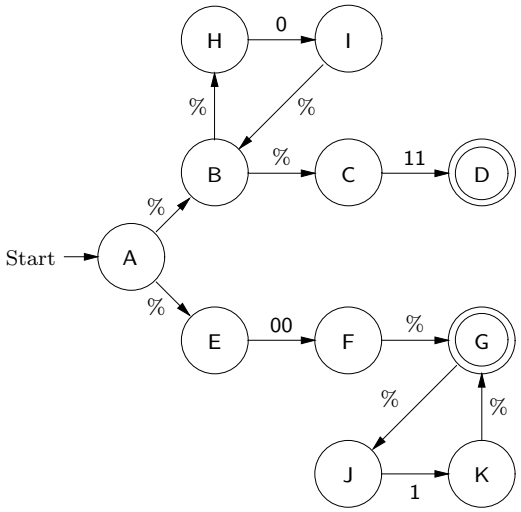
For example, $\mathbf{regToFA}(0101^*) = \mathbf{concat}(\mathbf{strToFA}(010), \mathbf{regToFA}(1^*))$.

Theorem 3.12.7
For all $\alpha \in$ **Reg**:

- $L(\mathbf{regToFA}\, \alpha) = L(\alpha)$; *and*
- $\mathbf{alphabet}(\mathbf{regToFA}\, \alpha) = \mathbf{alphabet}\, \alpha$.

Proof. Because of the form of recursion used, the proof uses well-founded induction on the height of α. □

For example, **regToFA**$(0^*11 + 001^*)$ is isomorphic to the FA



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
```

`emptyStr` and `emptySet` correspond to **emptyStr** and **emptySet**, respectively. The functions `fromStr` and `fromSym` correspond to **strToFA** and **symToNFA**, and are also available in the top-level environment with the names

```
val strToFA : str -> fa
val symToFA : sym -> fa
```

`union` and `concat` and `closure` correspond to **union**, **concat** and **closure**, respectively. The function `fromReg` corresponds to **regToFA** and is available in the top-level environment with that name:

```
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`, and is available in the top-level environment with the names

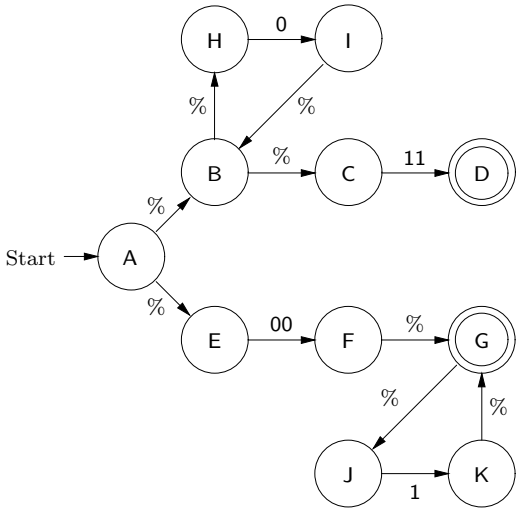
```
val symToNFA : sym -> nfa
val symToEFA : sym -> efa
```

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



Putting together our algorithm for converting regular expressions to finite automata with our algorithm for checking whether strings are accepted by finite automata, we are now able to check whether strings are generated by regular expressions:

```
- fun generated reg =
=       let val fa = FA.renameStatesCanonically(regToFA reg)
=       in FA.accepted fa end;
val generated = fn : reg -> str -> bool
- val generated = generated reg;
val generated = fn : str -> bool
- generated(Str.fromString "000011");
val it = true : bool
- generated(Str.fromString "001111");
val it = true : bool
- generated(Str.fromString "000111");
val it = false : bool
```

3.12.2 Converting FAs to Regular Expressions

Our algorithm for converting FAs to regular expressions makes use of a more general kind of finite automata that we call regular expression finite automata.

A *regular expression finite automaton* (RFA) M consists of:

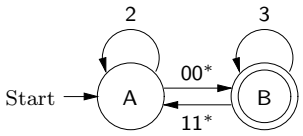
- a finite set Q_M of symbols;
- an element s_M of Q_M;
- a subset A_M of Q_M; and
- a finite subset T_M of $\{\,(q, \alpha, r) \mid q, r \in Q_M \text{ and } \alpha \in \mathbf{Reg}\,\}$ such that, for all $q, r \in Q_M$, there is at most one $\alpha \in \mathbf{Reg}$ such that $(q, \alpha, r) \in T_M$.

As usual Q_M consists of M's *states*, s_M is M's *start state*, A_M consists of M's *accepting states*, and T_M consists of M's *transitions*. We often write a transition (q, α, r) as

$$q \overset{\alpha}{\to} r$$

or $q, \alpha \to r$. We write **RFA** for the set of all RFAs, which is a countably infinite set. RFAs are drawn analogously to FAs, and the Forlan syntax for RFAs is analogous to that of FAs.

For example, the RFA M whose states are A and B, start state is A, only accepting state is B, and transitions are $(\mathsf{A}, \mathsf{2}, \mathsf{A})$, $(\mathsf{A}, \mathsf{00}^*, \mathsf{B})$, $(\mathsf{B}, \mathsf{3}, \mathsf{B})$ and $(\mathsf{B}, \mathsf{11}^*, \mathsf{A})$ can be drawn as



and expressed in Forlan as

```
{states} A, B {start state} A {accepting states} B
{transitions} A, 2 -> A; A, 00* -> B; B, 3 -> B; B, 11* -> A
```

We define a function **alphabet** $\in$ **RFA** $\to$ **Alp** by: for all $M \in$ **RFA**, **alphabet** M is $\{\, a \in$ **Sym** $\mid$ there are q, α, r such that $q, \alpha \to r \in T_M$ and $a \in$ **alphabet** $\alpha \,\}$. I.e., **alphabet** M is the union of the alphabets of all of the regular expressions appearing in M's transitions. We say that **alphabet** M is *the alphabet of* M. For example, the alphabet of our example FA M is $\{\mathsf{0}, \mathsf{1}, \mathsf{2}\}$.

The Forlan module `RFA` defines an abstract type `rfa` (in the top-level environment) of regular expression finite automata, as well as some functions for processing RFAs including:

```
val input          : string -> rfa
val output         : string * rfa -> unit
val alphabet       : rfa -> sym set
val numStates      : rfa -> int
val numTransitions : rfa -> int
val equal          : rfa * rfa -> bool
```

JForlan can be used to view and edit regular expression finite automata. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

The isomorphism relation between RFAs is defined in an analogous way to this relation for FAs. And the functions **renameStates** and **renameStatesCanonically** are also defined analogously, and have analogous properties. The `RFA` module has the functions

```
val renameStates            : rfa * sym_rel -> rfa
val renameStatesCanonically : rfa -> rfa
```

A labeled path

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

is *valid for* an RFA M iff, for all $i \in [1:n]$,

$$x_i \in L(\alpha), \text{ for some } \alpha \in \mathbf{Reg} \text{ such that } q_i, \alpha \to q_{i+1},$$

and $q_{n+1} \in Q_M$. For example, the labeled path

$$\mathsf{A} \overset{\mathsf{000}}{\Rightarrow} \mathsf{B} \overset{\mathsf{3}}{\Rightarrow} \mathsf{B}$$

is valid for our example FA M, because

- $\mathsf{000} \in L(\mathsf{00}^*)$ and $\mathsf{A}, \mathsf{00}^* \to \mathsf{B} \in T$, and
- $\mathsf{3} \in L(\mathsf{3})$ and $\mathsf{B}, \mathsf{3} \to \mathsf{B} \in T$.

The `RFA` module contains the functions

```
val checkLP : (str * reg -> bool) * rfa -> lp -> unit
val validLP : (str * reg -> bool) * rfa -> lp -> bool
```

which are analogous to the identically named functions provided by `FA`, except that they take a first argument whose job is to test whether a string is generated by a regular expression. For example, we can proceed as follows:

```
- val rfa = RFA.input "";
@ {states} A, B {start state} A {accepting states} B
@ {transitions} A, 2 -> A; A, 00* -> B; B, 3 -> B; B, 11* -> A
@ .
val rfa = - : rfa
- fun memb(x, reg) = FA.accepted (regToFA reg) x;
val memb = fn : str * reg -> bool
- val lp = LP.input "";
@ A, 000 => B, 3 => B
@ .
val lp = - : lp
- RFA.validLP (memb, rfa) lp;
val it = true : bool
- val lp' = LP.input "";
@ A, 0 => B, 0 => A
@ .
val lp' = - : lp
- RFA.checkLP (memb, rfa) lp';
transition from "B" to "A" has regular expression that doesn't
```

```
generate "0"

uncaught exception Error
```

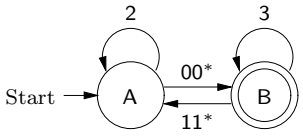
A string w is *accepted by* an RFA M iff there is a labeled path lp such that

- 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.

We have that, if w is accepted by M, then $\mathbf{alphabet}\, w \subseteq \mathbf{alphabet}\, M$. The *language accepted by* an RFA M ($L(M)$) is

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

Consider our example RFA M:



We have that 20 and 0000111103 are accepted by M, but that 23 and 122 are not accepted by M.

We define a function **combineTrans** that takes in a pair $(simp, U)$ such that

- $simp \in \mathbf{Reg} \to \mathbf{Reg}$ and
- U is a finite subset of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym} \text{ and } \alpha \in \mathbf{Reg} \,\}$,

and returns a finite subset V of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym} \text{ and } \alpha \in \mathbf{Reg} \,\}$ with the property that, for all $p, q \in \mathbf{Sym}$, there is at most one β such that $p, \beta \to q \in V$. Given such a pair $(simp, U)$, **combineTrans** returns the set of all transitions $p, \alpha \to q$ such that $\{\, \beta \mid p, \beta \to q \in U \,\}$ is nonempty, and $\alpha = simp(\beta_1 + \cdots + \beta_n)$, where $\beta_1, \ldots, \beta_n$ are all of the elements of this set, listed in increasing order and without repetition.

Now, we define a function/algorithm

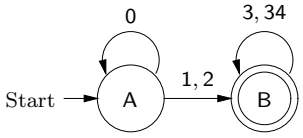
$$\mathbf{faToRFA} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{FA} \to \mathbf{RFA}.$$

faToRFA takes in $simp \in \mathbf{Reg} \to \mathbf{Reg}$, and returns a function that takes in $M \in \mathbf{FA}$, and returns the RFA N such that:

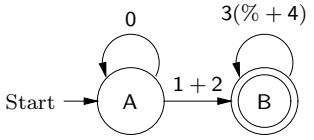
- $Q_N = Q_M$;

- $s_N = s_M$;
- $A_N = A_M$; and
- $T_N = \mathbf{combineTrans}(simp, \{\, p, \mathbf{strToReg}\, x \to q \mid p, x \to q \in T_M \,\})$.

For example, if the FA M is



and *simp* is **locallySimplify obviousSubset**, then **faToRFA** *simp* M is the RFA



Proposition 3.12.8
Suppose $simp \in \mathbf{Reg} \to \mathbf{Reg}$ *and* $M \in \mathbf{FA}$. *If, for all* $\alpha \in \mathbf{Reg}$, $L(simp\,\alpha) = L(\alpha)$ *and* $\mathbf{alphabet}(simp\,\alpha) \subseteq \mathbf{alphabet}\,\alpha$, *then*

(1) $L(\mathbf{faToRFA}\ simp\ M) = L(M)$, *and*

(2) $\mathbf{alphabet}(\mathbf{faToRFA}\ simp\ M) = \mathbf{alphabet}\,M$.

The `RFA` module has a function

```
val fromFA : (reg -> reg) -> fa -> rfa
```

that corresponds to **faToRFA**. Here is how our conversion example can be carried out in Forlan:

```
- val simp = #2 o Reg.locallySimplify(NONE, Reg.obviousSubset);
val simp = fn : reg -> reg
- val fa = FA.input "";
@ {states} A, B {start state} A {accepting states} B
@ {transitions}
@ A, 0 -> A; A, 1 -> B; A, 2 -> B;
@ B, 3 -> B; B, 34 -> B
@ .
val fa = - : fa
- val rfa = RFA.fromFA simp fa;
val rfa = - : rfa
- RFA.output("", rfa);
{states} A, B {start state} A {accepting states} B
{transitions} A, 0 -> A; A, 1 + 2 -> B; B, 3(% + 4) -> B
val it = () : unit
```

We say that an RFA M is *standard* iff

- M's start state is not an accepting state, and there are no transitions *into* M's start state (even from s_M to itself); and
- M has a single accepting state, and there are no transitions *from* that state (even from the accepting state to itself).

Proposition 3.12.9
Suppose M is a standard RFA with only two states, and that q is M's accepting state.

(1) For all $\alpha \in$ **Reg**, *if $s_M, \alpha \to q$, then $L(M) = L(\alpha)$.*

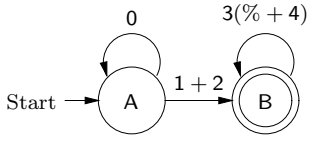
(2) If there is no $\alpha \in$ **Reg** *such that $s_M, \alpha \to q$, then $L(M) = \emptyset$.*

We define a function **standardize** $\in$ **RFA** $\to$ **RFA** that standardizes an RFA, as follows. Given an argument M, it returns the RFA N such that:

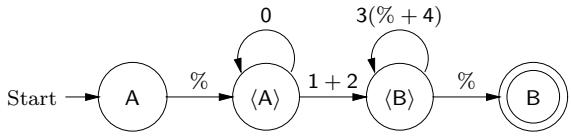
- $Q_N = \{ \langle q \rangle \mid q \in Q_M \} \cup \{\mathsf{A}, \mathsf{B}\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\mathsf{B}\}$; and
- T_N

$$\begin{aligned} &= \{\mathsf{A}, \% \to \langle s_M \rangle\} \\ &\cup \{ \langle q \rangle, \% \to \mathsf{B} \mid q \in A_M \} \\ &\cup \{ \langle q \rangle, \alpha \to \langle r \rangle \mid q, \alpha \to r \in T_M \}. \end{aligned}$$

For example, if M is the RFA



then **standardize** M is the RFA



Proposition 3.12.10
Suppose M is an RFA. Then:

- **standardize** M *is standard;*

- $L(\textbf{standardize}\, M) = L(M)$; *and*
- $\textbf{alphabet}(\textbf{standardize}\, M) = \textbf{alphabet}\, M$.

The `RFA` module has functions

```
val standard    : rfa -> bool
val standardize : rfa -> rfa
```

The function `standard` tests whether an RFA is standard, and the function `standardize` corresponds to **standardize**.

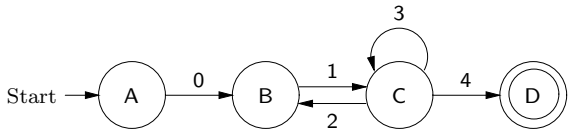
Here is how the above example can be carried out in Forlan:

```
- RFA.standard rfa;
val it = false : bool
- val rfa' = RFA.standardize rfa;
val rfa' = - : rfa
- RFA.output("", rfa');
{states} A, B, <A>, <B> {start state} A {accepting states} B
{transitions}
A, % -> <A>; <A>, 0 -> <A>; <A>, 1 + 2 -> <B>; <B>, % -> B;
<B>, 3(% + 4) -> <B>
val it = () : unit
- RFA.standard rfa';
val it = true : bool
```

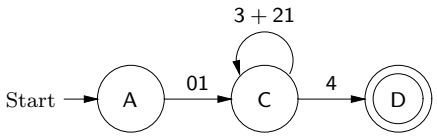
Next, we define a function **eliminateState** that takes in a function $simp \in \textbf{Reg} \to \textbf{Reg}$, and returns a function that takes in a pair (M, q), where M is an RFA and $q \in Q_M - (\{s_M\} \cup A_M)$, and returns an RFA. When called with such a *simp* and (M, q), **eliminateState** returns the RFA N such that:

- $Q_N = Q_M - \{q\}$;
- $s_N = s_M$;
- $A_N = A_M$; and
- $T_N = \textbf{combineTrans}(simp, U \cup V)$, where
  - $U = \{\, p, \alpha \to r \in T_M \mid p \neq q \text{ and } r \neq q \,\}$,
  - $V = \{\, p, simp(\alpha\beta^*\gamma) \to r \mid p \neq q, r \neq q, p, \alpha \to q \in T_M \text{ and } q, \gamma \to r \in T_M \,\}$, and
  - β is the unique $\alpha \in \textbf{Reg}$ such that $q, \alpha \to q \in T_M$, if such an α exists, and is $\%$, otherwise.

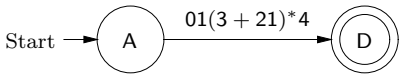
Suppose *simp* is **locallySimplify obviousSubset** and M is the FA



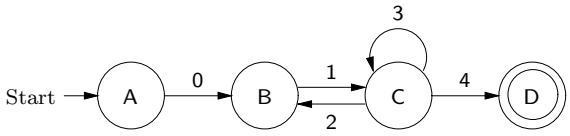
Then **eliminateState** *simp* (M, B) is



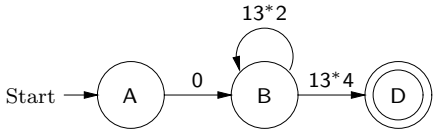
And, we can eliminate C from this RFA, yielding



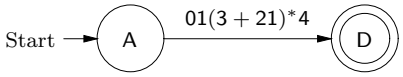
Alternatively, we could eliminate C from



yielding



And could then eliminate B from this RFA, yielding



$(simp(\mathsf{0}(\mathsf{13}^*\mathsf{2})^*(\mathsf{13}^*\mathsf{4})) = \mathsf{01}(\mathsf{3}+\mathsf{21})^*\mathsf{4}.)$

If we had an efficient regular expression simplifier that produced optimal results, then the order in which we eliminated states would be irrelevant. But using our existing simplifiers, it turns out that eliminating states in some orders produces much better results than doing so in other orders. Instead of eliminating first C and then B, we could have renamed M's states using the bijection

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

and then have eliminted states in ascending order, according to our usual ordering on symbols: first B and then C. This is the approach we'll use when looking for alternative answers.

Proposition 3.12.11
Suppose $simp \in \mathbf{Reg} \to \mathbf{Reg}$*,* M *is an RFA and* $q \in Q_M - (\{s_M\} \cup A_M)$*. Then:*

(1) **eliminateState** *simp* (M, q) *has one less state than* M*.*

(2) If M is standard, then **eliminateState** *simp* (M, q) *is standard.*

(3) If, for all $\alpha \in \mathbf{Reg}$, $L(simp\,\alpha) = L(\alpha)$, *then*

$$L(\mathbf{eliminateState}\, simp\, (M, q)) = L(M).$$

(4) If, for all $\alpha \in \mathbf{Reg}$, $\mathbf{alphabet}(simp\,\alpha) \subseteq \mathbf{alphabet}\,\alpha$, *then*

$$\mathbf{alphabet}(\mathbf{eliminateState}\, simp\, (M, q)) \subseteq \mathbf{alphabet}\, M.$$

The `RFA` module has a function

```
val eliminateState : (reg -> reg) -> rfa * sym -> rfa
```

that corresponds to **eliminateState**. Here is how our state-elimination examples can be carried out in Forlan:

```
- val rfa = RFA.input "";
@ {states} A, B, C, D {start state} A {accepting states} D
@ {transitions}
@ A, 0 -> B; B, 1 -> C; C, 2 -> B; C, 3 -> C; C, 4 -> D
@ .
val rfa = - : rfa
- val eliminateState = RFA.eliminateState simp;
val eliminateState = fn : rfa * sym -> rfa
- val rfa' = eliminateState(rfa, Sym.fromString "B");
val rfa' = - : rfa
- RFA.output("", rfa');
{states} A, C, D {start state} A {accepting states} D
{transitions} A, 01 -> C; C, 4 -> D; C, 3 + 21 -> C
val it = () : unit
- val rfa'' = eliminateState(rfa', Sym.fromString "C");
val rfa'' = - : rfa
- RFA.output("", rfa'');
{states} A, D {start state} A {accepting states} D
{transitions} A, 01(3 + 21)*4 -> D
val it = () : unit
- val rfa''' = eliminateState(rfa, Sym.fromString "C");
val rfa''' = - : rfa
- RFA.output("", rfa''');
{states} A, B, D {start state} A {accepting states} D
{transitions} A, 0 -> B; B, 13*2 -> B; B, 13*4 -> D
val it = () : unit
- val rfa'''' = eliminateState(rfa''', Sym.fromString "B");
val rfa'''' = - : rfa
- RFA.output("", rfa'''');
{states} A, D {start state} A {accepting states} D
{transitions} A, 01(3 + 21)*4 -> D
val it = () : unit
```

And `eliminateState` stops us from eliminating a start state or an accepting state:

```
- eliminateState(rfa, Sym.fromString "A");
cannot eliminate start state: "A"

uncaught exception Error
- eliminateState(rfa, Sym.fromString "D");
cannot eliminate accepting state: "D"

uncaught exception Error
```

Now, we use **eliminateState** to define a function/algorithm

$$\mathbf{rfaToReg} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{RFA} \to \mathbf{Reg}.$$

It takes elements $simp \in \mathbf{Reg} \to \mathbf{Reg}$ and $M \in \mathbf{RFA}$, and returns

$$f(\mathbf{standardize}\, M),$$

where f is the function from standard RFAs to regular expressions that is defined by well-founded recursion on the number of states of its input, M, as follows:

- If M has only two states, then f returns the label of the transition from s_M to M's accepting state, if such a transition exists, and returns \$, otherwise.
- Otherwise, f calls itself recursively on $\mathbf{eliminateState}\, simp\, (M, q)$, where q is the least element (in the standard ordering on symbols) of $Q_M - (\{s_M\} \cup A_M)$.

Proposition 3.12.12
Suppose M is an RFA and $simp \in \mathbf{Reg} \to \mathbf{Reg}$ has the property that, for all $\alpha \in \mathbf{Reg}$, $L(simp\, \alpha) = L(\alpha)$ and $\mathbf{alphabet}(simp\, \alpha) \subseteq \mathbf{alphabet}\, \alpha$. Then:

(1) $L(\mathbf{rfaToReg}\, simp\, M) = L(M)$*; and*

(2) $\mathbf{alphabet}(\mathbf{rfaToReg}\, simp\, M) \subseteq \mathbf{alphabet}\, M$.

Finally, we define our RFA to regular expression conversion algorithm/function:

$$\mathbf{faToReg} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{FA} \to \mathbf{Reg}.$$

faToReg takes in $simp \in \mathbf{Reg} \to \mathbf{Reg}$, and returns

$$\mathbf{rfaToReg}\, simp \circ \mathbf{faToRFA}\, simp.$$

Proposition 3.12.13
Suppose M is an FA and $simp \in \mathbf{Reg} \to \mathbf{Reg}$ has the property that, for all $\alpha \in \mathbf{Reg}$, $L(simp\, \alpha) = L(\alpha)$ and $\mathbf{alphabet}(simp\, \alpha) \subseteq \mathbf{alphabet}\, \alpha$. Then:

(1) $L($**faToReg** *simp* $M) = L(M)$*; and*

(2) **alphabet**(**faToReg** *simp* $M) \subseteq$ **alphabet** M.

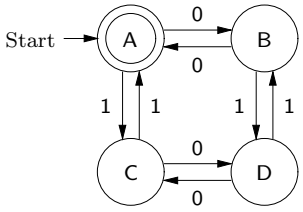
The Forlan module `RFA` includes functions

```
val toReg             : (reg -> reg) -> rfa -> reg
val faToReg           : (reg -> reg) -> fa -> reg
val faToRegPerms      : int option * (reg -> reg) -> fa -> reg
val faToRegPermsTrace : int option * (reg -> reg) -> fa -> reg
```

The function `toReg` corresponds to **rfaToReg**. The function `faToReg` is the implementation of **faToReg**. If *simp* is a simplification function and M is an FA, then `faToRegPerms (NONE,` *simp*`)` M applies **faToReg** to all of the FAs N that can be formed by renaming M's states using bijections (i.e., permutations) from Q_M to Q_M, and returns the simplest answer found (ties in complexity are broken by selecting the smallest regular expression in our total ordering on regular expressions.) `faToRegPerms (SOME` n*,* *simp*`)` M, for $n \geq 1$, works similarly, except that only n ways of renaming M's state are considered. And `faToRegPermsTrace` is like `faToRegPerms` except that it explains what it's doing. The functions `faToReg`, `faToRegPerms` and `faToRegTrace` are also available in the top-level environment with those names:

```
val faToReg           : (reg -> reg) -> fa -> reg
val faToRegPerms      : int option * (reg -> reg) -> fa -> reg
val faToRegPermsTrace : int option * (reg -> reg) -> fa -> reg
```

Suppose `fa` is the FA



which accepts $\{ w \in \{0,1\}^* \mid w$ has an even number of 0 and 1's$\}$. converting `fa` into a regular expression using `faToReg` and **weaklySimplify** yields a fairly complicated answer:

```
- val reg = faToReg Reg.weaklySimplify fa;
val reg = - : reg
- Reg.output("", reg);
% + 00(00)* + (1 + 00(00)*1)(11 + 100(00)*1)*(1 + 100(00)*) +
(0(00)*1 + (1 + 00(00)*1)(11 + 100(00)*1)*(0 + 10(00)*1))
(1(00)*1 + (0 + 10(00)*1)(11 + 100(00)*1)*(0 + 10(00)*1))*
(10(00)* + (0 + 10(00)*1)(11 + 100(00)*1)*(1 + 100(00)*))
val it = () : unit
```

But by using `faToRegPerms`, we can do much better:

```
- val reg' = faToRegPerms (NONE, Reg.weaklySimplify) fa;
val reg' = - : reg
- Reg.output("", reg');
(00 + 11 + (01 + 10)(00 + 11)*(01 + 10))*
val it = () : unit
```

By using `faToRegPermsTrace`, we can learn that this answer was found using the renaming

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

of M's states. That is, it was found by making M into a standard RFA, with new start and accepting states, and then eliminating the states corresponding to B, C, D and A, in that order.

3.12.3 Characterization of Regular Languages

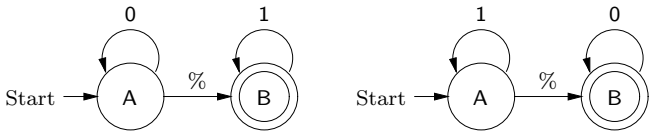
Since we have algorithms for converting back and forth between regular expressions and finite automata, as well as algorithms for converting FAs to RFAs, RFAs to regular expressions, FAs to EFAs, EFAs to NFAs, and NFAs to DFAs, we have the following theorem:

Theorem 3.12.14
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 regular expression finite automaton;*
- *L is accepted by a finite automaton;*
- *L is accepted by an EFA;*
- *L is accepted by an NFA; and*
- *L is accepted by a DFA.*

3.12.4 More Closure Properties/Algorithms

Consider the EFAs M_1 and M_2:



(M_1) (M_2)

How can we construct an EFA N such that $L(N) = L(M_1) \cap L(M_2)$? The idea is to make the states of N represent pairs of the form (q, r), where $q \in Q_{M_1}$ and $r \in Q_{M_2}$.

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 $\mathbf{nextSym}_{M_1,M_2}((q, r), a) =$

$$\{\, (q', r') \mid q, a \to q' \in T_{M_1} \text{ and } r, a \to r' \in T_{M_2} \,\}.$$

We often abbreviate $\mathbf{nextSym}_{M_1,M_2}$ to **nextSym**. If M_1 and M_2 are our example EFAs, then $\mathbf{nextSym}((\mathsf{A}, \mathsf{A}), \mathsf{0}) = \emptyset$ and $\mathbf{nextSym}((\mathsf{A}, \mathsf{B}), \mathsf{0}) = \{(\mathsf{A}, \mathsf{B})\}$. 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 $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid q, \% \to q' \in T_{M_1} \,\} \cup \{\, (q, r') \mid r, \% \to r' \in T_{M_2} \,\}.$$

We often abbreviate $\mathbf{nextEmp}_{M_1,M_2}$ to **nextEmp**. If M_1 and M_2 are our example EFAs, then $\mathbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$, $\mathbf{nextEmp}(\mathsf{A}, \mathsf{B}) = \{(\mathsf{B}, \mathsf{B})\}$, $\mathbf{nextEmp}(\mathsf{B}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{B})\}$ and $\mathbf{nextEmp}(\mathsf{B}, \mathsf{B}) = \emptyset$.

Now, we define a function/algorithm $\mathbf{inter} \in \mathbf{EFA} \times \mathbf{EFA} \to \mathbf{EFA}$ such that $L(\mathbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$, for all $M_1, M_2 \in \mathbf{EFA}$. Given EFAs M_1 and M_2, $\mathbf{inter}(M_1, M_2)$, the *intersection of M_1 and M_2*, is the EFA N that is constructed as follows. First, we let $\Sigma = \mathbf{alphabet}\, M_1 \cap \mathbf{alphabet}\, M_2$. 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 $\mathbf{nextSym}((q, r), a) \subseteq X$; and
- for all $q \in Q_{M_1}$ and $r \in Q_{M_2}$, if $(q, r) \in X$, then $\mathbf{nextEmp}(q, r) \subseteq X$.

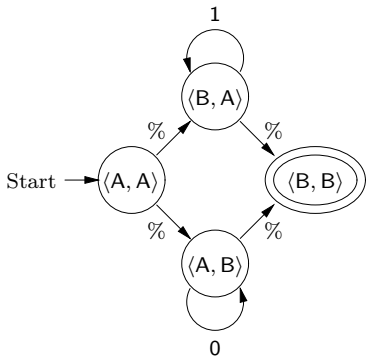
Then, the EFA N is defined by:

- $Q_N = \{\, \langle q, r \rangle \mid (q, r) \in X \,\}$;
- $s_N = \langle s_{M_1}, s_{M_2} \rangle$;
- $A_N = \{\, \langle q, r \rangle \mid (q, r) \in X \text{ and } q \in A_{M_1} \text{ and } r \in A_{M_2} \,\}$; and

- $T_N =$

$$\begin{aligned}&\{\, \langle q, r\rangle, a \to \langle q', r'\rangle \mid (q, r) \in X \text{ and } a \in \Sigma \text{ and} \\ &\qquad (q', r') \in \mathbf{nextSym}((q, r), a)\,\} \\ \cup\, &\{\, \langle q, r\rangle, \% \to \langle q', r'\rangle \mid (q, r) \in X \text{ and} \\ &\qquad (q', r') \in \mathbf{nextEmp}(q, r)\,\}.\end{aligned}$$

Suppose M_1 and M_2 are our example EFAs. Then $\mathbf{inter}(M_1, M_2)$ is



Theorem 3.12.15
For all $M_1, M_2 \in$ **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 3.12.16
For all $M_1, M_2 \in$ **NFA**, $\mathbf{inter}(M_1, M_2) \in$ **NFA**.

Proposition 3.12.17
For all $M_1, M_2 \in$ **DFA***:*

(1) $\mathbf{inter}(M_1, M_2) \in$ **DFA**.

(2) $\mathbf{alphabet}(\mathbf{inter}(M_1, M_2)) = \mathbf{alphabet}\, M_1 \cap \mathbf{alphabet}\, M_2$.

Next, we define a function $\mathbf{complement} \in \mathbf{DFA} \times \mathbf{Alp} \to \mathbf{DFA}$ such that, for all $M \in$ **DFA** and $\Sigma \in$ **Alp**,

$$L(\mathbf{complement}(M, \Sigma)) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M).$$

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, $\mathbf{complement}(M, \Sigma)$, the *complement of* M *with reference to* Σ, 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; and
- $\textbf{alphabet}\, M' = \textbf{alphabet}(L(M)) \cup \Sigma$.

Then, we define N by:

- $Q_N = Q_{M'}$;
- $s_N = s_{M'}$;
- $A_N = Q_{M'} - A_{M'}$; and
- $T_N = T_{M'}$.

Then, for all $w \in (\textbf{alphabet}\, M')^* = (\textbf{alphabet}\, N)^* = (\textbf{alphabet}(L(M)) \cup \Sigma)^*$,

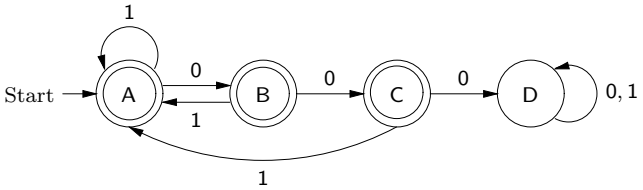
$$\begin{array}{rll} w \in L(N) & \text{iff} & \delta_N(s_N, w) \in A_N \\ & \text{iff} & \delta_N(s_N, w) \in Q_{M'} - A_{M'} \\ & \text{iff} & \delta_{M'}(s_{M'}, w) \notin A_{M'} \\ & \text{iff} & w \notin L(M') \\ & \text{iff} & w \notin L(M). \end{array}$$

Hence:

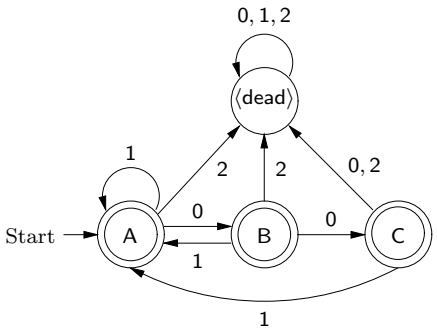
Theorem 3.12.18
For all $M \in \textbf{DFA}$ *and* $\Sigma \in \textbf{Alp}$*:*

- $L(\textbf{complement}(M, \Sigma)) = (\textbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$*; and*
- $\textbf{alphabet}(\textbf{complement}(M, \Sigma)) = \textbf{alphabet}(L(M)) \cup \Sigma$.

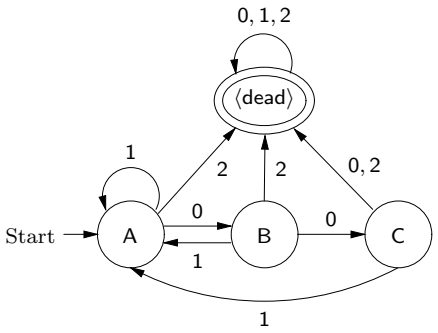
For example, suppose the DFA M is



Then $\textbf{determSimplify}(M, \{2\})$ is the DFA



Thus **complement**$(M, \{2\})$ is



Let $X = \{\, w \in \{0,1\}^* \mid 000 \text{ is not a substring of } w \,\}$. Then $L($**complement**$(M, \{2\}))$ is

$$\begin{aligned}
&(\textbf{alphabet}(L(M)) \cup \{2\})^* - L(M) \\
&= (\{0,1\} \cup \{2\})^* - X \\
&= \{\, w \in \{0,1,2\}^* \mid w \notin X \,\} \\
&= \{\, w \in \{0,1,2\}^* \mid 2 \in \textbf{alphabet}\, w \text{ or } 000 \text{ is a substring of } w \,\}.
\end{aligned}$$

We define a function/algorithm **minus** $\in$ **DFA** $\times$ **DFA** $\to$ **DFA** by: **minus**(M_1, M_2), the *difference of* M_1 *and* M_2, is

$$\textbf{inter}(M_1, \textbf{complement}(M_2, \textbf{alphabet}\, M_1)).$$

Theorem 3.12.19
For all $M_1, M_2 \in$ **DFA***:*

(1) $L($**minus**$(M_1, M_2)) = L(M_1) - L(M_2)$*; and*

(2) **alphabet**$($**minus**$(M_1, M_2)) =$ **alphabet** M_1.

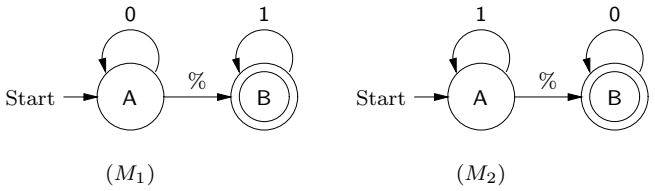
Proof. Suppose $w \in$ **Str**. Then:

$$\begin{aligned}
& w \in L(\textbf{minus}(M_1, M_2)) \\
\text{iff}\quad & w \in L(\textbf{inter}(M_1, \textbf{complement}(M_2, \textbf{alphabet}\, M_1))) \\
\text{iff}\quad & w \in L(M_1) \text{ and } w \in L(\textbf{complement}(M_2, \textbf{alphabet}\, M_1)) \\
\text{iff}\quad & w \in L(M_1) \text{ and } w \in (\textbf{alphabet}(L(M_2)) \cup \textbf{alphabet}\, M_1)^* \text{ and} \\
& w \notin L(M_2) \\
\text{iff}\quad & w \in L(M_1) \text{ and } w \notin L(M_2) \\
\text{iff}\quad & w \in L(M_1) - L(M_2).
\end{aligned}$$

□

To see why the second argument to **complement** is **alphabet** M_1, in the definition of **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 $w \in (\textbf{alphabet}\, M_1)^*$, so that $w \in (\textbf{alphabet}(L(M_2)) \cup \textbf{alphabet}(M_1))^*$.

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 DFAs N_1 and N_2 by:

$$N_1 = \textbf{nfaToDFA}(\textbf{efaToNFA}\, M_1), \text{ and}$$
$$N_2 = \textbf{nfaToDFA}(\textbf{efaToNFA}\, M_2).$$

Thus we have that

$$\begin{aligned}
L(N_1) &= L(\textbf{nfaToDFA}(\textbf{efaToNFA}(M_1))) \\
&= L(\textbf{efaToNFA}(M_1)) && \text{(Theorem 3.11.12)} \\
&= L(M_1) && \text{(Theorem 3.10.4)} \\
L(N_2) &= L(\textbf{nfaToDFA}(\textbf{efaToNFA}(M_2))) \\
&= L(\textbf{efaToNFA}(M_2)) && \text{(Theorem 3.11.12)} \\
&= L(M_2) && \text{(Theorem 3.10.4).}
\end{aligned}$$

Let the DFA $N = \textbf{minus}(N_1, N_2)$. Then

$$\begin{aligned}
L(N) &= L(\textbf{minus}(N_1, N_2)) \\
&= L(N_1) - L(N_2) && \text{(Theorem 3.12.19)} \\
&= L(M_1) - L(M_2) \\
&= \{0\}\{0\}^*\{1\}\{1\}^*.
\end{aligned}$$

Next, we consider the reversal of languages, regular expressions, finite automata and empty-string finite automata. The *reversal of* a language L ($L^R \in \textbf{Lan}$) 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, $\{011, 1011\}^R = \{110, 1101\}$.

Define $\mathbf{rev} \in \mathbf{Reg} \to \mathbf{Reg}$ by recursion:

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

We say that $\mathbf{rev}\,\alpha$ is *the reversal of* α. For example $\mathbf{rev}(\mathsf{01} + (\mathsf{10})^*) = \mathsf{10} + (\mathsf{01})^*$.

Theorem 3.12.20
For all $\alpha \in \mathbf{Reg}$*:*

- $L(\mathbf{rev}\,\alpha) = L(\alpha)^R$*; and*
- $\mathbf{alphabet}(\mathbf{rev}\,\alpha) = \mathbf{alphabet}\,\alpha$.

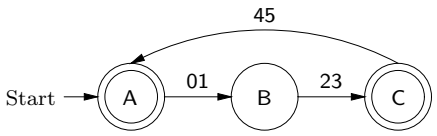
Proof. By induction on α. □

We can also define a reversal operation on FAs and EFAs. The idea is to reverse all of the transitions, add a new start state, with %-transitions to all of the original accepting states, and make the original start state be the unique accepting state. Formally, we define a function $\mathbf{rev} \in \mathbf{FA} \to \mathbf{FA}$ as follows. Given an FA M, $\mathbf{rev}\,M$, *the reversal of* M, is the FA N such that

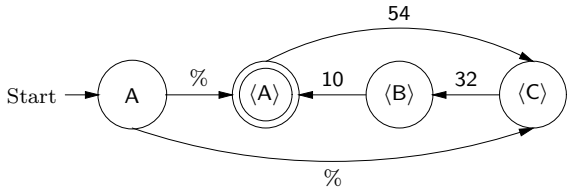
- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\langle s_M \rangle\}$; and
- $T_N = \{\, (\langle r \rangle, x^R, \langle q \rangle) \mid (q, x, r) \in T_M \,\} \cup \{\, (\mathsf{A}, \%, \langle q \rangle) \mid q \in A_M \,\}$.

It is easy to see that, for all $M \in \mathbf{EFA}$, $\mathbf{rev}\,M \in \mathbf{EFA}$.

For example, if M is the FA



then $\mathbf{rev}\,M$ is



We have that 012345 and 0123450123 are in $L(M)$, and 543210 and 3210543210 are in $L(\mathbf{rev}\, M) = L(M)^R$.

Theorem 3.12.21
*For all $M \in$ **FA**:*

- $L(\mathbf{rev}\, M) = L(M)^R$; *and*
- $\mathbf{alphabet}(\mathbf{rev}\, M) = \mathbf{alphabet}\, M$.

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., $\{012, 3\}^P = \{\%, 0, 01, 012, 3\}$.
- 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., $\{012, 3\}^S = \{\%, 2, 12, 012, 3\}$.
- 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., $\{012, 3\}^{SS} = \{\%, 0, 1, 2, 01, 12, 012, 3\}$.

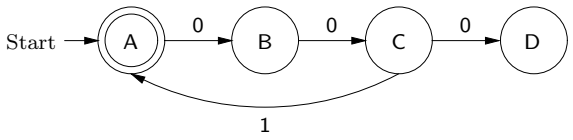
The following proposition shows that we can express suffix- and substring-closure in terms of prefix-closure and language reversal.

Proposition 3.12.22
For all languages L:

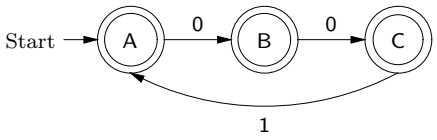
- $L^S = ((L^R)^P)^R$; *and*
- $L^{SS} = (L^P)^S$.

Now, we define a function $\mathbf{prefix} \in \mathbf{EFA} \to \mathbf{EFA}$ such that $L(\mathbf{prefix}\, M) = L(M)^P$, for all $M \in \mathbf{EFA}$. Given an EFA M, $\mathbf{prefix}\, M$, *the prefix-closure of* M, is the EFA N that is constructed as follows. First, we simplify M, producing an EFA 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 EFA



so that $L(M) = \{001\}^*$. Then **prefix** M is the EFA



which accepts $\{001\}^*\{\%, 0, 00\}$.

Theorem 3.12.23
For all $M \in$ **EFA***:*

- $L(\textbf{prefix}\, M) = L(M)^P$*; and*
- $\textbf{alphabet}(\textbf{prefix}\, M) = \textbf{alphabet}(L(M))$.

Proposition 3.12.24
For all $M \in$ **NFA**, $\textbf{prefix}\, M \in$ **NFA**.

Now we can define suffix-closure and substring-closure operations on EFAs as follows. The functions $\textbf{suffix}, \textbf{substring} \in \textbf{EFA} \to \textbf{EFA}$ are defined by:

$$\textbf{suffix}\, M = \textbf{rev}(\textbf{prefix}(\textbf{rev}\, M))), \text{ and}$$
$$\textbf{substring}\, M = \textbf{suffix}(\textbf{prefix}\, M)).$$

Theorem 3.12.25
For all $M \in$ **EFA***:*

- $L(\textbf{suffix}\, M) = L(M)^S$*; and*
- $L(\textbf{substring}\, M) = L(M)^{SS}$.

Next, we consider the renaming of regular expressions and finite automata using bijections on symbols. If x is a string and f is a bijection from a set of symbols that is a superset of **alphabet** x (maybe **alphabet** x itself, i.e., the symbols appearing in x), then the *renaming of* x *using* f ($x^f \in$ **Str**) is the result of applying f to each symbol of x. For example, if $f = \{(0,1),(1,2),(2,3)\}$, then $\%^f = \%$ and $(01102)^f = 12213$.

If L is a language, and f is a bijection from a set of symbols that is a superset of **alphabet** L (maybe **alphabet** L itself) to some set of symbols, then the *renaming of* L *using* f ($L^f \in$ **Lan**) 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 $\}$. Then, 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, $\mathbf{renameAlphabet}(012 + 12, \{(0,1),(1,2),(2,3)\}) = 123 + 23$.

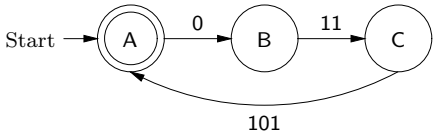
Theorem 3.12.26
For all $\alpha \in \mathbf{Reg}$ and bijections f from sets of symbols that are supersets of $\mathbf{alphabet}\,\alpha$ to sets of symbols:

- $L(\mathbf{renameAlphabet}(\alpha, f)) = L(\alpha)^f$*; and*
- $\mathbf{alphabet}(\mathbf{renameAlphabet}(\alpha, f)) = \{ f\,a \mid a \in \mathbf{alphabet}\,\alpha \}$.

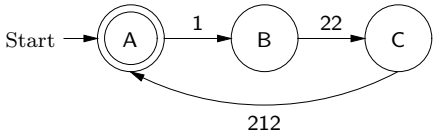
For example, if $f = \{(0,1),(1,2),(2,3)\}$, then

$$\begin{aligned} L(\mathbf{renameAlphabet}(012 + 12, f)) &= L(012 + 12)^f = \{012, 12\}^f \\ &= \{123, 23\}. \end{aligned}$$

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 $\}$. Then, the function $\mathbf{renameAlphabet} \in X \to \mathbf{FA}$ 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 $\mathbf{renameAlphabet}(M, f)$ is the FA



Theorem 3.12.27
For all $M \in \mathbf{FA}$ and bijections f from sets of symbols that are supersets of $\mathbf{alphabet}\,M$ 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 $\mathbf{renameAlphabet}(M, f)$ is an EFA;*

- *if M is an NFA, then* **renameAlphabet**(M, f) *is an NFA; and*
- *if M is a DFA, then* **renameAlphabet**(M, f) *is a DFA.*

Theorem 3.12.28
Suppose $L, L_1, L_2 \in$ **RegLan**. *Then:*

(1) $L_1 \cup L_2 \in$ **RegLan**;

(2) $L_1 L_2 \in$ **RegLan**;

(3) $L^* \in$ **RegLan**;

(4) $L_1 \cap L_2 \in$ **RegLan**;

(5) $L_1 - L_2 \in$ **RegLan**;

(6) $L^R \in$ **RegLan**;

(7) $L^P \in$ **RegLan**;

(8) $L^S \in$ **RegLan**;

(9) $L^{SS} \in$ **RegLan**; *and*

(10) $L^f \in$ **RegLan**, *where f is a bijection from a set of symbols that is a superset of* **alphabet** L *to some set of symbols.*

Proof. Parts (1)–(10) hold because of the operations **union**, **concat** and **closure** on FAs, the operation **inter** on EFAs, the operation **minus** on DFAs, the operation **rev** on regular expressions, the operations **prefix**, **suffix** and **substring** on EFAs, the operation **renameAlphabet** on regular expressions, and Theorem 3.12.14. □

The Forlan module `EFA` defines the function/algorithm

```
val inter : efa * efa -> efa
```

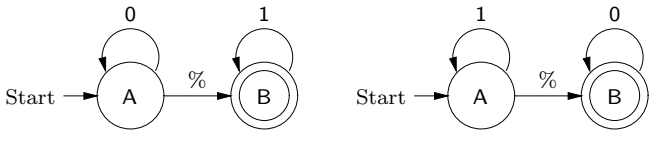
which corresponds to **inter**. It is also inherited by the modules `DFA` and `NFA`.

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:

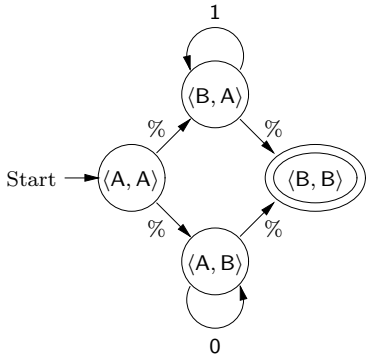


(M_1) (M_2)

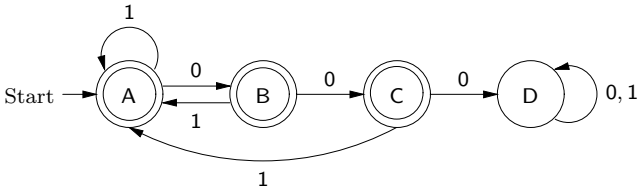
Then, we can construct **inter**(M_1, M_2) as follows:

```
- 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

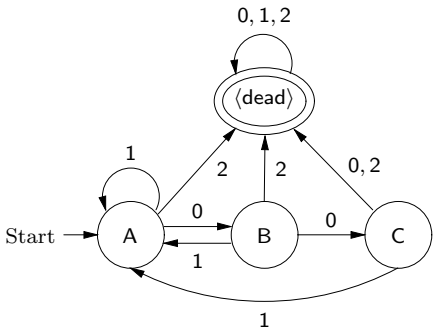


Then we can construct the DFA **complement**$(M, \{2\})$ as follows:

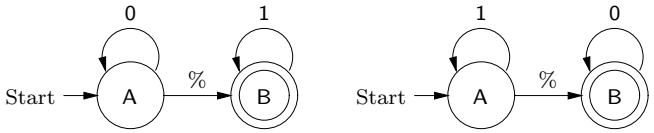
```
- val dfa' = DFA.complement(dfa, SymSet.input "");
@ 2
@ .
val dfa' = - : dfa
- DFA.output("", dfa');
{states} A, B, C, <dead> {start state} A {accepting states} <dead>
{transitions}
```

```
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
```

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 reversal of FAs and EFAs in Forlan. The Forlan module `FA` defines the function

```
val rev : fa -> fa
```

which corresponds to **rev**. It is also inherited by the module `EFA`. Here is an example of how this function can be used:

```
- val fa = FA.input "";
@ {states}
@ A, B, C
@ {start state}
@ A
@ {accepting states}
@ A, C
@ {transitions}
@ A, 01 -> B; B, 23 -> C; C, 45 -> A
@ .
val fa = - : fa
- val fa' = FA.rev fa;
val fa' = - : fa
- FA.output("", fa');
{states} A, <A>, <B>, <C> {start state} A {accepting states} <A>
{transitions}
A, % -> <A> | <C>; <A>, 54 -> <C>; <B>, 10 -> <A>; <C>, 32 -> <B>
val it = () : unit
```

Next, we see how we can carry out the prefix-closure of EFAs and NFAs in Forlan. The Forlan module `EFA` defines the function

```
val prefix : efa -> efa
```

which corresponds to **prefix**. It is also inherited by the module `NFA`. Here is an example of how one of these functions 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, 0 -> 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
```

3.12.5 Notes

The material in this section is mostly standard, but is worked-out in more detail than is common. We have given an intersection algorithm on EFAs, not just on DFAs. By giving a complementation algorithm on DFAs that allows us to partially control the alphabet of the resulting DFA, we are able to give an algorithm for computing the difference of DFAs that works even when the DFAs have different alphabets.

3.13 Equivalence-testing and Minimization of DFAs

In this section, we give algorithms for testing whether two DFAs are equivalent, and for minimizing the alphabet size and number of states of a DFA. We also see how these functions can be used in Forlan.

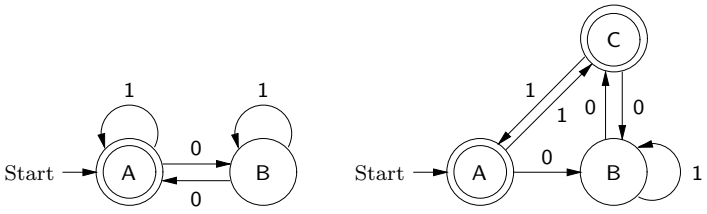
3.13.1 Testing the Equivalence of DFAs

Suppose M and N are DFAs. Our algorithm for checking whether they are equivalent proceeds as follows. First, it converts M and N into DFAs with identical alphabets. Let $\Sigma = \mathbf{alphabet}\, M \cup \mathbf{alphabet}\, N$, and define the DFAs M' and N' by:

$$M' = \mathbf{determSimplify}(M, \Sigma), \text{ and}$$
$$N' = \mathbf{determSimplify}(N, \Sigma).$$

Since $\mathbf{alphabet}(L(M)) \subseteq \mathbf{alphabet}\, M \subseteq \Sigma$, we have that $\mathbf{alphabet}\, M' = \mathbf{alphabet}(L(M)) \cup \Sigma = \Sigma$. Similarly, $\mathbf{alphabet}\, N' = \Sigma$. 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



then $\Sigma = \{0, 1\}$, $M' = M$ and $N' = N$.

Next, the algorithm generates the least subset X of $Q_{M'} \times Q_{N'}$ such that

- $(s_{M'}, s_{N'}) \in X$; and
- 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) $(\mathsf{B}, \mathsf{B}) \in X$; and
- 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 3.13.1
For all $w \in \Sigma^*$, $(\delta_{M'}(s_{M'}, w), \delta_{N'}(s_{N'}, w)) \in X$.

Proof. By left string induction on w. □

Lemma 3.13.2
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. □

Finally, the algorithm checks that, for all $(q, r) \in X$,

$$q \in A_{M'} \quad \text{iff} \quad r \in A_{N'}.$$

If this is true, it says that the machines are equivalent; otherwise it says they are not equivalent.

We can easily prove the correctness of our algorithm.

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 3.13.1 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 3.13.2, 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 $L(M) = L(N)$.

By annotating each element $(q, r) \in X$ with a string w such that $q = \delta_{M'}(s_{M'}, w)$ and $r = \delta_{N'}(s_{N'}, w)$, instead of just reporting that M' and N' are not equivalent, we can explain why they are not equivalent,

- 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.

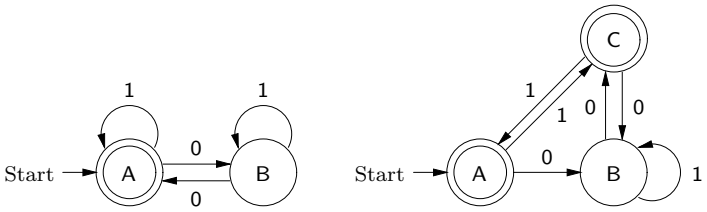
The Forlan module `DFA` defines the functions:

```
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, is neither a subset of the other), 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 the local and global simplification algorithms of Section 3.3.

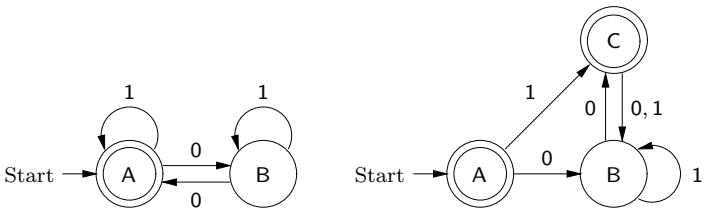
For example, suppose `dfa1` and `dfa2` of type `dfa` are bound to our example DFAs M and N, respectively:



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 textttdfa 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 generated by regular expressions `reg1` and `reg2` by:

- converting `reg1` and `reg2` to DFAs textttdfa1 and `dfa2`, and then
- running `DFA.relationship(dfa1, dfa2)` to find the relationship between those DFAs.

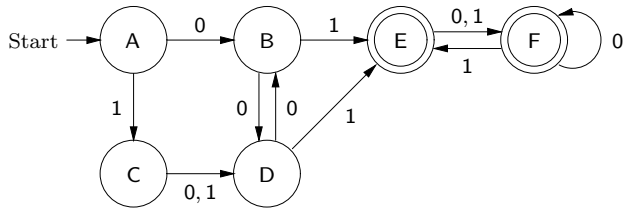
Of course, we can define an ML/Forlan function that carries out these actions:

```
- fun regToDFA reg =
=       nfaToDFA(efaToNFA(faToEFA(regToFA reg)));
val regToDFA = fn : reg -> dfa
- fun relationshipReg(reg1, reg2) =
=       DFA.relationship(regToDFA reg1, regToDFA reg2);
val relationshipReg = fn : reg * reg -> unit
```

3.13.2 Minimization of DFAs

Now, we consider an algorithm for minimizing the sizes of the alphabet and set of states of a DFA M. First, the algorithm minimizes the size of M's alphabet, and makes 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, the algorithm generates the least subset X of $Q_{M'} \times Q_{M'}$ such that:

(1) $A_{M'} \times (Q_{M'} - A_{M'}) \subseteq X$;

(2) $(Q_{M'} - A_{M'}) \times A_{M'} \subseteq X$; and

(3) for all $q, q', r, r' \in Q_{M'}$ and $a \in \mathbf{alphabet}\, M'$, 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 $(\mathsf{B}, \mathsf{1}, \mathsf{E})$, $(\mathsf{D}, \mathsf{1}, \mathsf{E})$, $(\mathsf{F}, \mathsf{1}, \mathsf{E})$ and $(\mathsf{C}, \mathsf{1}, \mathsf{D})$ are the $\mathsf{1}$-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 $\mathsf{0}$-transitions into E, nothing can be added to X using (E, D) and (D, E) and $\mathsf{0}$-transitions.

- Since $(\mathsf{F}, \mathsf{B}), (\mathsf{B}, \mathsf{F}) \in X$ and $(\mathsf{E}, \mathsf{0}, \mathsf{F})$, $(\mathsf{F}, \mathsf{0}, \mathsf{F})$, $(\mathsf{A}, \mathsf{0}, \mathsf{B})$, and $(\mathsf{D}, \mathsf{0}, \mathsf{B})$ are the $\mathsf{0}$-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 $\mathsf{1}$-transitions leading into B, no pairs can be added using (F, B) and (B, F) and $\mathsf{1}$-transitions.

- Since $(\mathsf{F}, \mathsf{C}), (\mathsf{C}, \mathsf{F}) \in X$ and $(\mathsf{E}, \mathsf{1}, \mathsf{F})$ and $(\mathsf{A}, \mathsf{1}, \mathsf{C})$ are the $\mathsf{1}$-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 $\mathsf{0}$-transitions leading into C, no pairs can be added using (F, C) and (C, F) and $\mathsf{0}$-transitions.

- Since $(\mathsf{F}, \mathsf{D}), (\mathsf{D}, \mathsf{F}) \in X$ and $(\mathsf{E}, \mathsf{0}, \mathsf{F})$, $(\mathsf{F}, \mathsf{0}, \mathsf{F})$, $(\mathsf{B}, \mathsf{0}, \mathsf{D})$ and $(\mathsf{C}, \mathsf{0}, \mathsf{D})$ are the $\mathsf{0}$-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 $(\mathsf{E}, \mathsf{1}, \mathsf{F})$ and $(\mathsf{C}, \mathsf{1}, \mathsf{D})$ are the $\mathsf{1}$-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 $\mathsf{1}$-transitions leading into B, and there are no $\mathsf{0}$-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 $(\mathsf{A},\mathsf{1},\mathsf{C})$ and $(\mathsf{C},\mathsf{1},\mathsf{D})$ are the $\mathsf{1}$-transitions leading into C and D, we add the pairs (A,C) and (C,A) to X. Since there are no $\mathsf{0}$-transitions leading into C, no pairs can be added to X using (C,D) and (D,C) and $\mathsf{0}$-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).

Back in the general case, we have the following lemmas.

Lemma 3.13.3
For all $(q,r) \in X$, there is a $w \in (\textbf{alphabet}\, M')^*$, such that exactly one of $\delta_{M'}(q,w)$ and $\delta_{M'}(r,w)$ is in $A_{M'}$.

Proof. By induction on X. □

Lemma 3.13.4
For all $w \in (\textbf{alphabet}\, M')^*$, for all $q,r \in Q_{M'}$, if exactly one of $\delta_{M'}(q,w)$ and $\delta_{M'}(r,w)$ is in $A_{M'}$, then $(q,r) \in X$.

Proof. By right string induction. □

Next, the algorithm lets the relation $Y = (Q_{M'} \times Q_{M'}) - X$. We read "$(q,r) \in Y$" as "$q$ and r can be merged". Back with our example, we have that Y is

$$\{(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{B}),(\mathsf{C},\mathsf{C}),(\mathsf{D},\mathsf{D}),(\mathsf{E},\mathsf{E}),(\mathsf{F},\mathsf{F})\}$$
$$\cup$$
$$\{(\mathsf{B},\mathsf{D}),(\mathsf{D},\mathsf{B}),(\mathsf{F},\mathsf{E}),(\mathsf{E},\mathsf{F})\}.$$

Lemma 3.13.5

(1) For all $q, r \in Q_{M'}$, $(q, r) \in Y$ iff, for all $w \in (\textbf{alphabet}\, M')^*$, $\delta_{M'}(q, w) \in A_{M'}$ iff $\delta_{M'}(r, w) \in A_{M'}$.

(2) For all $q, r \in Q_{M'}$, if $(q, r) \in Y$, then $q \in A_{M'}$ iff $r \in A_{M'}$.

(3) For all $q, r \in Q_{M'}$ and $a \in \textbf{alphabet}\, M'$, if $(q, r) \in Y$, then $(\delta_{M'}(q, a), \delta_{M'}(r, a)) \in Y$.

Proof.

(1) Follows using Lemmas 3.13.3 and 3.13.4.

(2) Follows by Part (1), when $w = \%$.

(3) Follows by Part (1).

□

The following lemma says that Y is an *equivalence relation on* $Q_{M'}$.

Lemma 3.13.6
Y is reflexive on $Q_{M'}$, symmetric and transitive.

Proof. Follows from Lemma 3.13.5(1). □

In order to define the DFA N that is the result of our minimization algorithm, we need a bit more notation. As in Section 3.11, 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 $\{\, p \in Q_{M'} \mid (p, q) \in Y \,\}$, which is called the *equivalence class* of q. Using Lemma 3.13.6, it is easy to show that, $q \in [q]$, for all $q \in Q_{M'}$, and $[q] = [r]$ iff $(q, r) \in Y$, for all $q, r \in Q_{M'}$.

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.

The algorithm lets $Z = \{\, [q] \mid q \in Q_{M'} \,\}$, which is finite since $Q_{M'}$ is finite. In the case of our example, Z is

$$\{\{\mathsf{A}\}, \{\mathsf{B}, \mathsf{D}\}, \{\mathsf{C}\}, \{\mathsf{E}, \mathsf{F}\}\}.$$

Finally, the algorithm defines the DFA N as follows:

- $Q_N = \{\, \overline{P} \mid P \in Z \,\}$;
- $s_N = \overline{[s_{M'}]}$;
- $A_N = \{\, \overline{P} \mid P \in Z \text{ and } \min\, P \in A_{M'} \,\}$; and
- $T_N = \{\, (\overline{P}, a, \overline{[\delta_{M'}(\min\, P, a)]}) \mid P \in Z \text{ and } a \in \textbf{alphabet}\, M' \,\}$.

Then N is a DFA with alphabet **alphabet** M' and, for all $P \in Z$ and $a \in$ **alphabet** M', $\delta_N(\overline{P}, a) = \overline{[\delta_{M'}(\min\ P, a)]}$.

In the case of our example, we have that

- $Q_N = \{\langle \mathsf{A}\rangle, \langle \mathsf{B}, \mathsf{D}\rangle, \langle \mathsf{C}\rangle, \langle \mathsf{E}, \mathsf{F}\rangle\}$;
- $s_N = \langle A\rangle$; and
- $A_N = \{\langle \mathsf{E}, \mathsf{F}\rangle\}$.

We compute the elements of T_N as follows.

- Since $\{\mathsf{A}\} \in Z$ and $[\delta_{M'}(\mathsf{A}, 0)] = [\mathsf{B}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{A}\rangle, 0, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{A}\} \in Z$ and $[\delta_{M'}(\mathsf{A}, 1)] = [\mathsf{C}] = \{\mathsf{C}\}$, we have that $(\langle \mathsf{A}\rangle, 1, \langle \mathsf{C}\rangle) \in T_N$.

- Since $\{\mathsf{C}\} \in Z$ and $[\delta_{M'}(\mathsf{C}, 0)] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{C}\rangle, 0, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{C}\} \in Z$ and $[\delta_{M'}(\mathsf{C}, 1)] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{C}\rangle, 1, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

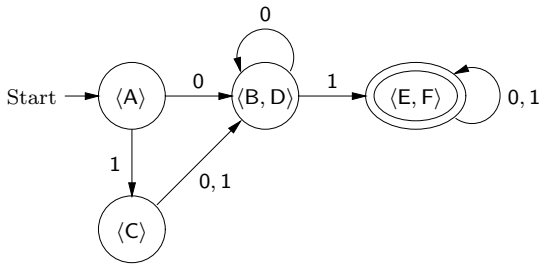
- Since $\{\mathsf{B}, \mathsf{D}\} \in Z$ and $[\delta_{M'}(\mathsf{B}, 0)] = [\mathsf{D}] = \{\mathsf{B}, \mathsf{D}\}$, we have that $(\langle \mathsf{B}, \mathsf{D}\rangle, 0, \langle \mathsf{B}, \mathsf{D}\rangle) \in T_N$.

  Since $\{\mathsf{B}, \mathsf{D}\} \in Z$ and $[\delta_{M'}(\mathsf{B}, 1)] = [\mathsf{E}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{B}, \mathsf{D}\rangle, 1, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

- Since $\{\mathsf{E}, \mathsf{F}\} \in Z$ and $[\delta_{M'}(\mathsf{E}, 0)] = [\mathsf{F}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{E}, \mathsf{F}\rangle, 0, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

  Since $\{\mathsf{E}, \mathsf{F}\} \in Z$ and $[\delta_{M'}(\mathsf{E}, 1)] = [\mathsf{F}] = \{\mathsf{E}, \mathsf{F}\}$, we have that $(\langle \mathsf{E}, \mathsf{F}\rangle, 1, \langle \mathsf{E}, \mathsf{F}\rangle) \in T_N$.

Thus our DFA N is:



Back in the general case, we have the following lemmas.

Lemma 3.13.7

(1) For all $q \in Q_{M'}$, $\overline{[q]} \in A_N$ iff $q \in A_{M'}$.

(2) For all $q \in Q_{M'}$ *and* $a \in$ **alphabet** M', $\delta_N(\overline{[q]}, a) = \overline{[\delta_{M'}(q, a)]}$.

(3) For all $q \in Q_{M'}$ *and* $w \in (\textbf{alphabet}\, M')^*$, $\delta_N(\overline{[q]}, w) = \overline{[\delta_{M'}(q, w)]}$.

(4) For all $w \in (\textbf{alphabet}\, M')^*$, $\delta_N(s_N, w) = \overline{[\delta_{M'}(s_{M'}, w)]}$.

Proof. (1) and (2) follow easily by Lemma 3.13.5(2)–(3). Part (3) follows from Part (2) by left string induction. For Part (4), suppose $w \in (\textbf{alphabet}\, M')^*$. By Part (3), we have

$$\delta_N(s_N, w) = \delta_N(\overline{[s_{M'}]}, w) = \overline{[\delta_{M'}(s_{M'}, w)]}.$$

□

Lemma 3.13.8
$L(N) = L(M')$.

Proof. Suppose $w \in L(N)$. Then $w \in (\textbf{alphabet}\, N)^* = (\textbf{alphabet}\, M')^*$ and $\delta_N(s_N, w) \in A_N$. By Lemma 3.13.7(4), we have that

$$\overline{[\delta_{M'}(s_{M'}, w)]} = \delta_N(s_N, w) \in A_N,$$

so that $\delta_{M'}(s_{M'}, w) \in A_{M'}$, by Lemma 3.13.7(1). Thus $w \in L(M')$.

Suppose $w \in L(M')$. Then $w \in (\textbf{alphabet}\, M')^* = (\textbf{alphabet}\, N)^*$ and $\delta_{M'}(s_{M'}, w) \in A_{M'}$. By Lemma 3.13.7(1) and (4), we have that

$$\delta_N(s_N, w) = \overline{[\delta_{M'}(s_{M'}, w)]} \in A_N.$$

Hence $w \in L(N)$. □

Lemma 3.13.9
N is deterministically simplified.

Proof. To see that all elements of N are reachable, suppose $q \in Q_{M'}$. Because M' is deterministically simplified, there is a $w \in (\textbf{alphabet}\, M')^*$ such that $q = \delta_{M'}(s_{M'}, w)$. Thus $\delta_N(s_N, w) = \overline{[\delta_{M'}(s_{M'}, w)]} = \overline{[q]}$.

Next, we show that, for all $q \in Q_{M'}$, if q is live, then $\overline{[q]}$ is live. Suppose $q \in Q_{M'}$ is live, so there is a $w \in (\textbf{alphabet}\, M')^*$ such that $\delta_{M'}(q, w) \in A_{M'}$. Thus $\delta_N(\overline{[q]}, w) = \overline{[\delta_{M'}(q, w)]} \in A_N$, showing that $\overline{[q]}$ is live.

Thus, we have that, for all $q \in Q_{M'}$, if $\overline{[q]}$ is dead, then q is dead. But, M' has at most one dead state, and thus we have that N has at most one dead state. □

Lemma 3.13.10
Suppose N' is a DFA such that $N' \approx M'$, **alphabet** $N' =$ **alphabet** M' *and $|Q_{N'}| \leq |Q_N|$. Then N' is isomorphic to N.*

Proof. We have that $L(N') = L(M')$. And the states of M' and N are all reachable. Let the relation h between $Q_{N'}$ and Q_N be

$$\{\, (\delta_{N'}(s_{N'}, w), \delta_N(s_N, w)) \mid w \in (\mathbf{alphabet}\, M')^* \,\}.$$

Since every state of N is reachable, it follows that $\mathbf{range}\, h = Q_N$.

To see that h is a function, suppose $x, y \in (\mathbf{alphabet}\, M')^*$ and $\delta_{N'}(s_{N'}, x) = \delta_{N'}(s_{N'}, y)$. We must show that $\delta_N(s_N, x) = \delta_N(s_N, y)$. Since $\delta_N(s_N, x) = \overline{[\delta_{M'}(s_{M'}, x)]}$ and $\delta_N(s_N, y) = \overline{[\delta_{M'}(s_{M'}, y)]}$, it will suffice to show that $(\delta_{M'}(s_{M'}, x), \delta_{M'}(s_{M'}, y)) \in Y$. By Lemma 3.13.5(1), it will suffice to show that, $\delta_{M'}(\delta_{M'}(s_{M'}, x), z) \in A_{M'}$ iff $\delta_{M'}(\delta_{M'}(s_{M'}, y), z) \in A_{M'}$, for all $z \in (\mathbf{alphabet}\, M')^*$. Suppose $z \in (\mathbf{alphabet}\, M')^*$. We must show that $\delta_{M'}(\delta_{M'}(s_{M'}, x), z) \in A_{M'}$ iff $\delta_{M'}(\delta_{M'}(s_{M'}, y), z) \in A_{M'}$

We will show the "only if" direction, the other direction being similar. Suppose $\delta_{M'}(\delta_{M'}(s_{M'}, x), z) \in A_{M'}$. We must show that $\delta_{M'}(\delta_{M'}(s_{M'}, y), z) \in A_{M'}$. Because $\delta_{M'}(s_{M'}, xz) = \delta_{M'}(\delta_{M'}(s_{M'}, x), z) \in A_{M'}$, we have that $xz \in L(M') = L(N')$. Since $xz \in L(N')$ and $\delta_{N'}(s_{N'}, x) = \delta_{N'}(s_{N'}, y)$, we have that

$$\begin{aligned}\delta_{N'}(s_{N'}, yz) &= \delta_{N'}(\delta_{N'}(s_{N'}, y), z) = \delta_{N'}(\delta_{N'}(s_{N'}, x), z) \\ &= \delta_{N'}(s_{N'}, xz) \in A_{N'},\end{aligned}$$

so that $yz \in L(N') = L(M')$. Hence $\delta_{M'}(\delta_{M'}(s_{M'}, y), z) = \delta_{M'}(s_{M'}, yz) \in A_{M'}$.

Because h is a function and $\mathbf{range}\, h = Q_N$, we have that $|Q_N| \leq |\mathbf{domain}\, f| \leq Q_{N'}$. But $|Q_{N'}| \leq |Q_N|$, and thus $|Q_{N'}| = |Q_N|$. Because $Q_{N'}$ and Q_N are finite, it follows that $\mathbf{domain}\, h = Q_{N'}$ and h is injective, so that h is a bijection from $Q_{N'}$ to Q_N. Thus, every state of N' is reachable, and, for all $w \in (\mathbf{alphabet}\, M')^* = (\mathbf{alphabet}\, N)^* = (\mathbf{alphabet}\, N')^*$, $h(\delta_{N'}(s_{N'}, w)) = \delta_N(s_N, w)$. The remainder of the proof that h is an isomorphism from N' to N is easy. □

Exercise 3.13.11
Expand on the last sentence of Lemma 3.13.10, proving that h is an isomorphism from N' to N is easy.

We define a function $\mathbf{minimize} \in \mathbf{DFA} \to \mathbf{DFA}$ by: $\mathbf{minimize}\, M$ is the result of running the above algorithm on input M.

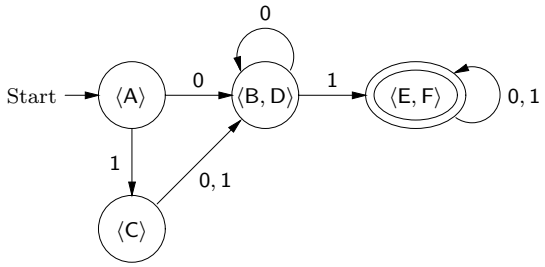
Putting the above results together, we have the following theorem:

Theorem 3.13.12
For all $M \in \mathbf{DFA}$*:*

- $\mathbf{minimize}\, M \approx M$*;*
- $\mathbf{alphabet}(\mathbf{minimize}\, M) = \mathbf{alphabet}(L(M))$*;*
- $\mathbf{minimize}\, M$ *is deterministically simplified; and*

- *for all* $N \in$ **DFA**, *if* $N \approx M$, **alphabet** $N =$ **alphabet**$(L(M))$ *and* $|Q_N| \leq |Q_{\textbf{minimize}\, M}|$, *then* N *is isomorphic to* **minimize** M.

Thus



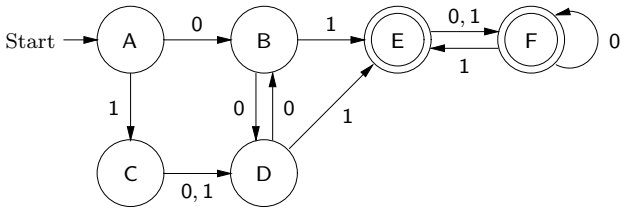
is, up to isomorphism, the only four-or-fewer state DFA with alphabet $\{0, 1\}$ that is equivalent to M.

The Forlan module DFA includes the function

```
val minimize : dfa -> dfa
```

for minimizing DFAs.

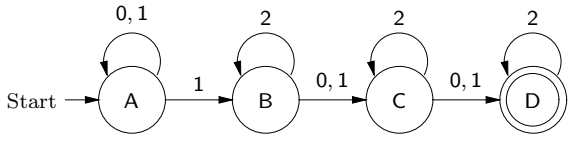
For example, if `dfa` of type `dfa` is bound to our example DFA



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
```

Finally, let's revisit an example from Section 3.11. Suppose `nfa` is the 4-state NFA



As we saw, our NFA-to-DFA conversion algorithm converts `nfa` to a DFA `dfa` with 16 states:

```
- val dfa = nfaToDFA nfa;
val dfa = - : dfa
- DFA.numStates dfa;
val it = 16 : int
```

We can now use Forlan to verify that there is no DFA with fewer than 16 states that accepts the same language as `nfa`:

```
- val dfa' = DFA.minimize dfa;
val dfa' = - : dfa
- DFA.isomorphic(dfa', dfa);
val it = true : bool
- DFA.numStates dfa';
val it = 16 : int
```

Thus we have an example where the smallest DFA accepting a language requires exponentially more states than the smallest NFA accepting that language. (This is true even though we haven't proven that an NFA must have at least 4 states to accept the same language as `nfa`.)

3.13.3 Notes

Our algorithm for testing whether two DFAs are equivalent can be found in the literature, but I don't know of other textbooks that present it. As described above, a simple extension of the algorithm provides counterexamples to justify non-equivalence. The material on DFA minimization is completely standard.

3.14 The Pumping Lemma for Regular Languages

In this section we consider techniques for showing that particular languages are not regular. Consider the language

$$L = \{ \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \} = \{\%, \mathsf{01}, \mathsf{0011}, \mathsf{000111}, \ldots\}.$$

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 $\mathsf{0}$'s it has seen so far, and thus won't be able to insist that the correct number of $\mathsf{1}$'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 3.14.1 (Pumping Lemma for Regular Languages)
For all regular languages L, there is an $n \in \mathbb{N} - \{0\}$ 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) $uv^i w \in L$, 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 non-zero natural number n such that the property of the lemma holds. We have no control over the value of n; all we know is that $n \geq 1$ and the property of the lemma holds. We can then give the lemma a string z that is in L and has at least n symbols. We'll have to choose z as a function of n, because we don't know what n is. 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; all we know is that $z = uvw$ and (1)–(3) hold. (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 $L = \{ \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \}$ is non-regular.

Proposition 3.14.2
L is not regular.

Proof. Suppose, toward a contradiction, that L is regular. Thus there is an $n \in \mathbb{N} - \{0\}$ with the property of the Pumping Lemma. Suppose $z = \mathsf{0}^n\mathsf{1}^n$. 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 $\mathsf{0}^n\mathsf{1}^n = z = uvw$, (1) tells us that there are $i, j, k \in \mathbb{N}$ such that

$$u = \mathsf{0}^i, \quad v = \mathsf{0}^j, \quad w = \mathsf{0}^k\mathsf{1}^n, \quad i + j + k = n.$$

By (2), we have that $j \geq 1$, and thus that $i + k = n - j < n$. By (3), we have that

$$\mathsf{0}^{i+k}\mathsf{1}^n = \mathsf{0}^i\mathsf{0}^k\mathsf{1}^n = uw = u\%w = uv^0w \in L.$$

Thus $i + k = n$—contradiction. Thus L is not regular. □

In the preceding proof, we obtained a contradiction by pumping zero times (uv^0w), but pumping two or more times (uv^2w, ...) would also have worked. For a case when pumping zeros times is insufficient, consider $A = \{ \mathsf{0}^n\mathsf{1}^m \mid n < m \}$.

Given $n \in \mathbb{N} - \{0\}$ by the Pumping Lemma, we can let $z = \mathsf{0}^n\mathsf{1}^{n+1}$, obliging the lemma to split z into uvw, in such a way that (1)–(3) hold. Hence v will consist entirely of $\mathsf{0}$'s. Pumping v zero times won't take us outside of A. On the other hand uv^2w will have at least as many $\mathsf{0}$'s as $\mathsf{1}$'s, giving us the needed contradiction.

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 $n = |Q_M|$. Since $Q_M \neq \emptyset$, we have $n \geq 1$, so that $n \in \mathbb{N} - \{0\}$. 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 $i \in \mathbb{N}$. □

Suppose $L' = \{\, w \in \{\mathsf{0}, \mathsf{1}\}^* \mid w \text{ has an equal number of } \mathsf{0}\text{'s and } \mathsf{1}\text{'s} \,\}$. 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 3.12 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 generated by the regular expressions $\mathsf{0}$ and $\mathsf{1}$). Thus, by Theorem 3.12.28, we have that $\{\mathsf{0}\}^*\{\mathsf{1}\}^*$ is regular. Hence, by Theorem 3.12.28 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 $n \in \mathbb{N} - \{0\}$ 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 may have chosen $n = 2$, and may break z up into $u = \%$, $v = \mathsf{01}$ and $w = (\mathsf{01})^{n-1}$—with the consequence that (1)–(3) hold.

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 $\mathsf{0}$'s and $\mathsf{1}$'s. Hence $i + k = n$, giving us the needed contradiction.

3.14.1 Experimenting with the Pumping Lemma Using Forlan

The Forlan module `LP` (see Section 3.4) 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 strsOfValidPumpingDivision :
      pumping_division -> str * str * str
val pumpValidPumpingDivision   : pumping_division * int -> lp
val findValidPumpingDivision   : lp -> pumping_division
```

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;
- the end state of lp_2 is equal to the start state of lp_3; and
- the label of lp_2 is nonempty.

The function `checkPumpingDivision` checks whether a pumping division is valid, silently returning `()`, if it is, and issuing an error message explaining why it isn't, if it isn't. The function `validPumpingDivision` tests whether a pumping division is valid. The function `strsOfValidPumpingDivision` returns the triple consisting of the labels of the three components of a pumping division, in order. It issues an error message if the pumping division isn't valid. The function `pumpValidPumpingDivision` expects a pair (pd, n), where pd is a valid pumping division and $n \geq 0$. It issues an error message if pd isn't valid, or n is negative. Otherwise, it returns

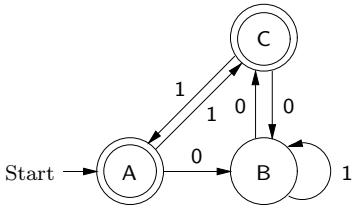
$$\mathbf{join}(\sharp 1\, pd, \mathbf{join}(lp', \mathbf{join}(\sharp 3\, pd))),$$

where lp' is the result of joining $\sharp 2\, pd$ with itself n times (the empty labeled path whose single state is $\sharp 2\, pd$'s start/end state, if $n = 0$). Finally, the function `findValidPumpingDivision` takes in a labeled path lp, and tries to find a pumping division (lp_1, lp_2, lp_3) such that:

- (lp_1, lp_2, lp_3) is valid;
- `pumpValidPumpingDivision`$((lp_1, lp_2, lp_3), 1) = lp$; and
- there is no repetition of states in the result of joining lp_1 and the result of removing the last step of lp_2.

`findValidPumpingDivision` issues an error message if no such pumping division exists.

For example, suppose the DFA `dfa` is bound to the DFA



Then we can proceed as follows:

```
- val lp = DFA.findAcceptingLP dfa (Str.input "");
@ 001010
@ .
val lp = - : lp
- LP.output("", lp);
A, 0 => B, 0 => C, 1 => A, 0 => B, 1 => B, 0 => C
val it = () : unit
- val pd = LP.findValidPumpingDivision 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, 0 => B, 1 => B, 0 => C
val it = () : unit
- val (u, v, w) = LP.strsOfValidPumpingDivision pd;
val u = [] : str
val v = [-,-,-] : str
val w = [-,-,-] : str
- (Str.toString u, Str.toString v, Str.toString v);
val it = ("%","001","001") : string * string * string
- val lp' = LP.pumpValidPumpingDivision(pd, 2);
val lp' = - : lp
- LP.output("", lp');
A, 0 => B, 0 => C, 1 => A, 0 => B, 0 => C, 1 => A, 0 => B, 1 =>
B, 0 => C
val it = () : unit
- Str.output("", LP.label lp');
001001010
val it = () : unit
```

3.14.2 Notes

The Pumping Lemma is usually proved using a DFA accepting the given regular language. But because we have described the meaning of automata via labeled paths, we can do the proof with an NFA, as it has nothing to do with determinacy. Forlan's support for experimenting with the Pumping Lemma is novel.

3.15 Applications of Finite Automata and Regular Expressions

In this section we consider three applications of the material from Chapter 3: searching for regular expressions in files; lexical analysis; and the design of finite state systems.

3.15.1 Representing Character Sets and Files

Our first two 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 ⟨newline⟩ and ⟨space⟩, respectively.

In the following two subsections, we will work with a mostly unspecified alphabet Σ representing some character set. We assume that the symbols 0–9, a–z, A–Z, ⟨space⟩ and ⟨newline⟩ are elements of Σ. A *line* is a string consisting of an element of $(\Sigma - \{\langle\mathsf{newline}\rangle\})^*$; and, a *file* consists of the concatenation of some number of lines, separated by occurrences of ⟨newline⟩. E.g., 0a⟨newline⟩⟨newline⟩6 is a file with three lines (0a, % and 6), and ⟨newline⟩ is a file with two lines, both consisting of %.

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 ⟨newline⟩, listed in strictly ascending 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

$$\mathsf{0} + \mathsf{1} + \cdots + \mathsf{9}.$$

3.15.2 Searching for Regular Expression 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 $\beta = [\mathbf{any}]^* \, \alpha \, [\mathbf{any}]^*$. To do this, we can first translate β to a DFA M with alphabet $\Sigma - \{\langle\mathsf{newline}\rangle\}$. 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 (or EFA or NFA) N, and use the algorithm from Section 3.6 to find all lines that are accepted by N.

3.15.3 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) $code_1, code_2, \ldots, code_n$ that process elements of Σ^*.

For example, we might have

$$\alpha_1 = \langle \mathsf{space} \rangle + \langle \mathsf{newline} \rangle,$$
$$\alpha_2 = [\mathbf{letter}]\,([\mathbf{letter}] + [\mathbf{digit}])^*,$$
$$\alpha_3 = [\mathbf{digit}]\,[\mathbf{digit}]^*\,(\% + \mathsf{E}\,[\mathbf{digit}]\,[\mathbf{digit}]^*),$$
$$\alpha_4 = [\mathbf{any}].$$

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 generated 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.

What happens when we process the file $\mathsf{123Easy}\langle\mathsf{space}\rangle\mathsf{1E2}\langle\mathsf{newline}\rangle$ using 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.

Now, we consider 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 $acc \in L(M_i)$;
- 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 $code_{mach}$, 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 ⟨newline⟩ from the remaining input, and supplies this string to code $code_1$.

3.15.4 Design of Finite State Systems

Deterministic finite automata give us a means to efficiently—both in terms of time and space—check membership in a regular language. In terms of time, a single left-to-right scan of the string is needed. And we only need enough space to encode the DFA, and to keep track of what state we are in at each point, as well as what part of the string remains to be processed. But if the string to be checked is supplied, symbol-by-symbol, from our environment, we don't need to store the string at all.

Consequently, DFAs may be easily and efficiently implemented in both hardware and software. One can design DFAs by hand, and test them using Forlan. But DFA minimization plus the operations on automata and regular expressions of Section 3.12, give us an alternative—and very powerful—way of designing finite state systems, which we will illustrate with two examples.

As the first example, suppose we wish to find a DFA M such that $L(M) = X$, where

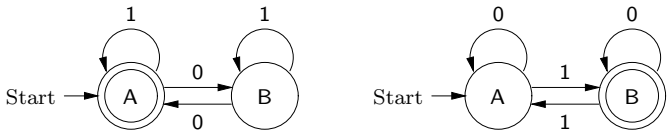
$$X = \{\, w \in \{0,1\}^* \mid w \text{ has an even number of 0's or an odd number of 1's} \,\}.$$

First, we can note that $X = Y_1 \cup Y_2$, where

$$Y_1 = \{\, w \in \{0,1\}^* \mid w \text{ has an even number of 0's} \,\}, \text{ and}$$
$$Y_2 = \{\, w \in \{0,1\}^* \mid w \text{ has an odd number of 1's} \,\}.$$

Since we have a union operation on EFAs (Forlan doesn't provide a union operation on DFAs), if we can find EFAs accepting Y_1 and Y_2, we can combine them into a EFA that accepts X. Then we can convert this EFA to a DFA, and then minimize the DFA.

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

$$\textbf{renameStatesCanonically}(\textbf{minimize}(N)),$$

where N is the DFA

$$\textbf{nfaToDFA}(\textbf{efaToNFA}(\textbf{union}(N_1, N_2))).$$

Then

$$\begin{aligned}
L(M) &= L(\textbf{renameStatesCanonically}(\textbf{minimize}\, N)) \\
&= L(\textbf{minimize}\, N) \\
&= L(N) \\
&= L(\textbf{nfaToDFA}(\textbf{efaToNFA}(\textbf{union}(N_1, N_2)))) \\
&= L(\textbf{efaToNFA}(\textbf{union}(N_1, N_2))) \\
&= L(\textbf{union}(N_1, N_2)) \\
&= L(N_1) \cup L(N_2) \\
&= Y_1 \cup Y_2 \\
&= X,
\end{aligned}$$

showing that M is correct.

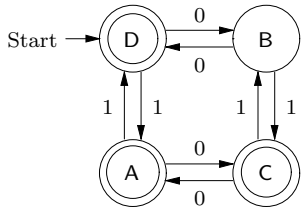
Suppose M' is a DFA that accepts X. Since $M' \approx N$, we have that **minimize**(N), and thus M, has no more states than M'. Thus M has as few states as is possible.

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 **union**, **efaToNFA**, **nfaToDFA**, **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 efa = EFA.union(injDFAToEFA dfa1, injDFAToEFA dfa2);
val efa = - : efa
- val dfa' = nfaToDFA(efaToNFA efa);
val dfa' = - : dfa
- DFA.numStates dfa';
val it = 5 : int
- val dfa = DFA.renameStatesCanonically(DFA.minimize dfa');
val dfa = - : dfa
- DFA.numStates dfa;
val it = 4 : int
- DFA.output("", dfa);
{states} A, B, C, D {start state} D {accepting states} A, C, D
{transitions}
A, 0 -> C; A, 1 -> D; B, 0 -> D; B, 1 -> C; C, 0 -> A; C, 1 -> B;
D, 0 -> B; D, 1 -> A
val it = () : unit
```

Thus M is:



Of course, this claim assumes that Forlan is correctly implemented.

We conclude this subsection by considering a second, more involved example of DFA design. 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**, which we can then mimimize. (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 3.15.1
AllLongStutter $= \{0,1\}^* -$ **SomeLongNotStutter**.

Proof. Suppose $w \in$ **AllLongStutter**, so that $w \in \{0,1\}^*$ and, for all substrings v of w, if v is long, then v stutters. Suppose, toward a contradiction, that $w \in$ **SomeLongNotStutter**. Then there is a substring v of w such that v is long and doesn't stutter—contradiction. Thus $w \notin$ **SomeLongNotStutter**, completing the proof that $w \in \{0,1\}^* -$ **SomeLongNotStutter**.

Suppose $w \in \{0,1\}^* -$ **SomeLongNotStutter**, so that $w \in \{0,1\}^*$ and $w \notin$ **SomeLongNotStutter**. To see that $w \in$ **AllLongStutter**, suppose v is a substring of w and v is long. Suppose, toward a contradiction, that v doesn't stutter. Then $w \in$ **SomeLongNotStutter**—contradiction. Hence v stutters. □

Next, it's convenient to work bottom-up for a bit. Let

$$\begin{aligned}
\textbf{Long} &= \{\, w \in \{0,1\}^* \mid w \text{ is long} \,\},\\
\textbf{Stutter} &= \{\, w \in \{0,1\}^* \mid w \text{ stutters} \,\},\\
\textbf{NotStutter} &= \{\, w \in \{0,1\}^* \mid w \text{ doesn't stutter} \,\}, \text{ and}\\
\textbf{LongAndNotStutter} &= \{\, w \in \{0,1\}^* \mid w \text{ is long and doesn't stutter} \,\}.
\end{aligned}$$

The following lemma is easy to prove:

Lemma 3.15.2
(1) **NotStutter** $= \{0,1\}^* -$ **Stutter**.

(2) **LongAndNotStutter** $=$ **Long** $\cap$ **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 $x, y \in \{0,1\}^*$ and $v \in$ **LongAndNotStutter**. This suggests the following lemma:

Lemma 3.15.3
SomeLongNotStutter $= \{0,1\}^*$ **LongAndNotStutter** $\{0,1\}^*$.

Proof. Suppose $w \in$ **SomeLongNotStutter**, so that $w \in \{0,1\}^*$ and there is a substring v of w such that v is long and doesn't stutter. Thus $v \in$ **LongAndNotStutter**, and $w = xvy$ for some $x, y \in \{0,1\}^*$. Hence $w = xvy \in \{0,1\}^*$ **LongAndNotStutter** $\{0,1\}^*$.

Suppose $w \in \{0,1\}^*$ **LongAndNotStutter** $\{0,1\}^*$, so that $w = xvy$ for some $x, y \in \{0,1\}^*$ and $v \in$ **LongAndNotStutter**. Hence v is long and doesn't stutter. Thus v is a long substring of w that doesn't stutter, showing that $w \in$ **SomeLongNotStutter**. □

Because of the preceding lemma, we can construct an EFA accepting **SomeLongNotStutter** from a DFA accepting $\{0,1\}^*$ 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 **regToEFA** $\in$ **Reg** $\to$ **EFA**, **efaToDFA** $\in$ **EFA** $\to$ **DFA**, **regToDFA** $\in$ **Reg** $\to$ **DFA** and **minAndRen** $\in$ **DFA** $\to$ **DFA** by:

$$\begin{aligned}
\textbf{regToEFA} &= \textbf{faToEFA} \circ \textbf{regToFA}, \\
\textbf{efaToDFA} &= \textbf{nfaToDFA} \circ \textbf{efaToNFA}, \\
\textbf{regToDFA} &= \textbf{efaToDFA} \circ \textbf{regToEFA}, \text{ and} \\
\textbf{minAndRen} &= \textbf{renameStatesCanonically} \circ \textbf{minimize}.
\end{aligned}$$

Lemma 3.15.4

(1) For all $\alpha \in$ **Reg**, $L($**regToEFA**$(\alpha)) = L(\alpha)$.

(2) For all $M \in$ **EFA**, $L($**efaToDFA**$(M)) = L(M)$.

(3) For all $\alpha \in$ **Reg**, $L($**regToDFA**$(\alpha)) = L(\alpha)$.

(4) For all $M \in$ **DFA**, $L($**minAndRen**$(M)) = L(M)$ *and, for all* $N \in$ **DFA**, *if* $L(N) = L(M)$, *then* **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 $M \in$ **DFA**. By Theorem 3.13.12(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 3.13.12(4), **minimize**(M) has no more states than N. Hence **renameStatesCanonically**(**minimize**(M)) has no more states than N, showing that **minAndRen**(M) has no more states than N. □

Let the regular expression **allStrReg** be $(0+1)^*$. Clearly $L(\mathbf{allStrReg}) = \{0,1\}^*$. Let the DFA **allStrDFA** be

$$\mathbf{minAndRen}(\mathbf{regToDFA}\,\mathbf{allStrReg}).$$

Lemma 3.15.5
$L(\mathbf{allStrDFA}) = \{0,1\}^*$.

Proof. By Lemma 3.15.4, we have that

$$\begin{aligned} L(\mathbf{allStrDFA}) &= L(\mathbf{minAndRen}(\mathbf{regToDFA}\,\mathbf{allStrReg})) \\ &= L(\mathbf{regToDFA}\,\mathbf{allStrReg}) \\ &= L(\mathbf{allStrReg}) \\ &= \{0,1\}^*. \end{aligned}$$

□

(Not surprisingly, **allStrDFA** will have a single state.) Let the EFA **allStrEFA** be the DFA **allStrDFA**. Thus $L(\mathbf{allStrEFA}) = \{0,1\}^*$.

Let the regular expression **longReg** be

$$(0+1)^5(0+1)^*.$$

Lemma 3.15.6
$L(\mathbf{longReg}) = \mathbf{Long}$.

Proof. Since $L(\mathbf{longReg}) = \{0,1\}^5\{0,1\}^*$, it will suffice to show that $\{0,1\}^5\{0,1\}^* = \mathbf{Long}$.

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 $w \in \mathbf{Long}$.

Suppose $w \in \mathbf{Long}$, 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 $w = (abcde)x \in \{0,1\}^5\{0,1\}^*$.
□

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

$$(0+1)^*(00+11)(0+1)^*.$$

Lemma 3.15.7
$L(\mathbf{stutterReg}) = \mathbf{Stutter}$.

Proof. Since $L(\mathbf{stutterReg}) = \{0,1\}^*\{00,11\}\{0,1\}^*$, it will suffice to show that $\{0,1\}^*\{00,11\}\{0,1\}^* = \mathbf{Stutter}$, and this is easy. □

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 3.15.8
$L(\mathbf{notStutterDFA}) = \mathbf{NotStutter}$.

Proof. Let M be

$$\mathbf{minAndRen}(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{stutterDFA})).$$

By Lemma 3.15.2(1), we have that

$$\begin{aligned} L(\mathbf{notStutterDFA}) &= L(M) \\ &= L(\mathbf{minus}(\mathbf{allStrDFA}, \mathbf{stutterDFA})) \\ &= L(\mathbf{allStrDFA}) - L(\mathbf{stutterDFA}) \\ &= \{0,1\}^* - \mathbf{Stutter} \\ &= \mathbf{NotStutter}. \end{aligned}$$

□

Let **longAndNotStutterDFA** be the DFA

$$\mathbf{minAndRen}(\mathbf{inter}(\mathbf{longDFA}, \mathbf{notStutterDFA})).$$

Lemma 3.15.9
$L(\mathbf{longAndNotStutterDFA}) = \mathbf{LongAndNotStutter}$.

Proof. Let M be

$$\mathbf{minAndRen}(\mathbf{inter}(\mathbf{longDFA}, \mathbf{notStutterDFA})).$$

By Lemma 3.15.2(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 simply let the EFA **longAndNotStutterEFA** be **longAndNotStutterDFA**. Thus we have that $L(\textbf{longAndNotStutterEFA}) = \textbf{LongAndNotStutter}$.

Let **someLongNotStutterEFA** be the EFA

$$\textbf{renameStatesCanonically}(\textbf{concat}(\textbf{allStrEFA}, \textbf{concat}(\textbf{longAndNotStutterEFA}, \textbf{allStrEFA}))).$$

Lemma 3.15.10
$L(\textbf{someLongNotStutterEFA}) = \textbf{SomeLongNotStutter}$.

Proof. We have that

$$\begin{aligned} L(\textbf{someLongNotStutterEFA}) &= L(\textbf{renameStatesCanonically}\, M) \\ &= L(M), \end{aligned}$$

where M is

$$\textbf{concat}(\textbf{allStrEFA}, \textbf{concat}(\textbf{longAndNotStutterEFA}, \textbf{allStrEFA})).$$

And, by Lemma 3.15.3, we have that

$$\begin{aligned} L(M) &= L(\textbf{allStrEFA})\, L(\textbf{longAndNotStutterEFA})\, L(\textbf{allStrEFA}) \\ &= \{0,1\}^*\, \textbf{LongAndNotStutter}\, \{0,1\}^* \\ &= \textbf{SomeLongNotStutter}. \end{aligned}$$

□

Let **someLongNotStutterDFA** be the DFA

$$\textbf{minAndRen}(\textbf{efaToDFA}\, \textbf{someLongNotStutterEFA}).$$

Lemma 3.15.11
$L(\textbf{someLongNotStutterDFA}) = \textbf{SomeLongNotStutter}$.

Proof. Follows by an easy calculation. □

Finally, let **allLongStutterDFA** be the DFA

$$\textbf{minAndRen}(\textbf{minus}(\textbf{allStrDFA}, \textbf{someLongNotStutterDFA})).$$

Lemma 3.15.12
$L(\textbf{allLongStutterDFA}) = \textbf{AllLongStutter}$ *and, for all* $N \in \textbf{DFA}$, *if* $L(N) = \textbf{AllLongStutter}$, *then* **allLongStutterDFA** *has no more states than* N.

Proof. We have that

$$L(\textbf{allLongStutterDFA}) = L(\textbf{minAndRen}(M)) = L(M),$$

where M is

$$\textbf{minus}(\textbf{allStrDFA}, \textbf{someLongNotStutterDFA}).$$

Then, by Lemma 3.15.1, we have that

$$\begin{aligned} L(M) &= L(\textbf{allStrDFA}) - L(\textbf{someLongNotStutterDFA}) \\ &= \{0,1\}^* - \textbf{SomeLongNotStutter} \\ &= \textbf{AllLongStutter}. \end{aligned}$$

Suppose $N \in \textbf{DFA}$ and $L(N) = \textbf{AllLongStutter}$. Thus $L(N) = L(M)$, so that **allLongStutterDFA** has no more states than N, by Lemma 3.15.4(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 3.1.

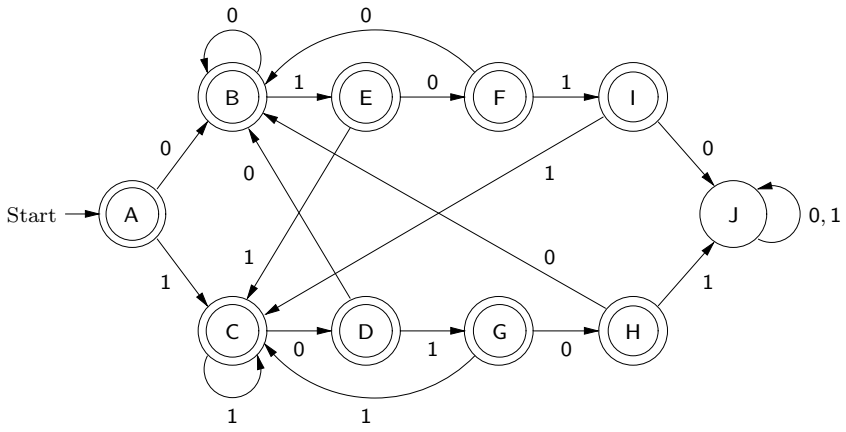


Figure 3.1: DFA Accepting **AllLongStutter**

Exercise 3.15.13
Define **diff** $\in \{0,1\}^* \to \mathbb{Z}$ by: for all $w \in \{0,1\}^*$,

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

Let **AllPrefixGood** $= \{\, w \in \{0,1\}^* \mid \text{for all prefixes } v \text{ of } w, |\textbf{diff}\, v| \leq 2 \,\}$. Suppose we have a DFA **allPrefixGoodDFA**, and we have already proved (see Exercise 3.11.7) $L(\textbf{allPrefixGoodDFA}) = \textbf{AllPrefixGood}$. Let **AllSubstringGood** $= \{\, w \in \{0,1\}^* \mid \text{for all substrings } v \text{ of } w, |\textbf{diff}\, v| \leq 2 \,\}$. Show how we can transform **allPrefixGoodDFA** into a minimized DFA **allSubstringGoodDFA** such that $L(\textbf{allSubstringGoodDFA}) = \textbf{AllSubstringGood}$. Prove that your transformation is correct. Finally, realize your transformation in Forlan, and draw the resulting DFA.

3.15.5 Notes

Our treatment of searching for regular expressions in text file is standard, as is that of lexical analysis. But our approach to designing finite state systems depends upon having access to a toolset, like Forlan, that is embedded in a programming language and implements our algorithms for manipulating finite automata and regular expressions.

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 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; and begin to show how grammars can be processed using Forlan.

4.1.1 Grammars

A *context-free grammar* (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); and
- 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, and we often write (q, x) as $q \to x$).

In a context where we are only referring to a single grammar, 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 grammar G (of arithmetic expressions) as follows:

- $Q_G = \{\mathsf{E}\}$;
- $s_G = \mathsf{E}$; and
- $P_G = \{\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\ \mathsf{E} \to \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E},\ \mathsf{E} \to \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle,\ \mathsf{E} \to \langle\mathsf{id}\rangle\}$.

E.g., we can read the production $\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E}$ 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, 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

$$\begin{aligned}
&\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\\
&\mathsf{E} \to \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E},\\
&\mathsf{E} \to \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle,\\
&\mathsf{E} \to \langle\mathsf{id}\rangle,
\end{aligned}$$

or

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

The Forlan syntax for grammars is very similar to the notation used above. E.g., our example grammar can be described in Forlan's syntax by

```
{variables} E {start variable} E
{productions}
E -> E<plus>E; E -> E<times>E; E -> <openPar>E<closPar>; E -> <id>
```

or

```
{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 functions `numVariables` and `numProductions` return the numbers of variables and productions, respectively, of a grammar. And the function `equal` tests whether two grammars are equal, i.e., whether they have the same sets of variables, start variables, and sets of productions. During printing, Forlan merges productions into production families whenever possible.

The *alphabet of* a grammar G (**alphabet** G) is

$$\{\, a \in \mathbf{Sym} \mid \text{there are } q, x \text{ such that } q \to x \in P_G \text{ and } a \in \mathbf{alphabet}\, x \,\} - Q_G.$$

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
```

4.1.2 Parse Trees and Grammar Meaning

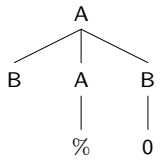
We will explain when strings are generated by grammars using the notion of a parse tree. The set **PT** of *parse trees* is the least subset of $\mathbf{Tree}(\mathbf{Sym} \cup \{\%\})$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$ and $pts \in \mathbf{List}\,\mathbf{PT}$, $(a, pts) \in \mathbf{PT}$; and

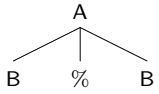
(2) for all $a \in \mathbf{Sym}$, $(a, [(\%, [\,])]) = a(\%) \in \mathbf{PT}$.

Note that the $(\mathbf{Sym} \cup \{\%\})$-tree $\% = (\%, [\,])$ is *not* a parse tree. 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.

Theorem 4.1.1 (Principle of Induction on Parse Trees)
Suppose $P(pt)$ is a property of an element $pt \in \mathbf{PT}$.

If

(1) for all $a \in \mathbf{Sym}$ and $trs \in \mathbf{List}\,\mathbf{PT}$, if (†) for all $i \in [1 : |trs|]$, $P(trs\,i)$, then $P((a, trs))$, and

(2) for all $a \in \mathbf{Sym}$, $P(a(\%))$,

then

$$\textit{for all } pt \in \mathbf{PT}, P(pt).$$

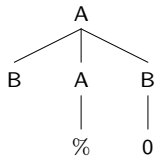
We refer to (†) as the inductive hypothesis.

We define the yield of a parse tree, as follows. The function $\mathbf{yield} \in \mathbf{PT} \to \mathbf{Str}$ is defined by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{yield}\,a = a$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$, $\mathbf{yield}(q(pt_1, \ldots, pt_n)) = \mathbf{yield}\,pt_1 \cdots \mathbf{yield}\,pt_n$; and
- 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 $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a = a \in \mathbf{alphabet}\, G$ **or** $a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\begin{aligned} &\mathbf{valid}_G(q(pt_1, \ldots, pt_n)) \\ &= q \to \mathbf{rootLabel}\, pt_1 \cdots \mathbf{rootLabel}\, pt_n \in P_G \textbf{ and} \\ &\quad \mathbf{valid}_G\, pt_1 \textbf{ and } \cdots \textbf{ and } \mathbf{valid}_G\, pt_n; \text{ and} \end{aligned}$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) = q \to \% \in P_G$.

We say that pt is *valid for* G iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to **valid**.

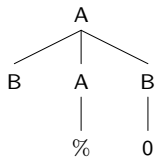
Suppose G is the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{BAB} \mid \%, \\ &\mathsf{B} \to \mathsf{0} \end{aligned}$$

(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

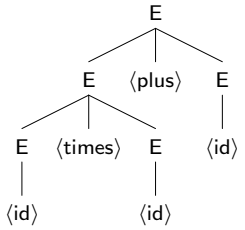


is valid for G.

And, if G is our grammar of arithmetic expressions,

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle,$$

then the parse tree



is valid for G.

Suppose G is a grammar, $w \in \mathbf{Str}$ and $a \in \mathbf{Sym}$. We say that w *is parsable from* a *using* G iff there is a parse tree pt such that:

- pt is valid for G;
- a is the root label of pt; and
- the yield of pt is w.

(Thus we will have that $w \in (Q_G \cup \mathbf{alphabet}\, G)^*$, and either $a \in Q_G$ or $a = w$.) We say that a string w *is generated from* a variable $q \in Q_G$ *using* G iff $w \in (\mathbf{alphabet}\, G)^*$ and w is parsable from q. And, we say that a string w *is generated by* a grammar G iff w is generated from s_G using G. The *language generated by* a grammar G ($L(G)$) is

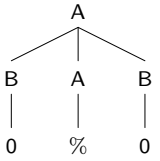
$$\{\, w \in \mathbf{Str} \mid w \text{ is generated by } G \,\}.$$

Proposition 4.1.2
For all grammars G, $\mathbf{alphabet}(L(G)) \subseteq \mathbf{alphabet}\, G$.

For example, if G is the grammar

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to \mathsf{0},$$

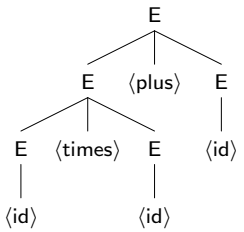
then $\mathsf{00}$ is generated by G, since $\mathsf{00} \in \{\mathsf{0}\}^* = (\mathbf{alphabet}\, G)^*$ and the parse tree



is valid for G, has $s_G = \mathsf{A}$ as its root label, and has $\mathsf{00}$ as its yield. And, if G is our grammar of arithmetic expressions,

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle,$$

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 $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle \in (\mathbf{alphabet}\, G)^*$ 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 $G \in \mathbf{Gram}$. 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 $\{0^0\}$, $\{0^1\}$, $\{0^2\}$, ..., 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 $\mathbf{CFLan} \subsetneq \mathbf{Lan}$, i.e., there are non-context-free languages. Later, we will see that $\mathbf{RegLan} \subsetneq \mathbf{CFLan}$.

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 Java program JForlan, can be used to view and edit parse trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

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

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to \mathsf{0}.$$

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)
@ .
line 1: "%" unexpected

uncaught exception Error
```

4.1.3 Grammar Synthesis

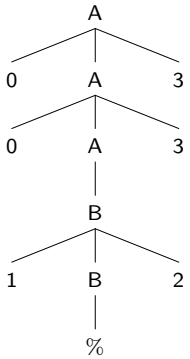
We conclude this section with a grammar synthesis example. Suppose $X = \{\, \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \mid n, m \in \mathbb{N} \,\}$. The key to finding a grammar G that generates X is to think of generating the strings of X from the outside in, in two phases. In the first phase, 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:

$$\begin{aligned} &\mathsf{0}\quad\ \mathsf{3},\\ &\mathsf{00}\quad\mathsf{33},\\ &\mathsf{001233}. \end{aligned}$$

This analysis leads us to the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{0A3},\\ &\mathsf{A} \to \mathsf{B},\\ &\mathsf{B} \to \mathsf{1B2},\\ &\mathsf{B} \to \%, \end{aligned}$$

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:



Exercise 4.1.3
Let

$$X = \{\, 0^i 1^j 2^k \mid i, j, k \in \mathbb{N} \text{ and either } i \neq j \text{ or } j \neq k \,\}.$$

Find a grammar G such that $L(G) = X$.

4.1.4 Notes

Traditionally, the meaning of grammars is defined using derivations, with parse trees being introduced subsequently. In contrast, we have no need for derivations, and find parse trees a much more intuitive way to define the meaning of grammars.

Pleasingly, parse trees are an instance of the trees introduced in Section 1.3. Thus the terminology, techniques and results of that section are applicable to them.

4.2 Isomorphism of Grammars

In this section, we study grammar isomorphism, i.e., the way in which grammars can have the same structure, even though they may have different variables.

4.2.1 Definition and Algorithm

Suppose G is the grammar with variables A and B, start variable A and productions:

$$\begin{aligned} &\mathsf{A} \to \mathsf{0A1} \mid \mathsf{B}, \\ &\mathsf{B} \to \% \mid \mathsf{2A}. \end{aligned}$$

And, suppose H is the grammar with variables B and A, start variable B and productions:

$$\begin{aligned} &\mathsf{B} \to \mathsf{0B1} \mid \mathsf{A}, \\ &\mathsf{A} \to \% \mid \mathsf{2B}. \end{aligned}$$

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:

$$\begin{aligned} &\mathsf{2} \to \mathsf{021} \mid \mathsf{A}, \\ &\mathsf{A} \to \% \mid \mathsf{22}. \end{aligned}$$

Then H can be formed from G by renaming the variables A and B to 2 and A, respectively. But we shouldn't consider G and H to be isomorphic, since the

symbol 2 is in both **alphabet** G and Q_H. 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., 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; and
- **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 the relation of being isomorphic is reflexive on **Gram**, symmetric and transitive, and that isomorphism implies having the same alphabet and equivalence.

There is an algorithm for finding an isomorphism from one grammar to another, if one exists, or reporting that there is no such isomorphism. It's similar to the algorithm for finding an isomorphism between finite automata.

The function **renameVariables** takes in a pair (G, f), where G is a grammar and f is a bijection from Q_G to a set of symbols with the property that **range** $f \cap$ **alphabet** $G = \emptyset$, and returns the grammar produced from G by renaming G's variables using the bijection f. The resulting grammar will be isomorphic to G.

The following function is a special case of **renameVariables**. The function **renameVariablesCanonically** $\in$ **Gram** $\rightarrow$ **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.

4.2.2 Isomorphism Finding/Checking in Forlan

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 `isomorphism` checks whether a relation on symbols is an isomorphism from one grammar to another. The function `findIsomorphism` tries to find an isomorphism from one grammar to another; it issues an error message if there isn't one. The function `isomorphic` checks whether two grammars are isomorphic. The function `renameVariables` issues an error message if the supplied relation isn't a bijection from the set of variables of the supplied grammar to some set; otherwise, it returns the result of **renameVariables**. And the function `renameVariablesCanonically` acts like **renameVariablesCanonically**.

Suppose the identifier `gram` of type `gram` is bound to the grammar with variables A and B, start variable A and productions:

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{B},$$
$$\mathsf{B} \to \% \mid \mathsf{2A}.$$

Suppose the identifier `gram'` of type `gram` is bound to the grammar with variables B and A, start variable B and productions:

$$\mathsf{B} \to \mathsf{0B1} \mid \mathsf{A},$$
$$\mathsf{A} \to \% \mid \mathsf{2B}.$$

And, suppose the identifier `gram''` of type `gram` is bound to the grammar with variables 2 and A, start variable 2 and productions:

$$\mathsf{2} \to \mathsf{021} \mid \mathsf{A},$$
$$\mathsf{A} \to \% \mid \mathsf{22}.$$

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
- val gram = Gram.input "";
@ {variables} B, C
@ {start variable} B
@ {productions} B -> AC; C -> <A>
@ .
val gram = - : gram
- SymSet.output("", Gram.alphabet gram);
A, <A>
```

```
val it = () : unit
- Gram.output("", Gram.renameVariablesCanonically gram);
{variables} <<A>>, <<B>> {start variable} <<A>>
{productions} <<A>> -> A<<B>>; <<B>> -> <A>
val it = () : unit
```

4.2.3 Notes

Considering grammar isomorphism is non-traditional, but relatively straightforward. We were led to doing so mostly because of the need to support variable renaming.

4.3 A Parsing Algorithm

In this section, we consider a simple, fairly inefficient parsing algorithm that works for all context-free grammars. In Section 4.6, we consider an efficient parsing method that works for grammars for languages of operators of varying precedences and associativities. Compilers courses cover efficient algorithms that work for various subsets of the context free grammars.

4.3.1 Algorithm

Suppose G is a grammar, $w \in \mathbf{Str}$ and $a \in \mathbf{Sym}$. We consider an algorithm for testing whether w is parsable from a using G. If $w \notin (Q_G \cup \mathbf{alphabet}\, G)^*$ or $a \notin Q_G \cup \mathbf{alphabet}\, w$, then the algorithm returns **false**. Otherwise, it proceeds as follows.

Let $A = Q_G \cup \mathbf{alphabet}\, w$ and $B = \{\, x \in \mathbf{Str} \mid x \text{ is a substring of } w \,\}$. The algorithm generates the least subset X of $A \times B$ such that:

(1) For all $a \in \mathbf{alphabet}\, w$, $(a, a) \in X$;

(2) For all $q \in Q_G$, if $q \to \% \in P_G$, then $(q, \%) \in X$; and

(3) 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 $i \in [1 : n]$, $(a_i, x_i) \in X$, 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 $w = \mathsf{0010}$ and $a = \mathsf{A} = s_G$. 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. To verify this, one must check that nothing new can be added to X using rule (3).

Back in the general case, we have these lemmas:

Lemma 4.3.1
For all $(b, x) \in X$, there is a $pt \in \mathbf{PT}$ such that

- *pt is valid for G,*
- **rootLabel** *pt* $= b$*, and*
- **yield** *pt* $= x$*.*

Lemma 4.3.2
For all $pt \in \mathbf{PT}$*, if*

- *pt is valid for G,*
- **rootLabel** *pt* $\in A$*, and*
- **yield** *pt* $\in B$*,*

then $(\mathbf{rootLabel}\, pt, \mathbf{yield}\, pt) \in X$*.*

Because of our lemmas, to determine if w is parsable from a, we just have to check whether $(a, w) \in X$. In the case of our example grammar, we have that $w = \mathsf{0010}$ is parsable from $a = \mathsf{A}$, since $(\mathsf{A}, \mathsf{0010}) \in X$. Hence $\mathsf{0010} \in L(G)$.

Note that any production whose right-hand side contains an element of $\mathbf{alphabet}\, G - \mathbf{alphabet}\, w$ won't affect the generation of X. Thus our algorithm ignores such productions.

Furthermore, our parsability algorithm actually generates X in a sequence of stages. At each point, it has subsets U and V of $A \times B$. U consists of the older elements of X, whereas V consists of the most recently added elements.

- First, it lets $U = \emptyset$ and sets V to be the union of $\{\, (a, a) \mid a \in \mathbf{alphabet}\, w \,\}$ and $\{\, (q, \%) \mid (q, \%) \in P_G \,\}$. It then enters its main loop.
- At a stage of the loop's iteration, it lets Y be $U \cup V$, and then lets Z be the set of all $(q, x_1 \cdots x_n)$ such that $n \geq 1$ and there are $a_1, \ldots, a_n \in A$ and $i \in [1 : n]$ such that
  - $q \to a_1 \cdots a_n \in P_G$,
  - $(a_i, x_i) \in V$,
  - for all $k \in [1 : n] - \{i\}$, $(a_k, x_k) \in Y$,
  - $x_1 \cdots x_n \in B$, and
  - $(q, x_1 \cdots x_n) \notin Y$.

  If $Z \neq \emptyset$, then it sets U to Y, and V to Z, and repeats; Otherwise, the result is Y.

We say that a parse tree *pt* is *a minimal parse* of a string w *from* a symbol a *using* a grammar G iff *pt* is valid for G, **rootLabel** *pt* $= a$ and **yield** *pt* $= w$, and there is no strictly smaller $pt' \in \mathbf{PT}$ such that pt' is valid for G, **rootLabel** $pt' = a$ and **yield** $pt' = w$.

We can convert our parsability algorithm into a parsing algorithm as follows. Given $w \in (Q_G \cup \textbf{alphabet}\, G)^*$ and $a \in (Q_G \cup \textbf{alphabet}\, w)$, we generate our set X as before, but we annotate each element (b, x) of X with a parse tree pt such that

- pt is valid for G,
- $\textbf{rootLabel}\, pt = b$, and
- $\textbf{yield}\, pt = x$,

Thus we can return the parse tree labeling (a, w), if this pair is in X, and indicate failure otherwise.

With a little 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. But we keep going as long as either new pairs are found, or smaller parse trees are found for existing pairs.

4.3.2 Parsing in Forlan

The Forlan module `Gram` defines the functions

```
val parsable              : gram -> sym * str -> bool
val generatedFromVariable : gram -> sym * str -> bool
val generated             : gram -> str -> bool
```

The function `parsable` tests whether a string w is parsable from a symbol a using a grammar G. The function `generatedFromVariable` tests whether a string w is generated from a variable q using a grammar G; it issues an error message if q isn't a variable of G. And the function `generated` tests whether a string w is generated by a grammar G.

`Gram` also includes:

```
val parse                     : gram -> sym * str -> pt
val parseAlphabetFromVariable : gram -> sym * str -> pt
val parseAlphabet             : gram -> str -> pt
```

The function `parse` tries to find a minimal parse of a string w from a symbol a using a grammar G; it issues an error message if $w \notin (Q_G \cup \textbf{alphabet}\, G)^*$, or $a \notin Q_G \cup \textbf{alphabet}\, w$, or such a parse doesn't exist. The function `parseAlphabetFromVariable` tries to find a minimal parse of a string $w \in (\textbf{alphabet}\, G)^*$ from a variable q using a grammar G; it issues an error message if $q \notin Q_G$, or $w \notin (\textbf{alphabet}\, G)^*$, or such a parse doesn't exist. And the function `parseAlphabet` tries to find a minimal parse of a string $w \in (\textbf{alphabet}\, G)^*$ from s_G using a grammar G; it issues an error message if $w \notin (\textbf{alphabet}\, G)^*$, or such a parse doesn't exist.

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 check whether some strings are generated by this grammar as follows:

```
- Gram.generated gram (Str.fromString "0010");
val it = true : bool
- Gram.generated gram (Str.fromString "0100");
val it = true : bool
- Gram.generated gram (Str.fromString "0101");
val it = false : bool
```

And we can try to find parses of some strings as follows:

```
- fun test s =
=       PT.output
=       ("",
=        Gram.parseAlphabet gram (Str.fromString s));
val test = fn : string -> unit
- test "0010";
A(C(D(0), 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
```

But we can also check parsability of strings containing variables, as well as try to find parses of such strings:

```
- Gram.parsable gram
= (Sym.fromString "A", Str.fromString "0D0C");
val it = true : bool
- PT.output
= ("",
=  Gram.parse gram
=  (Sym.fromString "A", Str.fromString "0D0C"));
A(C(D(0), D), D(B(0), C))
val it = () : unit
```

4.3.3 Notes

Our parsability and parsing algorithms are straightforward generalizations of the familiar algorithm for checking whether a grammar in Chomsky Normal Form generates a string.

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.

4.4.1 Definition and Algorithm

Suppose G is the grammar

$$\begin{aligned}&\mathsf{A} \to \mathsf{BB1},\\ &\mathsf{B} \to \mathsf{0} \mid \mathsf{A} \mid \mathsf{CD},\\ &\mathsf{C} \to \mathsf{12},\\ &\mathsf{D} \to \mathsf{1D2}.\end{aligned}$$

This grammar is odd for two, related, reasons. First D doesn't generate anything, i.e., there is no parse tree pt such that pt is valid for G, the root label of pt is D, and the yield of pt 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. But if we first removed D, and all productions involving it, then our objection to C could be simpler: there would be no parse tree pt such that pt is valid for G, the root label of pt is A, and C appears in the yield of pt.

This example leads us to the following definitions. Suppose G is a grammar. We say that a variable q of G is:

- *reachable in* G iff there is a $w \in \mathbf{Str}$ such that w is parsable from s_G using G, and $a \in \mathbf{alphabet}\, w$;
- *generating in* G iff there is a $w \in \mathbf{Str}$ such that q generates w using G, i.e., w is parsable from q using G, and $w \in (\mathbf{alphabet}\, G)^*$;
- *useful in* G iff q is both reachable and generating in G.

The reader should compare these definitions with the definitions given in Section 3.7 of reachable, live and useful states.

Also note that the standard definition of being useful is stronger than our definition: 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. For example, the variable C of our example grammar G is useful in our sense, but not useful in the standard

sense. But as we observed above, C will no longer be reachable (and thus useful) if all productions involving D are removed. In general, we have that, if all variables of a grammar are useful in our sense, that all variable of the grammar are useful in the standard sense.

Now, suppose H is the grammar

$$\mathsf{A} \to \% \mid \mathsf{0} \mid \mathsf{AA} \mid \mathsf{AAA}.$$

Here, we have that the productions A→AA and A→AAA are redundant, although only one of them can be removed:

Thus any use of $\mathsf{A} \to \mathsf{AA}$ in a parse tree can be replaced by uses of $\mathsf{A} \to \%$ and $\mathsf{A} \to \mathsf{AAA}$, and any use of $\mathsf{A} \to \mathsf{AAA}$ in a parse tree can be replaced by two uses of $\mathsf{A} \to \mathsf{AA}$.

This example leads us to the following definitions. Given a grammar G and a finite subset U of $\{\, (q, x) \mid q \in Q_G \text{ and } x \in \mathbf{Str} \,\}$, we write G/U for the grammar that is identical to G except that its set of productions is U. If G is a grammar and $(q, x) \in P_G$, we say that:

- (q, x) *is redundant in* G iff x is parsable from q using H, where $H = G/(P_G - \{(q, x)\})$; and
- (q, x) *is irredundant in* G iff (q, x) is not redundant in G.

Now we are able to say when a grammar is simplified. The reader should compare this definition with the definition in Section 3.7 of when a finite automaton is simplified. A grammar G is *simplified* iff either

- every variable of G is useful, and every production of G is irredundant; or
- $|Q_G| = 1$ and $P_G = \emptyset$.

The second case is necessary, because otherwise there would be no simplified grammar generating $\emptyset$.

Proposition 4.4.1
If G is a simplified grammar, then $\mathbf{alphabet}\, G = \mathbf{alphabet}(L(G))$.

Proof. Suppose $a \in \mathbf{alphabet}\, G$. We must show that $a \in \mathbf{alphabet}\, w$ for some $w \in L(G)$. We have that every variable of G is useful, and there are $q \in Q_G$ and $x \in \mathbf{Str}$ such that $(q, x) \in P_G$ and $a \in \mathbf{alphabet}\, x$. Thus x is parsable from q. Since every variable occurring in x is generating, we have that

q generates a string x' containing a. Since q is reachable, there is a string y such that y is parsable from s_G, and $q \in \mathbf{alphabet}\, y$. Since every variable occurring in y is generating, there is a string y' such that y' is parsable from s_G, and q is the only variable of $\mathbf{alphabet}\, y'$. Putting these facts together, we have that s_G generates a string w such that $a \in \mathbf{alphabet}\, w$, i.e., $a \in \mathbf{alphabet}\, w$ for some $w \in L(G)$. $\square$

Next, we give an algorithm for removing redundant productions. Given a grammar G, $q \in Q_G$ and $x \in \mathbf{Str}$, we say that (q, x) *is implicit in* G iff x is parsable from q using G.

Given a grammar G, we define a function $\mathbf{remRedun}_G \in \mathcal{P}\, P_G \times \mathcal{P}\, P_G \to \mathcal{P}\, P_G$ by well-founded recursion on the size of its second argument. For $U, V \subseteq P_G$, $\mathbf{remRedun}(U, V)$ proceeds as follows:

- If $V = \emptyset$, then it returns U.
- Otherwise, let v be the greatest element of $\{\, (q, x) \in V \mid$ there are no $p \in \mathbf{Sym}$ and $y \in \mathbf{Str}$ such that $(p, y) \in V$ and $|y| > |x| \,\}$, and $V' = V - \{v\}$. If v is implicit in $G/(U \cup V')$, then **remRedun** returns the result of evaluating $\mathbf{remRedun}(U, V')$. Otherwise, it returns the result of evaluating $\mathbf{remRedun}(U \cup \{v\}, V')$.

In general, there are multiple—incompatible—ways of removing redundant productions from a grammar. **remRedun** is defined so as to favor removing productions whose right-hand sides are longer; and among productions whose right-hand sides have equal length, to favor removing productions that are larger in our total ordering on productions.

Our algorithm for removing redundant productions of a grammar G returns $G/(\mathbf{remRedun}_G(\emptyset, P_G))$.

For example, if we run our algorithm for removing redundant productions on

$$\mathsf{A} \to \% \mid \mathsf{0} \mid \mathsf{AA} \mid \mathsf{AAA},$$

we obtain

$$\mathsf{A} \to \% \mid \mathsf{0} \mid \mathsf{AA}.$$

Our simplification algorithm for grammars proceeds as follows, given a grammar G.

- First, it determines which variables of G are generating. If s_G isn't one of these variables, then it returns the grammar with variable s_G and no productions.
- Next, it turns G into a grammar G' by deleting all non-generating variables, and deleting all productions involving such variables.

- Then, it determines which variables of G' are reachable.
- Next, it turns G' into a grammar G'' by deleting all non-reachable variables, and deleting all productions involving such variables.
- Finally, it removes redundant productions from G''.

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 production would never be deleted.
- Next, we form G'' from G' by deleting the variable C, yielding the grammar

$$\begin{aligned}&\mathsf{A} \to \mathsf{BB1},\\ &\mathsf{B} \to \mathsf{0} \mid \mathsf{A}.\end{aligned}$$

- Finally, we would remove redundant productions from G''. But G'' has no redundant productions, and so we are done.

We define a function $\mathbf{simplify} \in \mathbf{Gram} \to \mathbf{Gram}$ by: for all $G \in \mathbf{Gram}$, $\mathbf{simplify}\, G$ is the result of running the above algorithm on G.

Theorem 4.4.2
For all $G \in \mathbf{Gram}$*:*

(1) **simplify** G *is simplified;*

(2) **simplify** $G \approx G$*; and*

(3) $\mathbf{alphabet}(\mathbf{simplify}\, G) = \mathbf{alphabet}(L(G)) \subseteq \mathbf{alphabet}\, G$.

Our simplification function/algorithm **simplify** gives us an algorithm for testing whether a grammar is simplified: we apply **simplify** to it, and check that the resulting grammar is equal to the original one.

Our simplification algorithm gives us an algorithm for testing whether the language generated by a grammar G is empty. We first simplify G, calling the result H. We then test whether $P_H = \emptyset$. If the answer is "yes", clearly $L(G) = L(H) = \emptyset$. And if the answer is "no", then s_H is useful, and so H (and thus G) generates at least one string.

4.4.2 Simplification in Forlan

The Forlan module `Gram` defines the functions

```
val simplify   : gram -> gram
val simplified : gram -> bool
```

The function `simplify` corresponds to **simplify**, and `simplified` tests whether a grammar is simplified.

Suppose `gram` of type `gram` is bound to the grammar

$$\begin{aligned}&\mathsf{A} \to \mathsf{BB1},\\ &\mathsf{B} \to \mathsf{0} \mid \mathsf{A} \mid \mathsf{CD},\\ &\mathsf{C} \to \mathsf{12},\\ &\mathsf{D} \to \mathsf{1D2}.\end{aligned}$$

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
```

And, Suppose `gram''` of type `gram` is bound to the grammar

$$\mathsf{A} \to \% \mid \mathsf{0} \mid \mathsf{AA} \mid \mathsf{AAA} \mid \mathsf{AAAA}.$$

We can simplify our grammar as follows:

```
- val gram''' = Gram.simplify gram'';
val gram''' = - : gram
- Gram.output("", gram''');
{variables} A {start variable} A {productions} A -> % | 0 | AA
val it = () : unit
```

4.4.3 Hand-simplification Operations

Given a simplified grammar G, there are often ways we can hand-simplify the grammar further. Below are two examples:

- Suppose G has a variable q that is not s_G, and where no production having q as its left-hand side is *self-recursive*, i.e., has q as one of the symbols of its right-hand side. Let $x_1, \ldots, x_n$ be the right-hand sides of all of q's productions. ($n \geq 1$, as G is simplified.) Then we can form an equivalent grammar G' by deleting q and its productions from G, and transforming each remaining production $p \to y$ of G into all the productions from p that can be formed by substituting for each occurrence of q in y some choice of x_i.

  We refer to this operation as *eliminating q from G*.

- Suppose there is exactly one production of G involving s_G, where that production has the form $s_G \to q$, for some variable q of G. Then we can form an equivalent grammar G' by deleting s_G and $s_G \to q$ from G, and making q be the start variable of G'.

  We refer to this operation as *restarting G*.

The Forlan module `Gram` has functions corresponding to these two operations, first simplifying the supplied grammar:

```
val eliminateVariable : gram * sym -> gram
val restart           : gram -> gram
```

Both begin by simplifying the supplied grammar.

For instance, suppose `gram` is the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{B}, \\ &\mathsf{B} \to \mathsf{0} \mid \mathsf{C3C}, \\ &\mathsf{C} \to \mathsf{1B2} \mid \mathsf{2B1}. \end{aligned}$$

Then we can proceed as follows:

```
- val gram' = Gram.eliminateVariable(gram, Sym.fromString "C");
val gram' = - : gram
- Gram.output("", gram');
{variables} A, B {start variable} A
{productions}
A -> B; B -> 0 | 1B231B2 | 1B232B1 | 2B131B2 | 2B132B1
val it = () : unit
- val gram'' = Gram.restart gram;
val gram'' = - : gram
- Gram.output("", gram'');
{variables} B, C {start variable} B
{productions} B -> 0 | C3C; C -> 1B2 | 2B1
val it = () : unit
```

4.4.4 Notes

As described above, our definition of useless variable is weaker than the standard one. However, whenever every variable of a grammar is useful in our sense, it follows that every variable of the grammar is useful in the standard sense. Furthermore, our algorithm for removing useless variables is the standard one. Requiring that simplified grammars have no redundant productions is natural, although non-standard, and our algorithm for removing redundant productions is straightforward. The algorithms for restarting and eliminating variables from grammars are non-standard but obvious.

4.5 Proving the Correctness of Grammars

In this section, we consider techniques for proving the correctness of grammars, i.e., for proving that grammars generate the languages we want them to.

4.5.1 Preliminaries

Suppose G is a grammar and $a \in Q_G \cup \mathbf{alphabet}\, G$. Then

$$\Pi_{G,a} = \{\, w \in (\mathbf{alphabet}\, G)^* \mid w \text{ is parsable from } a \text{ using } G \,\}.$$

I.e., $\Pi_{G,a}$ is all strings over $\mathbf{alphabet}\, G$ that are the yields of valid parse trees for G whose root labels are a If it's clear which grammar we are talking about, we often abbreviate $\Pi_{G,a}$ to Π_a. Clearly, $\Pi_{G,s_G} = L(G)$.

For example, if G is the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1}$$

then $\Pi_{\mathsf{0}} = \{\mathsf{0}\}$, $\Pi_{\mathsf{1}} = \{\mathsf{1}\}$ and $\Pi_{\mathsf{A}} = \{\, \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \,\} = L(G)$.

Proposition 4.5.1
Suppose G is a grammar.

(1) For all $a \in \mathbf{alphabet}\, G$, $a \in \Pi_{G,a}$.

(2) For all $q \in Q_G$, if $q \to \% \in P_G$, then $\% \in \Pi_{G,q}$.

(3) For all $q \in Q_G$, $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in \mathbf{Sym}$ and $w_1, \ldots, w_n \in \mathbf{Str}$, if $q \to a_1 \cdots a_n \in P_G$ and $w_1 \in \Pi_{G,a_1}, \ldots, w_n \in \Pi_{G,a_n}$, then $w_1 \cdots w_n \in \Pi_{G,q}$.

Our main example will be the grammar G:

$$\begin{aligned}
&\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},\\
&\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},\\
&\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC}.
\end{aligned}$$

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

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

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

$$\begin{aligned} X &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = 0 \,\}, \\ Y &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = 1 \text{ and,} \\ &\qquad \text{for all proper prefixes } v \text{ of } w, \mathbf{diff}\, v \leq 0 \,\}, \text{ and} \\ Z &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = -1 \text{ and,} \\ &\qquad \text{for all proper prefixes } v \text{ of } w, \mathbf{diff}\, v \geq 0 \,\}. \end{aligned}$$

We will prove that $L(G) = \Pi_{G,\mathsf{A}} = X$, $\Pi_{G,\mathsf{B}} = Y$ and $\Pi_{G,\mathsf{C}} = Z$.

Lemma 4.5.2
Suppose $x \in \{0,1\}^*$.

(1) If $\mathbf{diff}\, x \geq 1$, then $x = yz$ for some $y, z \in \{0,1\}^*$ such that $y \in Y$ and $\mathbf{diff}\, z = \mathbf{diff}\, x - 1$.

(2) If $\mathbf{diff}\, x \leq -1$, then $x = yz$ for some $y, z \in \{0,1\}^*$ such that $y \in Z$ and $\mathbf{diff}\, z = \mathbf{diff}\, x + 1$.

Proof. We show the proof of (1), the proof of (2) being similar.

Let $y \in \{0,1\}^*$ be the shortest prefix of x such that $\mathbf{diff}\, y \geq 1$, and let $z \in \{0,1\}^*$ be such that $x = yz$.

Because $\mathbf{diff}\, y \geq 1$, we have that $y \neq \%$. Thus $y = y'a$ for some $y' \in \{0,1\}^*$ and $a \in \{0,1\}$. By the definition of y, we have that $\mathbf{diff}\, y' \leq 0$. Suppose, toward a contradiction, that $a = 0$. Since $\mathbf{diff}\, y' + -1 = \mathbf{diff}\, y \geq 1$, we have that $\mathbf{diff}\, y' \geq 2$, contradicting the definition of y. Thus $a = 1$, so that $y = y'1$.

Because $\mathbf{diff}\, y' + 1 = \mathbf{diff}\, y \geq 1$, we have that $\mathbf{diff}\, y' \geq 0$. Thus $\mathbf{diff}\, y' = 0$, so that $\mathbf{diff}\, y = \mathbf{diff}\, y' + 1 = 1$. By the definition of y, every prefix of y' has a diff that is ≤ 0. Thus $y \in Y$.

Finally, because $\mathbf{diff}\, x = \mathbf{diff}\, y + \mathbf{diff}\, z = 1 + \mathbf{diff}\, z$ we have that $\mathbf{diff}\, z = \mathbf{diff}\, x - 1$. □

4.5.2 Proving that Enough is Generated

First we study techniques for showing that everything we want a grammar to generate is really generated.

Since $X, Y, Z \subseteq \{0,1\}^*$, to prove that $X \subseteq \Pi_{G,\mathsf{A}}$, $Y \subseteq \Pi_{G,\mathsf{B}}$ and $Z \subseteq \Pi_{G,\mathsf{C}}$, it will suffice to use strong string induction to show that, for all $w \in \{0,1\}^*$:

(A) if $w \in X$, then $w \in \Pi_{G,\mathsf{A}}$;

(B) if $w \in Y$, then $w \in \Pi_{G,\mathsf{B}}$; and

(C) if $w \in Z$, then $w \in \Pi_{G,\mathsf{C}}$.

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

(A) if $x \in X$, then $x \in \Pi_{\mathsf{A}}$;

(B) if $x \in Y$, then $x \in \Pi_{\mathsf{B}}$; and

(C) if $x \in Z$, then $x \in \Pi_{\mathsf{C}}$.

We must prove that:

(A) if $w \in X$, then $w \in \Pi_{\mathsf{A}}$;

(B) if $w \in Y$, then $w \in \Pi_{\mathsf{B}}$; and

(C) if $w \in Z$, then $w \in \Pi_{\mathsf{C}}$.

(A) Suppose $w \in X$. We must show that $w \in \Pi_{\mathsf{A}}$. There are three cases to consider.

- Suppose $w = \%$. Because $\mathsf{A} \to \% \in P$, Proposition 4.5.1(2) tells us that $w = \% \in \Pi_{\mathsf{A}}$.
- Suppose $w = \mathsf{0}x$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Because $-1 + \mathbf{diff}\, x = \mathbf{diff}\, w = 0$, we have that $\mathbf{diff}\, x = 1$. Thus, by Lemma 4.5.2(1), we have that $x = yz$, for some $y, z \in \{\mathsf{0},\mathsf{1}\}^*$ such that $y \in Y$ and $\mathbf{diff}\, z = \mathbf{diff}\, x - 1 = 1 - 1 = 0$. Thus $w = \mathsf{0}yz$, $y \in Y$ and $z \in X$. By Proposition 4.5.1(1), we have $\mathsf{0} \in \Pi_{\mathsf{0}}$. Because $y \in Y$ and $z \in X$ are proper substrings of w, parts (B) and (A) of the inductive hypothesis tell us that $y \in \Pi_{\mathsf{B}}$ and $z \in \Pi_{\mathsf{A}}$. Thus, because $\mathsf{A} \to \mathsf{0BA} \in P$, Proposition 4.5.1(3) tells us that $w = \mathsf{0}yz \in \Pi_{\mathsf{A}}$.
- Suppose $w = \mathsf{1}x$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. The proof is analogous to the preceding case.

(B) Suppose $w \in Y$. We must show that $w \in \Pi_{\mathsf{B}}$. Because $\mathbf{diff}\, w = 1$, there are two cases to consider.

- Suppose $w = \mathsf{1}x$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Because all proper prefixes of w have diffs ≤ 0, we have that $x = \%$, so that $w = \mathsf{1}$. Since $\mathsf{B} \to \mathsf{1} \in P$, we have that $w = \mathsf{1} \in \Pi_{\mathsf{B}}$.
- Suppose $w = \mathsf{0}x$, for some $x \in \{\mathsf{0},\mathsf{1}\}^*$. Thus $\mathbf{diff}\, x = 2$. Because $\mathbf{diff}\, x \geq 1$, by Lemma 4.5.2(1), we have that $x = yz$, for some $y, z \in \{\mathsf{0},\mathsf{1}\}^*$ such that $y \in Y$ and $\mathbf{diff}\, z = \mathbf{diff}\, x - 1 = 2 - 1 = 1$. Hence $w = \mathsf{0}yz$. To finish the proof that $z \in Y$, suppose v is a

proper prefix of z. Thus $\mathsf{0}yv$ is a proper prefix of w. Since $w \in Y$, it follows that $\mathbf{diff}\, v = \mathbf{diff}(\mathsf{0}yv) \leq 0$, as required. Since $y, z \in Y$, part (B) of the inductive hypothesis tell us that $y, z \in \Pi_{\mathsf{B}}$. Thus, because $\mathsf{B} \to \mathsf{0BB} \in P$ we have that $w = \mathsf{0}yz \in \Pi_{\mathsf{B}}$.

(C) Suppose $w \in Z$. We must show that $w \in \Pi_{\mathsf{C}}$. The proof is analogous to the proof of part (B).

Suppose H is the grammar

$$\mathsf{A} \to \mathsf{B} \mid \mathsf{0A3}, \qquad \mathsf{B} \to \% \mid \mathsf{1B2},$$

and let

$$X = \{\, \mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \mid n, m \in \mathbb{N} \,\}, \qquad Y = \{\, \mathsf{1}^m\mathsf{2}^m \mid m \in \mathbb{N} \,\}.$$

We can prove that $X \subseteq \Pi_{H,\mathsf{A}} = L(H)$ and $Y \subseteq \Pi_{H,\mathsf{B}}$ using the above technique, but the production $\mathsf{A} \to \mathsf{B}$, which is called a *unit production* because its right side is a single variable, makes part (A) tricky. In Section 3.8, when considering techniques for showing the correctness of finite automata, we ran into a similar problem having to do with %-transitions.

If $w = \mathsf{0}^0\mathsf{1}^m\mathsf{2}^m\mathsf{3}^0 = \mathsf{1}^m\mathsf{2}^m \in Y$, we would like to use part (B) of the inductive hypothesis to conclude $w \in \Pi_{\mathsf{B}}$, and then use the fact that $\mathsf{A} \to \mathsf{B} \in P$ to conclude that $w \in \Pi_{\mathsf{A}}$. But w is not a proper substring of itself, and so the inductive hypothesis in not applicable. Instead, we must split into cases $m = 0$ and $m \geq 1$, using $\mathsf{A} \to \mathsf{B}$ and $\mathsf{B} \to \%$, in the first case, and $\mathsf{A} \to \mathsf{B}$ and $\mathsf{B} \to \mathsf{1B2}$, as well as the inductive hypothesis on $\mathsf{1}^{m-1}\mathsf{2}^{m-1} \in Y$, in the second case.

Because there are no productions from B back to A, we could also first use strong string induction to prove that, for all $w \in \{\mathsf{0}, \mathsf{1}\}^*$,

(B) if $w \in Y$, then $w \in \Pi_{\mathsf{B}}$,

and then use the result of this induction along with strong string induction to prove that for all $w \in \{\mathsf{0}, \mathsf{1}\}^*$,

(A) if $w \in X$, then $w \in \Pi_{\mathsf{A}}$.

This works whenever two parts of a grammar are not mutually recursive.

With this grammar, we could also first use mathematical induction to prove that, for all $m \in \mathbb{N}$, $\mathsf{1}^m\mathsf{2}^m \in \Pi_{\mathsf{B}}$, and then use the result of this induction to prove, by mathematical induction on n, that for all $n, m \in \mathbb{N}$, $\mathsf{0}^n\mathsf{1}^m\mathsf{2}^m\mathsf{3}^n \in \Pi_{\mathsf{A}}$.

Note that %-productions, i.e., productions of the form $q \to \%$, can cause similar problems to those caused by unit productions. E.g., if we have the productions

$$\mathsf{A} \to \mathsf{BC} \quad \text{and} \quad \mathsf{B} \to \%,$$

then $\mathsf{A} \to \mathsf{BC}$ behaves like a unit production.

4.5.3 Proving that Everything Generated is Wanted

When proving that everything generated by a grammar is wanted, we could sometimes use strong induction, simply reversing the implications used when proving that enough is generated. But this approach fails for grammars with unit productions, where we would have to resort to an induction on parse trees.

In Section 3.8, when considering techniques for showing the correctness of finite automata, we ran into a similar problem having to do with %-transitions, and this led to our introducing the Principle of Induction on Λ. Here, we introduce an induction principle called Induction on Π.

Theorem 4.5.3 (Principle of Induction on Π)
*Suppose G is a grammar, $P_q(w)$ is a property of a string $w \in \Pi_{G,q}$, for all $q \in Q_G$, and $P_a(w)$, for $a \in$ **alphabet** G, says "$w = a$". If*

(1) for all $q \in Q_G$, if $q \to \% \in P_G$, then $P_q(\%)$, and

*(2) for all $q \in Q_G$, $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in Q_G \cup$ **alphabet** G, and $w_1 \in \Pi_{G,a_1}, \ldots, w_n \in \Pi_{G,a_n}$, if $q \to a_1 \cdots a_n \in P_G$ and (†) $P_{a_1}(w_1), \ldots, P_{a_n}(w_n)$, then $P_q(w_1 \cdots w_n)$,*

then

$$\text{for all } q \in Q_G, \text{ for all } w \in \Pi_{G,q},\ P_q(w).$$

We refer to (†) as the inductive hypothesis.

Proof. It suffices to show that, for all $pt \in \mathbf{PT}$, for all $q \in Q_G$ and $w \in (\mathbf{alphabet}\, G)^*$, if pt is valid for G, $\mathbf{rootLabel}\, pt = q$ and $\mathbf{yield}\, pt = w$, then $P_q(w)$. We prove this using the principle of induction on parse trees. □

When proving part (2), we can make use of the fact that, for $a_i \in$ **alphabet** G, $\Pi_{a_i} = \{a_i\}$, so that $w_i \in \Pi_{a_i}$ will be a_i. Hence it will be unnecessary to assume that $P_{a_i}(a_i)$, since this says "$a_i = a_i$", and so is always true.

Consider, again, our main example grammar G:

$$\begin{aligned}
&\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},\\
&\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},\\
&\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC}.
\end{aligned}$$

Let

$$\begin{aligned}
X &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = 0 \,\},\\
Y &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = 1 \text{ and,}\\
&\qquad \text{for all proper prefixes } v \text{ of } w, \mathbf{diff}\, v \leq 0 \,\},\\
Z &= \{\, w \in \{0,1\}^* \mid \mathbf{diff}\, w = -1 \text{ and,}\\
&\qquad \text{for all proper prefixes } v \text{ of } w, \mathbf{diff}\, v \geq 0 \,\}.
\end{aligned}$$

We have already proven that $X \subseteq \Pi_{\mathsf{A}} = L(G)$, $Y \subseteq \Pi_{\mathsf{B}}$ and $Z \subseteq \Pi_{\mathsf{C}}$. To complete the proof that $L(G) = \Pi_{\mathsf{A}} = X$, $\Pi_{\mathsf{B}} = Y$ and $\Pi_{\mathsf{C}} = Z$, we will use induction on Π to prove that $\Pi_{\mathsf{A}} \subseteq X$, $\Pi_{\mathsf{B}} \subseteq Y$ and $\Pi_{\mathsf{C}} \subseteq Z$.

We use induction on Π to show that:

(A) for all $w \in \Pi_{\mathsf{A}}$, $w \in X$;

(B) for all $w \in \Pi_{\mathsf{B}}$, $w \in Y$; and

(C) for all $w \in \Pi_{\mathsf{C}}$, $w \in Z$.

Formally, this means that we let the properties $P_{\mathsf{A}}(w)$, $P_{\mathsf{B}}(w)$ and $P_{\mathsf{C}}(w)$ be "$w \in X$", "$w \in Y$" and "$w \in Z$", respectively, and then use the induction principle to prove that, for all $q \in Q_G$, for all $w \in \Pi_q$, $P_q(w)$. But we will actually work more informally.

There are seven productions to consider.

($\mathsf{A} \to \%$) We must show that $\% \in X$ (as "$w \in X$" is the property of part (A)). And this holds since $\mathbf{diff}\,\% = 0$.

($\mathsf{A} \to \mathsf{0BA}$) Suppose $w_1 \in \Pi_{\mathsf{B}}$ and $w_2 \in \Pi_{\mathsf{A}}$ (as $\mathsf{0BA}$ is the right-side of the production, and $\mathsf{0}$ is in G's alphabet), and assume the inductive hypothesis, $w_1 \in Y$ (as this is the property of part (B)) and $w_2 \in X$ (as this is the property of part (A)). We must show that $\mathsf{0}w_1w_2 \in X$, as the production shows that $\mathsf{0}w_1w_2 \in \Pi_{\mathsf{A}}$. Because $w_1 \in Y$ and $w_2 \in X$, we have that $\mathbf{diff}\,w_1 = 1$ and $\mathbf{diff}\,w_2 = 0$. Thus $\mathbf{diff}(\mathsf{0}w_1w_2) = -1+1+0 = 0$, showing that $\mathsf{0}w_1w_2 \in X$.

($\mathsf{B} \to \mathsf{0BB}$) Suppose $w_1, w_2 \in \Pi_{\mathsf{B}}$, and assume the inductive hypothesis, $w_1, w_2 \in Y$. Thus w_1 and w_2 are nonempty. We must show that $\mathsf{0}w_1w_2 \in Y$. Clearly, $\mathbf{diff}(\mathsf{0}w_1w_2) = -1 + 1 + 1 = 1$. So, suppose v is a proper prefix of $\mathsf{0}w_1w_2$. We must show that $\mathbf{diff}\,v \leq 0$. There are three cases to consider.

- Suppose $v = \%$. Then $\mathbf{diff}\,v = 0 \leq 0$.
- Suppose $v = \mathsf{0}u$, for a proper prefix u of w_1. Because $w_1 \in Y$, we have that $\mathbf{diff}\,u \leq 0$. Thus $\mathbf{diff}\,v = -1 + \mathbf{diff}\,u \leq -1 + 0 \leq 0$.
- Suppose $v = \mathsf{0}w_1u$, for a proper prefix u of w_2. Because $w_2 \in Y$, we have that $\mathbf{diff}\,u \leq 0$. Thus $\mathbf{diff}\,v = -1 + 1 + \mathbf{diff}\,u = \mathbf{diff}\,u \leq 0$.

The remaining productions are handled similarly.

Exercise 4.5.4
Let

$$X = \{\, \mathsf{0}^i\mathsf{1}^j\mathsf{2}^k\mathsf{3}^l \mid i, j, k, l \in \mathbb{N} \text{ and } i + j = k + l \,\}.$$

Find a grammar G such that $L(G) = X$, and prove that your answer is correct.

Exercise 4.5.5
Let

$$X = \{\, 0^i 1^j 2^k \mid i, j, k \in \mathbb{N} \text{ and either } i \neq j \text{ or } j \neq k \,\}.$$

Find a grammar G such that $L(G) = X$, and prove that your answer is correct.

4.5.4 Notes

Books on formal language theory typically give short shrift to the proof of correctness of grammars, carrying out one or two correctness proofs using induction on the length of strings. In contrast, we have introduced and applied elegant techniques for proving the correctness of grammars. Of particular note is our principle of induction on Π.

4.6 Ambiguity of Grammars

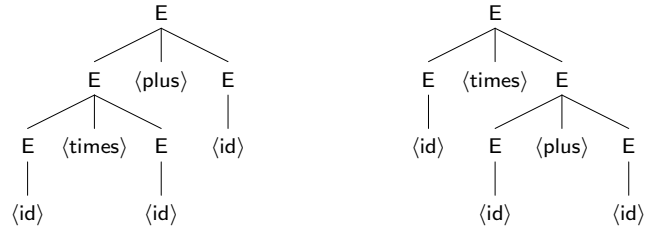
In this section, we say what it means for a grammar to be ambiguous. We also give a straightforward method for disambiguating grammars for languages with operators of various precedences and associativities, and consider an efficient parsing algorithm for such disambiguated grammars.

4.6.1 Definition

Suppose G is our grammar of arithmetic expressions:

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Unfortunately, there are multiple ways of parsing $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ 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*.

Let G be the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1A} \mid \mathsf{1A0A}$$

which generates all elements of $\{\mathsf{0},\mathsf{1}\}^*$ with a **diff** of 0, for the **diff** function such that $\mathbf{diff}\,\mathsf{0} = -1$ and $\mathbf{diff}\,\mathsf{1} = 1$. It is ambiguous as, e.g., $\mathsf{0101}$ can be parsed as $\mathsf{0\%1(01)}$ or $\mathsf{0(10)1\%}$. But in Section 4.5, we saw another grammar, H, for this language:

$$\begin{aligned}&\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},\\ &\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},\\ &\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC},\end{aligned}$$

which turns out to be unambiguous. The reason is that Π_{B} is all elements of $\{\mathsf{0},\mathsf{1}\}^*$ with a **diff** of 1, but with no proper prefixes with positive **diff**'s, and Π_{C} has the corresponding property for 0/negative.

Exercise 4.6.1
Prove that $L(G) = L(H)$.

Exercise 4.6.2
Prove that H is unambiguous.

4.6.2 Disambiguating Grammars of Operators

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, and this technique works for many commonly occurring grammars involving operators of various precedences and associativities.

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; and
- whether multiplication and addition are left or right associative.

As usual, we'll make multiplication have higher precedence than addition, and let addition and multiplication 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:

$$\begin{aligned}&\mathsf{E} \to \mathsf{T} \mid \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\\ &\mathsf{T} \to \mathsf{F} \mid \mathsf{T}\langle\mathsf{times}\rangle\mathsf{T},\\ &\mathsf{F} \to \langle\mathsf{id}\rangle \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle.\end{aligned}$$

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 $\mathsf{E} \to \mathsf{T}$, 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

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},$$
$$\mathsf{T} \to \mathsf{T}\langle\mathsf{times}\rangle\mathsf{T},$$

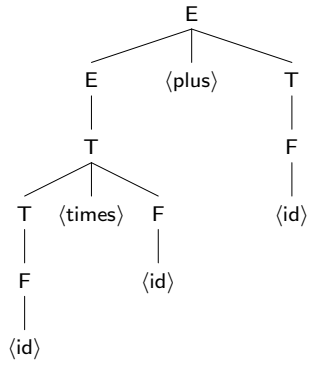
turning one of the E's into T, and one of the T's into F. To make our operators be left associative, we must use *left recursion*, changing the second E to T, and the second T to F; right associativity would result from making the opposite choices, i.e., using *right recursion*.

Thus, our unambiguous grammar of arithmetic expressions is

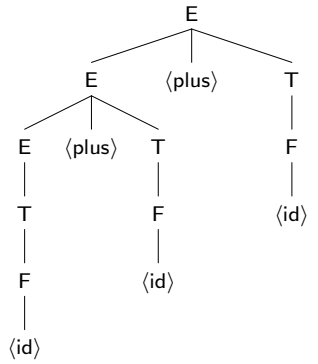
$$\mathsf{E} \to \mathsf{T} \mid \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{T},$$
$$\mathsf{T} \to \mathsf{F} \mid \mathsf{T}\langle\mathsf{times}\rangle\mathsf{F},$$
$$\mathsf{F} \to \langle\mathsf{id}\rangle \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle.$$

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



4.6.3 Top-down Parsing

Top-down parsing is a simple and efficient parsing method for unambiguous grammars of operators like

$$\begin{aligned}
&\mathsf{E} \to \mathsf{T} \mid \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{T},\\
&\mathsf{T} \to \mathsf{F} \mid \mathsf{T}\langle\mathsf{times}\rangle\mathsf{F},\\
&\mathsf{F} \to \langle\mathsf{id}\rangle \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle.
\end{aligned}$$

Let $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ be all of the parse trees that are valid for our grammar, have yields containing no variables, and whose root labels are E, T and F, respectively. Because this grammar has three mutually recursive variables, we will need three mutually recursive parsing functions,

$$\begin{aligned}
&\mathbf{parE} \in \mathbf{Str} \to \mathbf{Option}(\mathcal{E} \times \mathbf{Str}),\\
&\mathbf{parT} \in \mathbf{Str} \to \mathbf{Option}(\mathcal{T} \times \mathbf{Str}),\\
&\mathbf{parF} \in \mathbf{Str} \to \mathbf{Option}(\mathcal{F} \times \mathbf{Str}),
\end{aligned}$$

which attempt to parse an element pt of $\mathcal{E}$, $\mathcal{T}$ or $\mathcal{F}$ out of a string w, returning **none** to indicate failure, and $\mathbf{some}(pt, y)$, where y is the remainder of w, otherwise.

Although most programming languages support mutual recursion, in this book, we haven't formally justified well-founded mutual recursion. Instead, we can work with a single recursive function with domain $\{0, 1, 2\} \times \mathbf{Str}$, where the 0, 1 or 2 indicates whether it's **parE**, **parT** or **parF**, respectively, that is being called. The range of this function will be $\mathbf{Option}(\mathcal{U} \times \mathbf{Str})$, where $\mathcal{U}$ is the union of three disjoint sets: $\{0\} \times \mathcal{E}$, $\{1\} \times \mathcal{T}$ and $\{2\} \times \mathcal{F}$. E.g., when the function is called with $(0, w)$, it will either return **none** or $\mathbf{some}((0, pt), z)$, where $pt \in \mathcal{E}$ and $x \in \mathbf{Str}$. But to keep the notation simple, below, we'll assume the parsing functions can call each other.

The well-founded ordering we are using allows:

- **parE** to call **parT** with strings that are no longer than its argument;

- **parT** to call **parF** with strings that are no longer than its argument; and
- **parF** to call **parE** with strings that are strictly shorter than its argument.

When called with a string w, **parE** is supposed to determine whether there is a prefix x of w that is the yield of an element of $\mathcal{E}$. If there is such an x, then it finds the *longest* prefix x of w with this property, and returns $\mathbf{some}(pt, y)$, where pt is the element of $\mathcal{E}$ whose yield is x, and y is such that $w = xy$. Otherwise, it returns **none**. **parT** and **parF** have similar specifications.

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating $\mathbf{parT}\, w$. If this results in **none**, it returns **none**. Otherwise, it results in $\mathbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \mathbf{Str}$, in which case **parE** returns $\mathbf{parELoop}(E(pt), x)$, where $\mathbf{parELoop} \in \mathcal{E} \times \mathbf{Str} \to \mathbf{Option}(\mathcal{E} \times \mathbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \mathbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle \mathsf{plus} \rangle y$ for some y, then **parELoop** evaluates $\mathbf{parT}\, y$.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in $\mathbf{some}(pt', z)$ for some $pt' \in \mathcal{T}$ and $z \in \mathbf{Str}$, and **parELoop** returns $\mathbf{parELoop}(E(pt, \langle \mathsf{plus} \rangle, pt'), z)$.
- Otherwise, **parELoop** returns $\mathbf{some}(pt, x)$.

The function **parT** operates analogously.

Given a string w, **parF** proceeds as follows.

- If $w = \langle \mathsf{id} \rangle x$ for some x, then it returns $\mathbf{some}(F(\langle \mathsf{id} \rangle), x)$.
- Otherwise, if $w = \langle \mathsf{openPar} \rangle x$, then **parF** evaluates $\mathbf{parE}\, x$.
  - If this results in **none**, it returns **none**.
  - Otherwise, this results in $\mathbf{some}(pt, y)$ for some $pt \in \mathcal{E}$ and $y \in \mathbf{Str}$.
    - If $y = \langle \mathsf{closPar} \rangle z$ for some z, then **parF** returns
      $$\mathbf{some}(F(\langle \mathsf{openPar} \rangle, pt, \langle \mathsf{closPar} \rangle), z).$$
    - Otherwise, **parF** returns **none**.
- Otherwise **parF** returns **none**.

Given a string w to parse, the algorithm evaluates $\mathbf{parE}\, w$. If the result of this evaluation is:

- **none**, then the algorithm reports failure;
- $\mathbf{some}(pt, \%)$, then the algorithm returns pt;
- $\mathbf{some}(pt, y)$, where $y \neq \%$, then the algorithm reports failure, because not all of the input could be parsed.

4.6.4 Notes

The standard approach to doing top-down parsing in the presence of left recursive productions is to first translate the left recursion to right recursion, and then restructure the parse trees produced by the parser. In contrast, we showed a direct approach to handling left recursion that works for grammars of operators.

4.7 Closure Properties of Context-free Languages

In this section, we define union, concatenation, closure, reversal, alphabet-renaming and prefix-closure operations/algorithms on grammars. As a result, we will have that the context-free languages are closed under union, concatenation, closure, reversal, alphabet-renaming and prefix-, suffix- and substring-closure.

In Section 4.10, we will see that the context-free languages aren't closed under intersection, complementation and set difference. But we are able to define operations/algorithms for:

- intersecting a grammar and an empty-string finite automaton; and
- subtracting a deterministic finite automaton from a grammar.

Thus, if L_1 is a context-free language, and L_2 is a regular language, we will have that $L_1 \cap L_2$ and $L_1 - L_2$ are context-free.

4.7.1 Operations on Grammars

First, we consider some basic grammars and operations on grammars. The grammar, **emptyStr**, with variable A and production $\mathsf{A} \to \%$ generates the language $\{\%\}$. The grammar, **emptySet**, with variable A and no productions generates the language $\emptyset$. If w is a string, then the grammar with variable A and production $\mathsf{A} \to w$ generates the language $\{w\}$. Actually, we must be careful to chose a variable that doesn't occur in w. We can do that by adding as many nested `<` and `>` around `A` as needed (`A`, `<A>`, `<<A>>`, etc.). This defines functions **strToGram** $\in$ **Str** $\to$ **Gram** and **symToGram** $\in$ **Sym** $\to$ **Gram**.

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

$$q \to s_{G_1} \mid s_{G_2}.$$

For the above to be valid, we need to know that:

- $Q_{G_1} \cap Q_{G_2} = \emptyset$ and $q \notin Q_{G_1} \cup Q_{G_2}$; and
- **alphabet** $G_1 \cap Q_{G_2} = \emptyset$, **alphabet** $G_2 \cap Q_{G_1} = \emptyset$ and $q \notin$ **alphabet** $G_1 \cup$ **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. This gives us a function **union** $\in$ **Gram** $\times$ **Gram** $\to$ **Gram**.

We do something similar when defining the other closure operations. In what follows, though, we'll ignore this issue, so as to keep things simple.

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

$$q \to s_{G_1} s_{G_2}.$$

This gives us a function **concat** $\in$ **Gram** $\times$ **Gram** $\to$ **Gram**.

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

$$q \to \% \mid s_G q.$$

This gives us a function **closure** $\in$ **Gram** $\to$ **Gram**.

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. This gives a function **rev** $\in$ **Gram** $\to$ **Gram**.

Given a grammar G and a bijection f from a set of symbols that is a superset of **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 **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. Let $X = \{ (G, f) \mid G \in \textbf{Gram}$ and f is a bijection from a set of symbols that is a superset of **alphabet** G to some set of symbols $\}$. Then the above definition gives us a function **renameAlphabet** $\in X \to$ **Gram**.

From Section 3.12, we know that if we can define a prefix-closure operation on grammmars, then we can obtain suffix-closure and substring-closure operations on grammars from the prefix-closure and grammar reversal operations.

So how can we turn a grammar G into a grammar H such that $L(H) = L(G)^P$? We begin by simplifying G, producing grammar G'. Thus all of the variables of G' will be useful, unless G' has a single variable and no productions. Now, we form the grammmar H from G', as follows. We make a copy of G', renaming each variable q to $\langle 1, q \rangle$. (Actually, we may have to rename variables to avoid clashes with alphabet symbols.) Next, for each alphabet symbol a, we introduce a new variable $\langle 2, a \rangle$, along with productions $\langle 2, a \rangle \to \% \mid a$. Next, for each variable q of G', we add a new variable $\langle 2, q \rangle$ that generates all prefixes of what q generated in G'. Suppose we are given a production $q \to a_1 a_2 \cdots a_n$ of G'. If $n = 0$, then we replace it with the production $\langle 2, q \rangle \to \%$. Otherwise, we

replace it with the productions

$$\langle 2, q\rangle \to \langle 2, a_1\rangle \mid f(a_1)\langle 2, a_2\rangle \mid \cdots \mid f(a_1)\, f(a_2) \cdots \langle 2, a_n\rangle,$$

where $f(a) = a$, if $a \in \mathbf{alphabet}\, G'$, and $f(a) = \langle 1, a\rangle$, if $a \in Q_{G'}$. It's crucial that G' is simplified; otherwise productions with useless symbols would be turned into productions that generated strings. Finally, the start variable of H is $\langle 2, s_{G'}\rangle$.

The above definition gives a function **prefix** $\in$ **Gram**$\to$**Gram**. For example, the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1}$$

is turned into the grammar

$$\begin{aligned}
&\langle 2, \mathsf{A}\rangle \to \% \mid \langle 2, \mathsf{0}\rangle \mid \mathsf{0}\langle 2, \mathsf{A}\rangle \mid \mathsf{0}\langle 1, A\rangle\langle 2, \mathsf{1}\rangle,\\
&\langle 1, A\rangle \to \% \mid \mathsf{0}\langle 1, \mathsf{A}\rangle\mathsf{1},\\
&\langle 2, \mathsf{0}\rangle \to \% \mid \mathsf{0},\\
&\langle 2, \mathsf{1}\rangle \to \% \mid \mathsf{1}.
\end{aligned}$$

We now consider an algorithm for intersecting a grammar G with an EFA M, resulting in **simplify** H, where the grammar H is defined as follows. For all $p \in Q_G$ and $q, r \in Q_M$, H has a variable $\langle p, q, r\rangle$ that generates

$$\{\, w \in (\mathbf{alphabet}\, G)^* \mid w \in \Pi_{G,p} \text{ and } r \in \Delta_M(\{q\}, w)\,\}.$$

The remaining variable of H is A, which is its start variable.

For each $r \in A_M$, H has a production

$$\mathsf{A} \to \langle s_G, s_M, r\rangle.$$

And for each %-production $p \to \%$ of G and $q, r \in Q_M$, if $r \in \Delta_M(\{q\}, \%)$, then H will have the production

$$\langle p, q, r\rangle \to \%.$$

To say what the remaining productions of H are, define a function

$$f \in (\mathbf{alphabet}\, G \cup Q_G) \times Q_M \times Q_M \to \mathbf{alphabet}\, G \cup Q_H$$

by: for all $a \in \mathbf{alphabet}\, G \cup Q_G$ and $q, r \in Q_M$,

$$f(a, q, r) = \begin{cases} a, & \text{if } a \in \mathbf{alphabet}\, G, \text{ and} \\ \langle a, q, r\rangle, & \text{if } a \in Q_G. \end{cases}$$

Then, for all $p \in Q_G$, $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in \mathbf{Sym}$ and $q_1, \ldots, q_{n+1} \in Q_M$, if

- $p \to a_1 \cdots a_n \in P_G$, and
- for all $i \in [1 : n]$, if $a_i \in \mathbf{alphabet}\, G$, then $q_{i+1} \in \Delta_M(\{q_i\}, a_i)$,

then we let

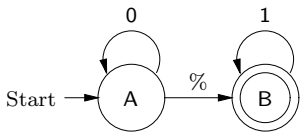
$$\langle p, q_1, q_{n+1}\rangle \to f(a_1, q_1, q_2) \cdots f(a_n, q_n, q_{n+1})$$

be a production of H.

The above definition gives us a function $\mathbf{inter} \in \mathbf{Gram} \times \mathbf{Gram} \to \mathbf{Gram}$. For example, let G be the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1A} \mid \mathsf{1A0A},$$

and M be the EFA



so that G generates all elements of $\{\mathsf{0},\mathsf{1}\}^*$ with an equal number of $\mathsf{0}$'s and $\mathsf{1}$'s, and M accepts $\{\mathsf{0}\}^*\{\mathsf{1}\}^*$. Then $\mathbf{simplify}\, H$ is

$$\begin{aligned}
\mathsf{A} &\to \langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{A}\rangle &\to \%,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle &\to \%,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle &\to \mathsf{0}\langle \mathsf{A}, \mathsf{A}, \mathsf{A}\rangle\mathsf{1}\langle \mathsf{A}, \mathsf{B}, \mathsf{B}\rangle,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle &\to \mathsf{0}\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle\mathsf{1}\langle \mathsf{A}, \mathsf{B}, \mathsf{B}\rangle,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle &\to \mathsf{0}\langle \mathsf{A}, \mathsf{B}, \mathsf{B}\rangle\mathsf{1}\langle \mathsf{A}, \mathsf{B}, \mathsf{B}\rangle,\\
\langle \mathsf{A}, \mathsf{B}, \mathsf{B}\rangle &\to \%.
\end{aligned}$$

Note that simplification eliminated the variable $\langle \mathsf{A}, \mathsf{B}, \mathsf{A}\rangle$. If we hand simplify further, we can turn this into:

$$\begin{aligned}
\mathsf{A} &\to \langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle,\\
\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle &\to \% \mid \mathsf{0}\langle \mathsf{A}, \mathsf{A}, \mathsf{B}\rangle\mathsf{1}
\end{aligned}$$

To prove that our intersection algorithm is correct, we'll need two lemmas.

Lemma 4.7.1
For $p \in Q_G$, let the property $P_p(w)$, for $w \in \Pi_{G,p}$, be:

for all $q, r \in Q_M$, if $r \in \Delta_M(\{q\}, w)$, then $w \in \Pi_{H,\langle p,q,r\rangle}$.

Then, for all $p \in Q_G$, for all $w \in \Pi_{G,p}$, $P_p(w)$.

Proof. By induction on Π. In (2), we use the fact that, if $n \in \mathbb{N} - \{0\}$, $q_1, q_{n+1} \in Q_M$, $w_1, \ldots, w_n \in \mathbf{Str}$ and $q_{n+1} \in \Delta_M(\{q_1\}, w_1 \cdots w_n)$, then there are $q_2, \ldots, q_n \in Q_M$ such that $q_{i+1} \in \Delta_M(\{q_i\}, w_i)$, for all $i \in [1 : n]$. (This is true because M is an EFA; if M were an FA, we wouldn't be able to conclude this.) $\square$

Lemma 4.7.2
Let the property $P_{\mathsf{A}}(w)$, for $w \in \Pi_{H,\mathsf{A}}$, be

$$w \in L(G) \text{ and } w \in L(M).$$

For $p \in Q_G$ and $q, r \in Q_M$, let the property $P_{\langle p,q,r\rangle}(w)$, for $w \in \Pi_{H,\langle p,q,r\rangle}$, be

$$w \in \Pi_{G,p} \text{ and } r \in \Delta_M(\{q\}, w).$$

Then:

(1) For all $w \in \Pi_{H,\mathsf{A}}$, $P_{\mathsf{A}}(w)$.

(2) For all $p \in Q_G$ and $q, r \in Q_M$, for all $w \in \Pi_{H,\langle p,q,r\rangle}$, $P_{\langle p,q,r\rangle}(w)$.

Proof. We proceed by induction on Π. $\square$

Lemma 4.7.3
$L(H) = L(G) \cap L(M)$.

Proof. $L(H) \subseteq L(G) \cap L(M)$ follows by Lemma 4.7.2(1).

For the other inclusion, suppose $w \in L(G) \cap L(M)$, so that $w \in \Pi_{G,s_M}$ and $r \in \Delta_M(\{s_m\}, w)$, for some $r \in A_M$. By Lemma 4.7.1, it follows that $w \in \Pi_{H,\langle s_G, s_M, r\rangle}$. But because $r \in A_M$, we have that $\mathsf{A} \to \langle s_G, s_M, r\rangle$ is a production of H. Thus $w \in \Pi_{H,\mathsf{A}} = L(H)$. $\square$

Finally, we consider a difference operation/algorithm. Given a grammar G and a DFA M, we can define the difference of G and M to be

$$\mathbf{inter}(G, \mathbf{complement}(M, \mathbf{alphabet}\, G)).$$

This is analogous to what we did when defining the difference of DFAs (see Section 3.12). This definition gives us a function $\mathbf{minus} \in \mathbf{Gram} \times \mathbf{DFA} \to \mathbf{Gram}$.

The following theorem summarizes the closure properties for context-free languages.

Theorem 4.7.4
Suppose $L, L_1, L_2 \in \mathbf{CFLan}$ and $L' \in \mathbf{RegLan}$. Then:

(1) $L_1 \cup L_2 \in \mathbf{CFLan}$;

(2) $L_1 L_2 \in$ **CFLan**;

(3) $L^* \in$ **CFLan**;

(4) $L^R \in$ **CFLan**;

(5) $L^f \in$ **CFLan**, *where* f *is a bijection from a set of symbols that is a superset of* **alphabet** L *to some set of symbols;*

(6) $L^P \in$ **CFLan**;

(7) $L^S \in$ **CFLan**;

(8) $L^{SS} \in$ **CFLan**;

(9) $L \cap L' \in$ **CFLan**; *and*

(10) $L - L' \in$ **CFLan**.

4.7.2 Operations on Grammars in Forlan

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
val prefix         : gram -> gram
val inter          : gram * efa -> gram
val minus          : gram * dfa -> gram
```

Most of these functions implement the mathematical functions with the same names. The function `renameAlphabet` raises an exception if its second argument isn't a bijection whose domain is a superset of the alphabet of its first argument. The functions `fromStr` and `fromSym` correspond to **strToGram** and **symToGram**, and are also available in the top-level environment with those names

```
val strToGram : str -> gram
val symToGram : sym -> gram
```

For example, we can construct a grammar G such that $L(G) = \{01\} \cup \{10\}\{11\}^*$, as follows.

```
- val gram1 = strToGram(Str.fromString "01");
val gram1 = - : gram
- val gram2 = strToGram(Str.fromString "10");
val gram2 = - : gram
- val gram3 = strToGram(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
```

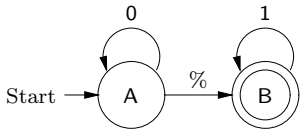
And here is an example use of the prefix-closure operation:

```
- val gram = Gram.input "";
@ {variables}
@ A
@ {start variable}
@ A
@ {productions}
@ A -> % | 0A1
@ .
val gram = - : gram
- val gram' = Gram.prefix gram;
val gram' = - : gram
- Gram.output("", gram');
{variables} <1,A>, <2,0>, <2,1>, <2,A> {start variable} <2,A>
{productions}
<1,A> -> % | 0<1,A>1; <2,0> -> % | 0; <2,1> -> % | 1;
<2,A> -> % | <2,0> | 0<2,A> | 0<1,A><2,1>
val it = () : unit
- fun test s = Gram.generated gram' (Str.fromString s);
val test = fn : string -> bool
- test "000111";
val it = true : bool
- test "0001";
val it = true : bool
- test "0001111";
val it = false : bool
```

Finally, to see how we can use `Gram.inter` and `Gram.minus`, let `gram` be the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1A} \mid \mathsf{1A0A},$$

and `efa` be the EFA




```
- val gram' = Gram.inter(gram, efa);
val gram' = - : gram
- Gram.output("", gram');
{variables} A, <A,A,A>, <A,A,B>, <A,B,B> {start variable} A
{productions}
A -> <A,A,B>; <A,A,A> -> %;
<A,A,B> ->
% | 0<A,A,A>1<A,B,B> | 0<A,A,B>1<A,B,B> | 0<A,B,B>1<A,B,B>;
<A,B,B> -> %
val it = () : unit
- val gram'' =
```

```
=       Gram.eliminateVariable
=       (Gram.eliminateVariable(gram', Sym.fromString "<A,A,A>"),
=        Sym.fromString "<A,B,B>");
val gram'' = - : gram
- val gram''' =
=       Gram.renameVariablesCanonically
=       (Gram.restart(Gram.simplify gram''));
val gram''' = - : gram
- Gram.output("", gram''');
{variables} A {start variable} A {productions} A -> % | 0A1
val it = () : unit
- Gram.generated gram''' (Str.fromString "0011");
val it = true : bool
- Gram.generated gram''' (Str.fromString "0101");
val it = false : bool
- Gram.generated gram''' (Str.fromString "0001");
val it = false : bool
- val dfa =
=       DFA.renameStatesCanonically
=       (DFA.minimize(nfaToDFA(efaToNFA efa)));
val dfa = - : dfa
- val gram'' = Gram.minus(gram, dfa);
val gram'' = - : gram
- Gram.generated gram'' (Str.fromString "0101");
val it = true : bool
- Gram.generated gram'' (Str.fromString "0011");
val it = false : bool
```

4.7.3 Notes

The algorithm for intersecting a grammar with an EFA would normally be given only indirectly, using push down automata (PDAs): one could convert a grammar to a PDF, do the intersection there, and convert back to a grammar. Our direct algorithm is motivated by this process, but produces grammars that are more intelligible.

4.8 Converting Regular Expressions and FA to Grammars

In this section, we give 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 better results can be obtained by defining direct conversions.

4.8.1 Converting Regular Expressions to Grammars

Regular expressions are converted to grammars using a recursive algorithm that makes use of some 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 (see Section 3.12). This gives us a function **regToGram** $\in$ **Reg** $\to$ **Gram**.

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 $01 + 10(11)^*$ 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
```

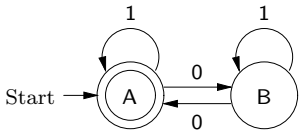
4.8.2 Converting Finite Automata to Grammars

Suppose M is an FA. We define a function/algorithm **faToGram** $\in$ **FA** $\to$ **Gram** by, for all FAs M, **faToGram** M is the grammar G defined below. If $Q_M \cap$ **alphabet** $M = \emptyset$, then G is defined by

- $Q_G = Q_M$;
- $s_G = s_M$;
- $P_G = \{\, q \to xr \mid q, x \to r \in T_M \,\} \cup \{\, q \to \% \mid q \in A_M \,\}$.

Otherwise, we first rename the states of M using a uniform number of $\langle$ and $\rangle$ pairs, so as to avoid conflicts with the elements of M's alphabet.

For example, suppose M is the DFA



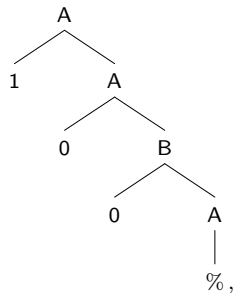
Our algorithm converts M into the grammar

$$\mathsf{A} \to \% \mid \mathsf{0B} \mid \mathsf{1A},$$
$$\mathsf{B} \to \mathsf{0A} \mid \mathsf{1B}.$$

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 $\mathsf{100} \in L(M)$. It corresponds to the valid parse tree for G



which explains why $\mathsf{100} \in L(G)$.

If we have converted an FA M to a grammar G, we can prove $L(M) \subseteq L(G)$ by induction on the lengths of labeled paths, and we can prove $L(G) \subseteq L(M)$ by induction on parse trees. Thus we have $L(G) = L(M)$.

Exercise 4.8.1
State and prove a pair of lemmas that can be used to prove the inclusions $L(M) \subseteq L(G)$ and $L(G) \subseteq L(M)$.

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
```

4.8.3 Consequences of Conversion Functions

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 $\{ \mathsf{0}^n\mathsf{1}^n \mid n \in \mathbb{N} \}$ is context-free, because of the grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1},$$

but is not regular, as we proved in Section 3.14.

Summarizing, we have:

Theorem 4.8.2
The regular languages are a proper subset of the context-free languages: **RegLan** $\subsetneq$ **CFLan**.

4.8.4 Notes

The material in this section is standard.

4.9 Chomsky Normal Form

In this section, we study a special form of grammars called Chomsky Normal Form (CNF), which was 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.

We will begin by giving an algorithm for turning a grammar G into a simplified grammar with no productions of the form $q \to \%$ and $q \to r$. This will enable us to give an algorithm that takes in a grammar G, and calculates $L(G)$, when it is finite, and reports that it is infinite, otherwise.

4.9.1 Removing %-Productions

A %-*production* is a production of the form $q \to \%$. 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

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{BB},$$
$$\mathsf{B} \to \% \mid \mathsf{2B}.$$

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.
- Since $\mathsf{A} \to \mathsf{BB} \in P_G$, it follows that A is nullable.

Now we use this information to compute the productions of our new grammar.

- Since A is nullable, we replace the production $\mathsf{A} \to \mathsf{0A1}$ with the productions $\mathsf{A} \to \mathsf{0A1}$ and $\mathsf{A} \to \mathsf{01}$. 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 $\mathsf{A} \to \mathsf{BB}$ with the productions $\mathsf{A} \to \mathsf{BB}$ and $\mathsf{A} \to \mathsf{B}$ (the result of deleting either one of the B's).
- The production $\mathsf{B} \to \%$ is deleted.
- Since B is nullable, we replace the production $\mathsf{B} \to \mathsf{2B}$ with the productions $\mathsf{B} \to \mathsf{2B}$ and $\mathsf{B} \to \mathsf{2}$.

(If a production has n occurrences of nullable variables in its right side, then there will be 2^n new right sides, corresponding to all ways of deleting or not deleting those n variable occurrences. But if a right side of % would result, we don't include it.) This give us the grammar

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{B},$$
$$\mathsf{B} \to \mathsf{2B} \mid \mathsf{2}.$$

In general, we finish by simplifying our new grammar. The new grammar of our example is already simplified, however.

4.9.2 Removing Unit Productions

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

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{B},$$
$$\mathsf{B} \to \mathsf{2B} \mid \mathsf{2}.$$

We begin by applying our algorithm for removing %-productions to our grammar; the algorithm has no effect in this case.

Our new grammar will have the same variables and start variable as G. Its set of productions is the set of all $q \to w$ such that q is a variable of G, $w \in \mathbf{Str}$ doesn't consist of a single variable of G, and there is a variable r such that

- r is parsable from q, and
- $r \to w$ is a production of G.

(Determining whether r is parsable from q is easy, since we are working with a grammar with no %-productions.)

This process results in the grammar

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{2B} \mid \mathsf{2},$$
$$\mathsf{B} \to \mathsf{2B} \mid \mathsf{2}.$$

Finally, we simplify our grammar, which gets rid of the production $\mathsf{A} \to \mathsf{2B}$, giving us the grammar

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{2},$$
$$\mathsf{B} \to \mathsf{2B} \mid \mathsf{2}.$$

Removing % and Unit Productions in Forlan

The Forlan module `Gram` defines the following functions:

```
val eliminateEmptyProductions        : gram -> gram
val eliminateEmptyAndUnitProductions : gram -> gram
```

For example, if `gram` is the grammar

$$\mathsf{A} \to \mathsf{0A1} \mid \mathsf{BB},$$
$$\mathsf{B} \to \% \mid \mathsf{2B}.$$

then we can proceed as follows.

```
- val gram' = Gram.eliminateEmptyProductions gram;
val gram' = - : gram
- Gram.output("", gram');
{variables} A, B {start variable} A
{productions} A -> B | 01 | BB | 0A1; B -> 2 | 2B
val it = () : unit
- val gram'' = Gram.eliminateEmptyAndUnitProductions gram;
val gram'' = - : gram
- Gram.output("", gram'');
{variables} A, B {start variable} A
{productions} A -> 2 | 01 | BB | 0A1; B -> 2 | 2B
val it = () : unit
```

4.9.3 Generating a Grammar's Language When Finite

We can now give an algorithm that takes in a grammar G and generates $L(G)$, when it is finite, and reports that $L(G)$ is infinite, otherwise. The algorithm begins by letting G' be the result of eliminating %-productions and unit productions from G. Thus G' is simplified and generates $L(G) - \{\%\}$.

If there is recursion in the productions of G'—either direct or mutual—then there is a variable q of G' and a valid parse tree pt for G', such that the height of pt is at least one, q is the root label of pt, and the yield of pt has the form xqy, for strings x and y, each of whose symbols is in $\mathbf{alphabet}\, G' \cup Q_{G'}$. (In particular, q may appear in x or y.) Because G' lacks %- and unit-productions, it follows that $x \neq \%$ or $y \neq \%$. Because each variable of G' is generating, we can turn pt into a valid parse tree pt' whose root label is q, and whose yield has the form uqv, for $u, v \in (\mathbf{alphabet}\, G')^*$, where $u \neq \%$ or $v \neq \%$.

Thus we have that uqv is parsable from q in G', and an easy mathemtical induction shows that $u^n q v^n$ is parsable from q in G', for all $n \in \mathbb{N}$. Because $u \neq \%$ or $v \neq \%$, and q is generating, it follows that there are infinitely many strings generated from q in G'. And, since q is reachable, and every variable of G' is generating, it follows that $L(G')$, and thus $L(G)$, is infinite.

Consequently, our algorthim can continue as follows. If the productions of G' have recursion, then it reports that $L(G)$ is infinite. Otherwise, it calculates $L(G')$ from the bottom-up, and adds % iff G generates %.

The Forlan module `Gram` defines the following function:

```
val toStrSet : gram -> str set
```

Suppose `gram` is the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{BB}, \\ &\mathsf{B} \to \mathsf{CC}, \\ &\mathsf{C} \to \% \mid \mathsf{0} \mid \mathsf{1}, \end{aligned}$$

and `gram'` is the grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{BB}, \\ &\mathsf{B} \to \mathsf{CC}, \\ &\mathsf{C} \to \% \mid \mathsf{0} \mid \mathsf{1} \mid \mathsf{A}. \end{aligned}$$

Then we can proceed as follows:

```
- StrSet.output("", Gram.toStrSet gram);
%, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111,
0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010,
1011, 1100, 1101, 1110, 1111
val it = () : unit
- StrSet.output("", Gram.toStrSet gram');
```

```
language is infinite

uncaught exception Error
```

Suppose we have a grammar G and a natural number n, and we wish to generate the set of all elements of $L(G)$ of length n. We can start by creating an EFA M accepting all strings over the alphabet of G with length n. Then, we can intersect G with M, and apply `Gram.toStrSet` to the resulting grammar.

4.9.4 Chomsky Normal Form

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; and
- $q \to pr$, 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

$$\begin{aligned} &\mathsf{A} \to \mathsf{0A1} \mid \mathsf{01} \mid \mathsf{BB} \mid \mathsf{2}, \\ &\mathsf{B} \to \mathsf{2B} \mid \mathsf{2}. \end{aligned}$$

- We begin by applying our algorithm for removing %-productions and unit productions to this grammar. In this case, it has no effect.
- Since the productions $\mathsf{A} \to \mathsf{BB}$, $\mathsf{A} \to \mathsf{2}$ and $\mathsf{B} \to \mathsf{2}$ 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

$$\langle \mathsf{0} \rangle \to \mathsf{0}, \quad \langle \mathsf{1} \rangle \to \mathsf{1}, \quad \langle \mathsf{2} \rangle \to \mathsf{2}.$$

- Now, we can replace the production $\mathsf{A} \to \mathsf{01}$ with $\mathsf{A} \to \langle \mathsf{0} \rangle \langle \mathsf{1} \rangle$. And, we can replace the production $\mathsf{B} \to \mathsf{2B}$ with the production $\mathsf{B} \to \langle \mathsf{2} \rangle \mathsf{B}$.
- Finally, we replace the production $\mathsf{A} \to \mathsf{0A1}$ with the productions

$$\mathsf{A} \to \langle \mathsf{0} \rangle \mathsf{C}, \quad \mathsf{C} \to \mathsf{A} \langle \mathsf{1} \rangle,$$

  and add C to the set of variables of our new grammar.

Summarizing, our new grammar is

$$\begin{aligned}
\mathsf{A} &\to \mathsf{BB} \mid 2 \mid \langle 0\rangle\langle 1\rangle \mid \langle 0\rangle\mathsf{C},\\
\mathsf{B} &\to 2 \mid \langle 2\rangle\mathsf{B},\\
\langle 0\rangle &\to 0,\\
\langle 1\rangle &\to 1,\\
\langle 2\rangle &\to 2,\\
\mathsf{C} &\to \mathsf{A}\langle 1\rangle.
\end{aligned}$$

The official version of our algorithm names variables in a different way.

Converting to Chomsky Normal Form in Forlan

The Forlan module `Gram` defines the following function:

```
val chomskyNormalForm : gram -> gram
```

Suppose `gram` of type `gram` is bound to the grammar with variables A and B, start variable A, and productions

$$\begin{aligned}
\mathsf{A} &\to 0\mathsf{A}1 \mid \mathsf{BB},\\
\mathsf{B} &\to \% \mid 2\mathsf{B}.
\end{aligned}$$

Here is how Forlan can be used to turn this grammar into a CNF grammar that generates the nonempty strings that are generated by textttgram:

```
- 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>;
<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; B -> 2 | EB; C -> 0; D -> 1; E -> 2;
F -> AD
val it = () : unit
```

4.9.5 Notes

The material in this section is standard.

4.10 The Pumping Lemma for Context-free Languages

Consider the language $L = \{ \mathsf{0}^n\mathsf{1}^n\mathsf{2}^n \mid n \in \mathbb{N} \}$. Is L context-free, i.e., is there a grammar that generates L? Although it's easy to find a grammar that keeps the 0's and 1's matched, or one that keeps the 1's and 2's matched, or one that keeps the 0's and 2's matched, it seems that there is no 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.

4.10.1 Statement, Application and Proof of Pumping Lemma

Lemma 4.10.1 (Pumping Lemma for Context Free Languages)
*For all context-free languages L, there is a $n \in \mathbb{N} - \{0\}$ 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 $L = \{ \mathsf{0}^n\mathsf{1}^n\mathsf{2}^n \mid n \in \mathbb{N} \}$ is not context-free. Suppose, toward a contradiction that L is context-free. Thus there is an $n \in \mathbb{N} - \{0\}$ with the property of the lemma. Let $z = \mathsf{0}^n\mathsf{1}^n\mathsf{2}^n$. 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 $i \in \mathbb{N}$.

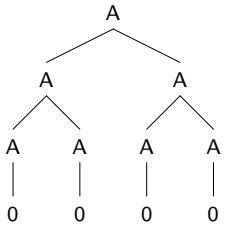
Since $\mathsf{0}^n\mathsf{1}^n\mathsf{2}^n = z = uvwxy$, (1) tells us that vwx doesn't contain both a 0 and a 2. Thus, either vwx has no 0's, or vwx has no 2's, so that there are two cases to consider.

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.

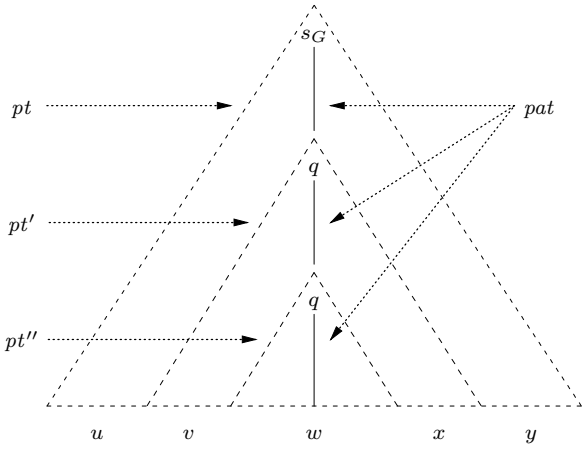
When we prove the pumping lemma for context-free languages, we will make use of a fact about grammars in Chomsky Normal Form. Suppose G is a grammar in CNF and that $w \in (\textbf{alphabet}\, G)^*$ is the yield of a valid parse tree pt for G whose root label is a variable. For instance, if G is the grammar with variable A and productions $\mathsf{A} \to \mathsf{AA}$ and $\mathsf{A} \to \mathsf{0}$, then w could be 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 $4 = 2^2 = 2^{3-1}$. Generalizing still more, we can prove that, for all parse trees pt, for all strings w, if w is the yield of pt, then $|w| \leq 2^{k-1}$. This can be proved by induction on pt.

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 $n = 2^k$. Thus $n \in \mathbb{N} - \{0\}$. Suppose $z \in Str$, $z \in L$ and $|z| \geq n$. Since $n \geq 1$, 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 our fact about CNF grammars, 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



Let pat be a valid path for pt whose length is equal to the height of pt. Thus the length of pat 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.

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 $uv^iwx^iy \in L(G)$. □

4.10.2 Experimenting with the Pumping Lemma Using Forlan

The Forlan module `PT` defines a type and several functions that implement the idea behind the pumping lemma:

```
type pumping_division = (pt * int list) * (pt * int list) * pt

val checkPumpingDivision       : pumping_division -> unit
```

```
val validPumpingDivision       : pumping_division -> bool
val strsOfValidPumpingDivision :
      pumping_division -> str * str * str * str * str
val pumpValidPumpingDivision   : pumping_division * int -> pt
val findValidPumpingDivision   : pt -> pumping_division
```

A *pumping division* is a triple $((pt_1, pat_1), (pt_2, pat_2), pt3)$, where $pt_1, pt_2, pt_3 \in \mathbf{PT}$ and $pat_1, pat_2 \in \mathbf{List}\,\mathbb{Z}$. We say that a pumping division $((pt_1, pat_1), (pt_2, pat_2), pt_3)$ is *valid* iff

- pat_1 is a valid path for pt_1, pointing to a leaf whose label isn't %;
- pat_2 is a valid path for pt_2, pointing to a leaf whose label isn't %;
- the label of the leaf of pt_1 pointed to by pat_1 is equal to the root label of pt_2;
- the label of the leaf of pt_2 pointed to by pat_2 is equal to the root label of pt_2;
- the root label of pt_3 is equal to the root label of pt_2;
- the yield of pt_2 has at least two symbols;
- the yield of pt_1 has only one occurrence of the root label of pt_2;
- the yield of pt_2 has only one occurrence of the root label of pt_2; and
- the yield of pt_3 does not contain the root label of pt_2.

The function `checkPumpingDivision` checks whether a pumping division is valid, silently returning `()` if it is, and explaining why it isn't, otherwise. The function `validPumpingDivision` tests whether a pumping division is valid.

When the function `strsOfValidPumpingDivision` is applied to a valid pumping division $((pt_1, pat_1), (pt_2, pat_2), pt_3)$, it returns (u, v, w, x, y), where:

- u is the prefix of $\mathbf{yield}\, pt_1$ that precedes the unique occurrence of the root label of pt_2;
- v is the prefix of $\mathbf{yield}\, pt_2$ that precedes the unique occurrence of the root label of pt_2;
- $w = \mathbf{yield}\, pt_3$;
- x is the suffix of $\mathbf{yield}\, pt_2$ that follows the unique occurrence of the root label of pt_2; and
- y is the suffix of $\mathbf{yield}\, pt_1$ that follows the unique occurrence of the root label of pt_2.

The function issues an error message if the supplied pumping division isn't valid.

When the function `pumpValidPumpingDivision` is applied to the pair of a valid pumping division $((pt_1, pat_1), (pt_2, pat_2), pt_3)$ and a natural number n, it returns $\mathbf{update}(pt_1, pat_1, \mathbf{pow}\, n)$, where the function $\mathbf{pow} \in \mathbb{N} \to \mathbf{PT}$ is defined by:

$$\begin{aligned} \mathbf{pow}\, 0 &= pt_3, \\ \mathbf{pow}(n+1) &= \mathbf{update}(pt_2, pat_2, \mathbf{pow}\, n), \text{ for all } n \in \mathbb{N}. \end{aligned}$$

The function issues an error message if its first argument isn't valid, or its second argument is negative.

When the function `findValidPumpingDivision` is called with a parse tree pt, it tries to find a valid pumping division pd such that

$$\texttt{pumpValidPumpingDivision}(pd, 1) = pt.$$

It works as follows. First, the leftmost, maximum length path pat through pt is found. If this path points to %, then an error message is issued. Otherwise, `findValidPumpingDivision` generates the following list of variables paired with prefixes of pat:

- the root label of the subtree pointed to by the path consisting of all but the last element of pat, paired with that path;
- the root label of the subtree pointed to by the path consisting of all but the last two elements of pat, paired with that path;
- ...;
- the root label of the subtree pointed to by the path consisting of the first element of pat, paired with that path; and
- the root label of the subtree pointed to by [], paired with [].

(Of course, the left-hand side of the last of these pairs is the root label of pt.) As it works through these pairs, it looks for the first repetition of variables. If there is no such repetition, it issues an error message. Otherwise, suppose that:

- q was the first repeated variable;
- pat_1 was the path paired with q at the point of the first repetition; and
- pat' was the path paired with q when it was first seen.

Now, it lets:

- pat_2 be the result of dropping pat_1 from the beginning of pat';

- pt_1 be **update**(pt, pat_1, q);
- pt' be the subtree of pt pointed to by pat_1;
- pt_2 be **update**(pt', pat_2, q);
- pt_3 be the subtree of pt' pointed to by pat_2; and
- $pd = ((pt_1, pat_1), (pt_2, pat_2), pt_3)$.

If pd is a valid pumping division (only the last four conditions of the definition of validity remain to be checked), it is returned by `findValidPumpingDivision`. Otherwise, an error message is issued.

For example, suppose that `gram` is bound to the grammar

$$\begin{aligned}&\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},\\ &\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},\\ &\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC}.\end{aligned}$$

Then we can proceed as follows:

```
- val pt = Gram.parseAlphabet gram (Str.input "");
@ 1110010010
@ .
val pt = - : pt
- PT.output("", pt);
A
(1, C(1, C(1, C(0), C(0)), C(1, C(0), C(0))),
 A(1, C(0), A(%)))
val it = () : unit
- val pd = PT.findValidPumpingDivision pt;
val pd = ((-,[2,2]),(-,[2]),-) : PT.pumping_division
- val ((pt1, pat1), (pt2, pat2), pt3) = pd;
val pt1 = - : pt
val pat1 = [2,2] : int list
val pt2 = - : pt
val pat2 = [2] : int list
val pt3 = - : pt
- PT.output("", pt1);
A(1, C(1, C, C(1, C(0), C(0))), A(1, C(0), A(%)))
val it = () : unit
- PT.output("", pt2);
C(1, C, C(0))
val it = () : unit
- PT.output("", pt3);
C(0)
val it = () : unit
- val (u, v, w, x, y) = PT.strsOfValidPumpingDivision pd;
val u = [-,-] : str
```

```
val v = [-] : str
val w = [-] : str
val x = [-] : str
val y = [-,-,-,-,-] : str
- (Str.toString u, Str.toString v, Str.toString w,
=  Str.toString x, Str.toString y);
val it = ("11","1","0","0","10010")
  : string * string * string * string * string
- val pt' = PT.pumpValidPumpingDivision(pd, 2);
val pt' = - : pt
- PT.output("", pt');
A
(1, C(1, C(1, C(1, C(0), C(0)), C(0)), C(1, C(0), C(0))),
 A(1, C(0), A(%)))
val it = () : unit
- Str.output("", PT.yield pt');
111100010010
val it = () : unit
```

4.10.3 Consequences of Pumping Lemma

We are now in a position to show that the context-free languages are *not* closed under either intersection or set difference. Suppose

$$\begin{aligned} L &= \{\, 0^n 1^n 2^n \mid n \in \mathbb{N} \,\}, \\ A &= \{\, 0^n 1^n 2^m \mid n, m \in \mathbb{N} \,\}, \text{ and} \\ B &= \{\, 0^n 1^m 2^m \mid n, m \in \mathbb{N} \,\}. \end{aligned}$$

As we proved above, L is not context-free. In contrast, it's easy to find grammars generating A and B, showing that A and B are context-free. But $A \cap B = L$, and thus we have a counterexample to the context-free languages being closed under intersection.

Now, we have that $\{0,1,2\}^* - A$ context-free, since it is the union of the context-free languages

$$\{0,1,2\}^* - \{0\}^*\{1\}^*\{2\}^*$$

and

$$\{\, 0^{n_1} 1^{n_2} 2^m \mid n_1, n_2, m \in \mathbb{N} \text{ and } n_1 \neq n_2 \,\},$$

(the first of these languages is regular), and the context-free languages are closed under union. Similarly, we have that $\{0,1,2\}^* - B$ is context-free.

Let

$$C = (\{0,1,2\}^* - A) \cup (\{0,1,2\}^* - B).$$

Thus C is a context-free subset of $\{0,1,2\}^*$. Since $A, B \subseteq \{0,1,2\}^*$, it is easy to show that

$$\begin{aligned} A \cap B &= \{0,1,2\}^* - ((\{0,1,2\}^* - A) \cup (\{0,1,2\}^* - B)) \\ &= \{0,1,2\}^* - C. \end{aligned}$$

Thus

$$\{0,1,2\}^* - C = A \cap B = L$$

is not context-free, giving us a counterexample to the context-free languages being closed under set difference. Of course, this is also a counterexample to the context-free languages being closed under complementation.

4.10.4 Notes

Apart from the subsection on Forlan's support for experimenting with the pumping lemma, the material in this section is completely standard.

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 halts when run on a given input), 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 simple functional programming language. This language will have the same power as Turing machines, but will be much easier to program in and reason about than 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 Programs and Recursive and RE Languages

In this section, we introduce our functional programming language, and then use it to define the recursive and recursively enumerable languages.

5.1.1 Programs

Our programming language is statically scoped, i.e., nonlocal names used by functions are interpreted relative to the environments in which they are declared. It is dynamically typed, in that all type checking happens at runtime. It is deterministic, in the sense that every evaluation of a program has the same result. It is functional, not imperative (assignment-oriented), i.e., there is no mutable state. And it is eager, not lazy, in the sense that a function's argument must be completely evaluated before the function is called.

To say what programs are, we need some preliminary definitions:

- A *variable* is a nonempty string of letters ($\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}, \mathsf{A}, \mathsf{B}, \ldots, \mathsf{Z}$) and digits ($\mathsf{0}, \mathsf{1}, \ldots, \mathsf{9}$) that begins with a letter. We write **Var** for the set of all variables, and we order variables using the restriction of our total ordering on strings to **Var**.
- A *constant* is one of the strings true, false and nil, and we write **Const** for the set of all constants.
- A *program operator* is one of the strings isNil, isInt, isNeg, isZero, isPos, isSym, isStr, isPair, isLam, plus, minus, compare, fst, snd, consSym, deconsSym, symListToStr and strToSymList, and we write **Oper** for the set of all operators.

Programs are trees (see Section 1.3) whose labels come from the set **ProgLab** of *program labels*, which consists of the union of:

(variable) $\{\,\mathbf{var}(v) \mid v \in \mathbf{Var}\,\}$;

(constant) $\{\,\mathbf{const}(con) \mid con \in \mathbf{Const}\,\}$;

(integer) $\{\,\mathbf{int}(n) \mid n \in \mathbb{Z}\,\}$;

(symbol) $\{\,\mathbf{sym}(a) \mid a \in \mathbf{Sym}\,\}$;

(string) $\{\,\mathbf{str}(x) \mid x \in \mathbf{Str}\,\}$;

(pair) $\{\mathbf{pair}\}$;

(calculation) $\{\,\mathbf{calc}(oper) \mid oper \in \mathbf{Oper}\,\}$;

(conditional) $\{\mathbf{cond}\}$;

(function application) $\{\mathbf{app}\}$;

(anonymous function) $\{\,\mathbf{lam}(v) \mid v \in \mathbf{Var}\,\}$ (**lam** stands for "lambda", recalling the notation for anonymous functions in the λ-calculus);

(simple let) $\{\,\mathbf{letSimp}(v) \mid v \in \mathbf{Var}\,\}$; and

(recursive let) $\{\, \mathbf{letRec}(v_1, v_2) \mid v_1, v_2 \in \mathbf{Var} \,\}$.

Let the set **Prog** of *programs* be the least subset of **Tree ProgLab** such that:

(variable) for all $v \in \mathbf{Var}$,

$$\mathbf{var}(v) \in \mathbf{Prog}$$

(a leaf);

(constant) for all $con \in \mathbf{Const}$,

$$\mathbf{const}(con) \in \mathbf{Prog}$$

(a leaf);

(integer) for all $n \in \mathbb{Z}$,

$$\mathbf{int}(n) \in \mathbf{Prog}$$

(a leaf);

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

$$\mathbf{sym}(a) \in \mathbf{Prog}$$

(a leaf);

(string) for all $x \in \mathbf{Str}$,

$$\mathbf{str}(x) \in \mathbf{Prog}$$

(a leaf);

(pair) for all $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{pair}(pr_1, pr_2) \in \mathbf{Prog}$$

(a node labeled **pair** with two children);

(calculation) for all $oper \in \mathbf{Oper}$ and $pr \in \mathbf{Prog}$,

$$\mathbf{calc}(oper)(pr) \in \mathbf{Prog}$$

(a node labeled $\mathbf{calc}(oper)$, with one child; we abbreviate it as $\mathbf{calc}(oper, pr)$);

(conditional) for all $pr_1, pr_2, pr_3 \in \mathbf{Prog}$,

$$\mathbf{cond}(pr_1, pr_2, pr_3) \in \mathbf{Prog}$$

(a node labeled **cond**, with three children);

(function application) for all $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{app}(pr_1, pr_2) \in \mathbf{Prog}$$

(a node labeled **app**, with two children);

(anonymous function) for all $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$,

$$\mathbf{lam}(v)(pr) \in \mathbf{Prog}$$

(a node labeled $\mathbf{lam}(v)$, with one child; we abbreviate it as $\mathbf{lam}(v, pr)$);

(simple let) for all $v \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{letSimp}(v)(pr_1, pr_2) \in \mathbf{Prog}$$

(a node labeled $\mathbf{letSimp}(v)$, with two children; we abbreviate it as $\mathbf{letSimp}(v, pr_1, pr_2)$); and

(recursive let) for all $v_1, v_2 \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{letRec}(v_1, v_2)(pr_1, pr_2) \in \mathbf{Prog}$$

(a node labeled $\mathbf{letRec}(v_1, v_2)$, with two children; we abbreviate it as $\mathbf{letRec}(v_1, v_2, pr_1, pr_2)$).

Prog is countably infinite.

Informally:

- A pair $\mathbf{pair}(pr_1, pr_2)$ is evaluated by evaluating pr_1 and then pr_2.
- A calculation $\mathbf{calc}(oper, pr)$ applies the operator *oper* to the result of evaluating pr.
- A conditional $\mathbf{cond}(pr_1, pr_2, pr_3)$ first evaluates pr_1 to a boolean constant; if this constant is $\mathbf{const}(\mathsf{true})$, it evaluates pr_2; and if this constant is $\mathbf{const}(\mathsf{false})$, it evaluates pr_3.
- A function application $\mathbf{app}(pr_1, pr_2)$ evaluates pr_1 to an anonymous function, and then applies this function to the result of evaluating pr_2.
- If an anonymous function $\mathbf{lam}(v, pr)$ is applied to a fully evaluated argument, then pr is evaluated in an environment in which v is bound to the argument.
- A simple let $\mathbf{letSimp}(v, pr_1, pr_2)$ is evaluated by evaluating pr_2 in an environment in which v is bound to the result of evaluating pr_1.

- A recursive let $\mathbf{letRec}(v_1, v_2, pr_1, pr_2)$ is evaluated by evaluating pr_2 in an envionment in which v_1 has been recursively declared by:

$$v_1 = \mathbf{lam}(v_2, pr_1).$$

Lists are represented by pairs:

$$\mathbf{pair}(pr_1, \mathbf{pair}(pr_2, \ldots \mathbf{pair}(pr_n, \mathbf{const}(\mathsf{nil}))))$$

represents the n-length list whose elements are pr_1, pr_2, ..., pr_n (so $\mathbf{const}(\mathsf{nil})$ represents the empty list).

In the Forlan syntax for programs, the elements of **Prog** are written using our standard syntax for trees, except that:

- The integer arguments to **int** are written as base-10 numerals, preceded by `~` in the case of negative integers. The integer 0 is written as `0`; extra zeros aren't used/allowed. Positive integers are written without leading zeros. And negative integers are written written as `~` followed by a nonempty string of digits with no leading zeros.
- The symbol arguments to **sym**, and the string arguments to **str** are written in abbreviated form, as usual.
- All nodes are abbreviated. E.g.,

$$\mathbf{letSimp}(\mathsf{x})(\mathbf{int}(-12), \mathbf{var}(\mathsf{x}))$$

is written as

```
letSimp(x, int(~12), var(x))
```

Programs can also be described as strings over the alphabet consisting of the letters and digits, plus the elements of

$$\{\langle\mathsf{comma}\rangle, \langle\mathsf{perc}\rangle, \langle\mathsf{tilde}\rangle, \langle\mathsf{openPar}\rangle, \langle\mathsf{closPar}\rangle, \langle\mathsf{less}\rangle, \langle\mathsf{great}\rangle\}.$$

A program pr is described the string formed by writing pr in Forlan's syntax, using no whitespace, and then performing the following substitutions:

- `,` is replaced by $\langle\mathsf{comma}\rangle$;
- `%` is replaced by $\langle\mathsf{perc}\rangle$;
- `~` is replaced by $\langle\mathsf{tilde}\rangle$;
- `(` is replaced by $\langle\mathsf{openPar}\rangle$;
- `)` is replaced by $\langle\mathsf{closPar}\rangle$;

- < is replaced by ⟨less⟩; and
- > is replaced by ⟨great⟩.

For example, the program

$$\mathbf{calc}(\mathsf{plus}, \mathbf{pair}(\mathbf{int}(4), \mathbf{int}(-5)))$$

is described by the string

calc⟨openPar⟩plus⟨comma⟩pair⟨openPar⟩int⟨openPar⟩4⟨closPar⟩
⟨comma⟩int⟨openPar⟩⟨tilde⟩5⟨closPar⟩⟨closPar⟩⟨closPar⟩.

Every program is described by a unique string, and every string describes at most one program. (E.g., the string ⟨comma⟩⟨closPar⟩ doesn't describe a program.)

The Forlan module `Var` defines the abstract type (in the top-level environment) `var` of variables, along with functions, including:

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

The function `compare` implements our total ordering on variables, and `equal` tests whether two variables are equal.

The module `VarSet` defines various functions for processing finite sets of variables (elements of type `var set`),

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

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

The function `fromList` returns a set with the same elements of the list of variables it is called with. The function `memb` tests whether a variable is a member (element) of a set of variables, `subset` tests whether a first set of variables is a subset of a second one, and `equal` tests whether two sets of variables are equal. The functions `union`, `inter` and `minus` compute the union, intersection

and difference of two sets of variables. The function `genUnion` computes the generalized intersection of a list of sets of variables *xss*, returning the set of all variables appearing in at least one element of *xss*. And, the function `genInter` computes the generalized intersection of a nonempty list of sets of variables *xss*, returning the set of all variables appearing in all elements of *xss*.

The Forlan module `Prog` defines the abstract type (in the top-level environment) `prog` of programs, along with a number of types and functions, including:

```
datatype const = True | False | Nil
datatype oper = IsNil | IsInt | IsNeg | IsZero | IsPos | IsSym
              | IsStr | IsPair | IsLam | Plus | Minus | Compare
              | Fst | Snd | ConsSym | DeconsSym | SymListToStr
              | StrToSymList
val var     : Var.var -> prog
val const   : const -> prog
val int     : IntInf.int -> prog
val sym     : sym -> prog
val str     : str -> prog
val pair    : prog * prog -> prog
val calc    : oper * prog -> prog
val cond    : prog * prog * prog -> prog
val app     : prog * prog -> prog
val lam     : var * prog -> prog
val letSimp : var * prog * prog -> prog
val letRec  : var * var * prog * prog -> prog
val input   : string -> prog
val output  : string * prog -> unit
val equal   : prog * prog -> bool
val height  : prog -> int
val size    : prog -> int
val fromStr : str -> prog
val toStr   : prog -> str
```

`const` and `oper` are the datatypes of program constants and operators. The function `var` takes in a program variable v, and returns the program $\mathbf{var}(v)$. The function `const` takes in a program constant *con*, and returns the program $\mathbf{const}(con)$. The function `int` takes in a (infinite precision) integer n, and returns the program $\mathbf{int}(n)$. The function `sym` takes in a symbol a, and returns the program $\mathbf{sym}(a)$. The function `str` takes in a string x, and returns the program $\mathbf{str}(x)$. The function `pair` takes in a pair (pr_1, pr_2) of programs, and returns the program $\mathbf{pair}(pr_1, pr_2)$. The function `calc` takes in a pair $(oper, pr)$ of a program operator and a program, and returns the program $\mathbf{calc}(oper, pr)$. The function `cond` takes in a triple (pr_1, pr_2, pr_3) of programs, and returns the program $\mathbf{cond}(pr_1, pr_2, pr_3)$. The function `app` takes in a pair (pr_1, pr_2) of programs, and returns the program $\mathbf{app}(pr_1, pr_2)$. The function `lam` takes in a pair (v, pr) of a variable v and a program pr, and returns the program $\mathbf{lam}(v, pr)$. The function `letSimp` takes in a triple (v, pr_1, pr_2) of a variable

v and programs pr_1 and pr_2, and returns the program **letSimp**(v, pr_1, pr_2). The function `letRec` takes in a quadruple (v_1, v_2, pr_1, pr_2) of variables v_1 and v_2 and programs pr_1 and pr_2, and returns the program **letRec**(v_1, v_2, pr_1, pr_2). The function `equal` tests two programs for equality. The functions `height` and and `size` return the height and size, respectively, of a program. The function `fromStr` issues an error message if its argument isn't a string over the alphabet (given above) for describing programs, or if it is a string over this alphabet, but doesn't describe a program; otherwise it returns the unique program described by its argument. And the function `toStr` converts a program to the unique string describing it.

For example, we can proceed as follows:

```
- val pr = Prog.input "";
@ calc(plus, pair(int(~6), int(5)))
@ .
val pr = - : prog
- Prog.height pr;
val it = 2 : int
- Prog.size pr;
val it = 4 : int
- val x = Prog.toStr pr;
val x =
  [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-]
  : str
- Str.output("", x);
calc<openPar>plus<comma>pair<openPar>int<openPar><tilde>6<closPar>
<comma>int<openPar>5<closPar><closPar><closPar>
val it = () : unit
- val pr' = Prog.fromStr x;
val pr' = - : prog
- Prog.equal(pr', pr);
val it = true : bool
```

The Java program JForlan, can be used to view and edit program trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

5.1.2 Program Meaning

Variables are *bound* (declared) in programs, as follows:

- **lam**(v, pr) binds v in pr;
- **letSimp**(v, pr_1, pr_2) binds v in pr_2; and
- **letSimp**(v_1, v_2, pr_1, pr_2) binds v_1 in pr_1 and pr_2, and binds v_2 in pr_1.

We say that an occurrence of a variable in a program is *free* iff it isn't bound.

Formally, we define a function $\mathbf{free} \in \mathbf{Prog} \to \{\, X \subseteq \mathbf{Var} \mid X \text{ is finite} \,\}$ by structural recursion:

- $\mathbf{free}(\mathbf{var}(v)) = \emptyset$, for all $v \in \mathbf{Var}$;
- $\mathbf{free}(\mathbf{const}(con)) = \emptyset$, for all $con \in \mathbf{Const}$;
- $\mathbf{free}(\mathbf{int}(n)) = \emptyset$, for all $n \in \mathbb{Z}$;
- $\mathbf{free}(\mathbf{sym}(a)) = \emptyset$, for all $a \in \mathbf{Sym}$;
- $\mathbf{free}(\mathbf{str}(x)) = \emptyset$, for all $x \in \mathbf{Str}$;
- $\mathbf{free}(\mathbf{pair}(pr_1, pr_2)) = \mathbf{free}\, pr_1 \cup \mathbf{free}\, pr_2$, for all $pr_1, pr_2 \in \mathbf{Prog}$;
- $\mathbf{free}(\mathbf{calc}(oper, pr)) = \mathbf{free}\, pr$, for all $oper \in \mathbf{Oper}$ and $pr \in \mathbf{Prog}$;
- $\mathbf{free}(\mathbf{cond}(pr_1, pr_2, pr_3)) = \mathbf{free}\, pr_1 \cup \mathbf{free}\, pr_2 \cup \mathbf{free}\, pr_3$, for all $pr_1, pr_2, pr_3 \in \mathbf{Prog}$;
- $\mathbf{free}(\mathbf{app}(pr_1, pr_2)) = \mathbf{free}\, pr_1 \cup \mathbf{free}\, pr_2$, for all $pr_1, pr_2 \in \mathbf{Prog}$;
- $\mathbf{free}(\mathbf{lam}(v, pr)) = \mathbf{free}\, pr - \{v\}$, for all $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$;
- $\mathbf{free}(\mathbf{letSimp}(v, pr_1, pr_2)) = \mathbf{free}\, pr_1 \cup (\mathbf{free}\, pr_2 - \{v\})$, for all $v \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$; and
- $\mathbf{free}(\mathbf{letRec}(v_1, v_2, pr_1, pr_2)) = (\mathbf{free}\, pr_1 - \{v_1, v_2\}) \cup (\mathbf{free}\, pr_2 - \{v_1\})$, for all $v_1, v_2 \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$.

If $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$, we say that v *is free in* pr iff $v \in \mathbf{free}\, pr$.

A program pr is *closed* iff it has no free variables, i.e., $\mathbf{free}\, pr = \emptyset$. We write $\mathbf{CP}$ for the set of all closed programs.

The module `Prog` also defines the following type and functions:

```
val free : prog -> var set
type cp
val toClosed   : prog -> cp
val fromClosed : cp -> prog
```

The function `free` returns the free variables of its argument. The type `cp` (in the top-level environment) is the type of closed programs. The function `toClosed` issues an error message if its argument isn't closed; otherwise, it returns its argument. And the function `fromClosed` simply returns its argument.

For example, we can proceed as follows:

```
- val pr1 = Prog.input "";
@ lam(x, letSimp(y, var(x), app(var(z), var(w))))
@ .
val pr1 = - : prog
```

```
- VarSet.output("", Prog.free pr1);
w, z
val it = () : unit
- val pr2 = Prog.input "";
@ letRec
@ (x, y,
@  app(var(x), app(var(y), var(z))),
@  app(var(x), var(w)))
@ .
val pr2 = - : prog
- VarSet.output("", Prog.free pr2);
w, z
val it = () : unit
- Prog.toClosed pr2;
program has free variables: "w, z"

uncaught exception Error
- val pr' = Prog.input "";
@ lam(x, letSimp(y, var(x), app(var(x), var(y))))
@ .
val pr' = - : prog
- val cp = Prog.toClosed pr';
val cp = - : cp
- val pr'' = Prog.fromClosed cp;
val pr'' = - : prog
- Prog.equal(pr'', pr');
val it = true : bool
```

Next, we define a function $\mathbf{subst} \in \mathbf{CP} \times \mathbf{Var} \times \mathbf{Prog} \rightarrow \mathbf{Prog}$ for substituting a closed program for all of the free occurrences of a variable in a program. **subst** is defined by structural recursion on its third argument, as follows:

- for all $pr' \in \mathbf{Prog}$, $v' \in \mathbf{Var}$ and $v \in \mathbf{Var}$,

$$\mathbf{subst}(pr', v', \mathbf{var}(v)) = \begin{cases} pr', & \text{if } v' = v, \\ \mathbf{var}(v), & \text{if } v' \neq v; \end{cases}$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $con \in \mathbf{Const}$,

$$\mathbf{subst}(pr', v', \mathbf{const}(con)) = \mathbf{const}(con);$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $n \in \mathbb{Z}$,

$$\mathbf{subst}(pr', v', \mathbf{int}(n)) = \mathbf{int}(n);$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $a \in \mathbf{Sym}$,

$$\mathbf{subst}(pr', v', \mathbf{sym}(a)) = \mathbf{sym}(a);$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $x \in \mathbf{Str}$,

$$\mathbf{subst}(pr', v', \mathbf{str}(x)) = \mathbf{str}(x);$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{pair}(pr_1, pr_2)) = \mathbf{pair}(\mathbf{subst}(pr', v', pr_1), \mathbf{subst}(pr', v', pr_2));$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$, $oper \in \mathbf{Oper}$ and $pr \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{calc}(oper, pr)) = \mathbf{calc}(oper, \mathbf{subst}(pr', v', pr));$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $pr_1, pr_2, pr_3 \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{cond}(pr_1, pr_2, pr_3)) = \mathbf{cond}(\mathbf{subst}(pr', v', pr_1), \mathbf{subst}(pr', v', pr_2), \mathbf{subst}(pr', v', pr_3));$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{app}(pr_1, pr_2)) = \mathbf{app}(\mathbf{subst}(pr', v', pr_1), \mathbf{subst}(pr', v', pr_2));$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$, $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{lam}(v, pr)) = \begin{cases} \mathbf{lam}(v, pr), & \text{if } v' = v, \\ \mathbf{lam}(v, \mathbf{subst}(pr', v', pr)), & \text{if } v' \neq v; \end{cases}$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$, $v \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{letSimp}(v, pr_1, pr_2)) = \begin{cases} \mathbf{letSimp}(v, \mathbf{subst}(pr', v', pr_1), pr_2), & \text{if } v' = v, \\ \mathbf{letSimp}(v, \mathbf{subst}(pr', v', pr_1), \mathbf{subst}(pr', v', pr_2)), & \text{if } v' \neq v; \end{cases}$$

- for all $pr' \in \mathbf{CP}$, $v' \in \mathbf{Var}$, $v_1, v_2 \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\mathbf{subst}(pr', v', \mathbf{letRec}(v_1, v_2, pr_1, pr_2)) = \begin{cases} \mathbf{letRec}(v_1, v_2, pr_1, pr_2), \\ \quad \text{if } v' = v_1, \\ \mathbf{letRec}(v_1, v_2, pr_1, \mathbf{subst}(pr', v', pr_2)), \\ \quad \text{if } v' \neq v_1 \text{ and } v' = v_2, \\ \mathbf{letRec}(v_1, v_2, \mathbf{subst}(pr', v', pr_1), \mathbf{subst}(pr', v', pr_2)), \\ \quad \text{if } v' \neq v_1 \text{ and } v' \neq v_2. \end{cases}$$

The module `Prog` also defines the function

```
val subst : cp * var * prog -> prog
```

corresponding to **subst**. Here are some examples of its use:

```
- val cp = Prog.toClosed(Prog.fromString "const(true)");
val cp = - : cp
- val pr1 = Prog.input "";
@ cond(var(x), var(y), int(4))
@ .
val pr1 = - : prog
- val pr1' = Prog.subst(cp, Var.fromString "y", pr1);
val pr1' = - : prog
- Prog.output("", pr1');
cond(var(x), const(true), int(4))
val it = () : unit
- val pr2 = Prog.input "";
@ letSimp(x, var(x), pair(var(x), var(y)))
@ .
val pr2 = - : prog
- val pr2' = Prog.subst(cp, Var.fromString "x", pr2);
val pr2' = - : prog
- Prog.output("", pr2');
letSimp(x, const(true), pair(var(x), var(y)))
val it = () : unit
- val pr3 = Prog.input "";
@ letSimp(x, var(y), pair(var(x), var(y)))
@ .
val pr3 = - : prog
- val pr3' = Prog.subst(cp, Var.fromString "y", pr3);
val pr3' = - : prog
- Prog.output("", pr3');
letSimp(x, const(true), pair(var(x), const(true)))
val it = () : unit
- val pr4 = Prog.input "";
@ letRec
@ (x, y,
@  app(var(x), app(var(y), var(z))),
@  app(var(x), app(var(y), var(z))))
@ .
val pr4 = - : prog
- val pr4' = Prog.subst(cp, Var.fromString "x", pr4);
val pr4' = - : prog
- Prog.output("", pr4');
letRec(x, y, app(var(x), app(var(y), var(z))),
       app(var(x), app(var(y), var(z))))
val it = () : unit
- val pr4'' = Prog.subst(cp, Var.fromString "y", pr4);
val pr4'' = - : prog
```

```
- Prog.output("", pr4'');
letRec(x, y, app(var(x), app(var(y), var(z))),
       app(var(x), app(const(true), var(z))))
val it = () : unit
- val pr4''' = Prog.subst(cp, Var.fromString "z", pr4);
val pr4''' = - : prog
- Prog.output("", pr4''');
letRec(x, y, app(var(x), app(var(y), const(true))),
       app(var(x), app(var(y), const(true))))
val it = () : unit
```

Next, we single out certain closed programs as completely evaluated, or values. Let the set **Val** of *values* be the least subset of **CP** such that:

(constant) for all $con \in \mathbf{Const}$, $\mathbf{const}(con) \in \mathbf{Val}$;

(integer) for all $n \in \mathbb{Z}$, $\mathbf{int}(n) \in \mathbf{Val}$;

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

(string) for all $x \in \mathbf{Str}$, $\mathbf{str}(x) \in \mathbf{Val}$;

(pair) for all $pr_1, pr_2 \in \mathbf{Val}$, $\mathbf{pair}(pr_1, pr_2) \in \mathbf{Val}$; and

(anonymous function) for all $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$, if $\mathbf{free}\, pr \subseteq \{v\}$, then $\mathbf{lam}(v, pr) \in \mathbf{Val}$.

The module `Prog` also defines a function

```
val isValue : cp -> bool
```

that tests whether a closed program is a value. Here are some examples of its use:

```
- val pr1 = Prog.input "";
@ pair(int(4), pair(lam(x, var(x)), const(true)))
@ .
val pr1 = - : prog
- Prog.isValue(Prog.toClosed pr1);
val it = true : bool
- val pr2 = Prog.input "";
@ app(lam(x, var(x)), lam(x, var(x)))
@ .
val pr2 = - : prog
- Prog.isValue(Prog.toClosed pr2);
val it = false : bool
```

To explain the meaning of program operators, we define a function $\mathbf{calculate} \in \mathbf{Oper} \times \mathbf{Val} \to \mathbf{Option}\,\mathbf{Val}$, which returns **none** to indicate an error, and returns $\mathbf{some}\, pr$ when the application of the operator produced the value pr. We proceed as follows, using a case analysis on the form of the argument value:

- $((\mathsf{isNil}, \mathbf{const}(\mathsf{nil})))$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isNil}, pr)$, where $pr \notin \{\mathbf{const}(\mathsf{nil})\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{isInt}, \mathbf{int}(n))$, where $n \in \mathbb{Z})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isInt}, pr)$, where $pr \notin \{\, \mathbf{int}(n) \mid n \in \mathbb{Z} \,\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{isNeg}, \mathbf{int}(n))$, where $n \in \mathbb{Z})$ Return: $\mathbf{some}(\mathbf{const}(\mathsf{true}))$, if $n < 0$; and $\mathbf{some}(\mathbf{const}(\mathsf{false}))$, if $n \geq 0$.
- $((\mathsf{isZero}, \mathbf{int}(n))$, where $n \in \mathbb{Z})$ Return: $\mathbf{some}(\mathbf{const}(\mathsf{true}))$, if $n = 0$; and $\mathbf{some}(\mathbf{const}(\mathsf{false}))$, if $n \neq 0$.
- $((\mathsf{isPos}, \mathbf{int}(n))$, where $n \in \mathbb{Z})$ Return: $\mathbf{some}(\mathbf{const}(\mathsf{true}))$, if $n > 0$; and $\mathbf{some}(\mathbf{const}(\mathsf{false}))$, if $n \leq 0$.
- $((\mathsf{isSym}, \mathbf{sym}(a))$, where $a \in \mathbf{Sym})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isSym}, pr)$, where $pr \notin \{\, \mathbf{sym}(a) \mid a \in \mathbf{Sym} \,\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{isStr}, \mathbf{str}(x))$, where $x \in \mathbf{Str})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isStr}, pr)$, where $pr \notin \{\, \mathbf{str}(x) \mid x \in \mathbf{Str} \,\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{isPair}, \mathbf{pair}(pr_1, pr_2))$, where $pr_1, pr_2 \in \mathbf{Val})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isPair}, pr)$, where $pr \notin \{\, \mathbf{pair}(pr_1, pr_2) \mid pr_1, pr_2 \in \mathbf{Val} \,\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{isLam}, \mathbf{lam}(v, pr))$, where $v \in \mathbf{Var}$, $pr \in \mathbf{Prog}$ and $\mathbf{free}\, pr \subseteq \{v\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{true}))$.
- $((\mathsf{isLam}, pr)$, where $pr \notin \{\, \mathbf{lam}(v, pr) \mid v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$ and $\mathbf{free}\, pr \subseteq \{v\} \,\})$ Return $\mathbf{some}(\mathbf{const}(\mathsf{false}))$.
- $((\mathsf{plus}, \mathbf{pair}(\mathbf{int}(m), \mathbf{int}(n)))$, where $m, n \in \mathbb{Z})$ Return $\mathbf{some}(\mathbf{int}(m + n))$.
- $((\mathsf{minus}, \mathbf{pair}(\mathbf{int}(m), \mathbf{int}(n)))$, where $m, n \in \mathbb{Z})$ Return $\mathbf{some}(\mathbf{int}(m - n))$.
- $((\mathsf{compare}, \mathbf{pair}(\mathbf{int}(m), \mathbf{int}(n)))$, where $m, n \in \mathbb{Z})$ Return: $\mathbf{some}(\mathbf{int}(-1))$, if $m < n$; $\mathbf{some}(\mathbf{int}(0))$, if $m = n$; and $\mathbf{some}(\mathbf{int}(1))$, if $m > n$.
- $((\mathsf{compare}, \mathbf{pair}(\mathbf{sym}(a), \mathbf{sym}(b)))$, where $a, b \in \mathbf{Sym})$ Return: $\mathbf{some}(\mathbf{int}(-1))$, if $a < b$; $\mathbf{some}(\mathbf{int}(0))$, if $a = b$; and $\mathbf{some}(\mathbf{int}(1))$, if $a > b$.
- $((\mathsf{compare}, \mathbf{pair}(\mathbf{str}(x), \mathbf{str}(y)))$, where $x, y \in \mathbf{Str})$ Return: $\mathbf{some}(\mathbf{int}(-1))$, if $x < y$; $\mathbf{some}(\mathbf{int}(0))$, if $x = y$; and $\mathbf{some}(\mathbf{int}(1))$, if $x > y$.

- $((\mathsf{fst}, \mathbf{pair}(pr_1, pr_2))$, where $pr_1, pr_2 \in \mathbf{Val})$ Return $\mathbf{some}\, pr_1$.
- $((\mathsf{snd}, \mathbf{pair}(pr_1, pr_2))$, where $pr_1, pr_2 \in \mathbf{Val})$ Return $\mathbf{some}\, pr_2$.
- $((\mathsf{consSym}, \mathbf{int}(n))$, where $n \in [1:62])$ Return $\mathbf{some}(\mathbf{sym}(a))$, where a is the nth (counting from 1) element of the following sequence of symbols: $\mathsf{0}, \ldots, \mathsf{9}, \mathsf{a}, \ldots, \mathsf{z}, \mathsf{A}, \ldots, \mathsf{Z}$.
- $((\mathsf{consSym}, \mathbf{pair}(pr_1, \ldots \mathbf{pair}(pr_n, \mathbf{const}(\mathsf{nil})) \ldots))$, where $n \in \mathbb{N}$ and $pr_1, \ldots, pr_n \in \{\mathbf{const}(\mathsf{nil})\} \cup \{\, \mathbf{sym}(a) \mid a \in \mathbf{Sym}\,\})$ Return

  $$\mathbf{some}(\mathbf{sym}([\,\langle\,] \mathbin{@} f\, pr_1 \mathbin{@} \cdots \mathbin{@} f\, pr_n \mathbin{@} [\,\rangle\,])),$$

  where f is the function from $\{\mathbf{const}(\mathsf{nil})\} \cup \{\, \mathbf{sym}(a) \mid a \in \mathbf{Sym}\,\}$ to $\{[\,,]\} \cup \mathbf{Sym}$ defined by:

  $$\begin{aligned} f(\mathbf{const}(\mathsf{nil})) &= [\,,], \\ f(\mathbf{sym}(a)) &= a, \text{ for all } a \in \mathbf{Sym}. \end{aligned}$$

  (Remember that symbols are lists; see Section 2.1. If $n = 0$, then $\mathbf{some}(\mathbf{sym}(\langle\rangle))$ is returned.)
- $((\mathsf{deconsSym}, \mathbf{sym}(a))$, where $a \in \{\mathsf{0}, \ldots, \mathsf{9}, \mathsf{a}, \ldots, \mathsf{z}, \mathsf{A}, \ldots, \mathsf{Z}\})$ Return $\mathbf{some}(\mathbf{int}(n))$, where n is the position (counting from 1) of a in the following sequence of symbols: $\mathsf{0}, \ldots, \mathsf{9}, \mathsf{a}, \ldots, \mathsf{z}, \mathsf{A}, \ldots, \mathsf{Z}$.
- $((\mathsf{deconsSym}, \mathbf{sym}([\,\langle\,] \mathbin{@} x_1 \mathbin{@} \cdots \mathbin{@} x_n \mathbin{@} [\,\rangle\,]))$, where $n \in \mathbb{N}$ and $x_1, \ldots, x_n \in \{[\,,]\} \cup \mathbf{Sym})$ Return

  $$\mathbf{some}(\mathbf{pair}(f\, x_1, \ldots \mathbf{pair}(f\, x_n, \mathbf{const}(\mathsf{nil})) \ldots)),$$

  where f is the function from $\{[\,,]\} \cup \mathbf{Sym}$ to $\{\mathbf{const}(\mathsf{nil})\} \cup \{\, \mathbf{sym}(a) \mid a \in \mathbf{Sym}\,\}$ defined by

  $$\begin{aligned} f[\,,] &= \mathbf{const}(\mathsf{nil}), \\ f\, a &= \mathbf{sym}(a), \text{ for all } a \in \mathbf{Sym}. \end{aligned}$$

  (If $n = 0$, then $\mathbf{some}(\mathbf{const}(\mathsf{nil}))$ is returned.)
- $((\mathsf{symListToStr}, \mathbf{const}(\mathsf{nil})))$ Return $\mathbf{some}(\mathbf{str}(\%))$.
- $((\mathsf{symListToStr}, \mathbf{pair}(\mathbf{sym}(a_1), \ldots \mathbf{pair}(\mathbf{sym}(a_n), \mathbf{const}(\mathsf{nil})) \ldots))$, where $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \mathbf{Sym})$ Return $\mathbf{some}(\mathbf{str}(a_1 \cdots a_n))$.
- $((\mathsf{strToSymList}, \mathbf{str}(\%)))$ Return $\mathbf{some}(\mathbf{const}(\mathsf{nil}))$.
- $((\mathsf{strToSymList}, \mathbf{str}(a_1 \cdots a_n))$, where $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \mathbf{Sym})$ Return

  $$\mathbf{some}(\mathbf{pair}(\mathbf{sym}(a_1), \ldots \mathbf{pair}(\mathbf{sym}(a_n), \mathbf{const}(\mathsf{nil})) \ldots)).$$

- (otherwise) Return **none**.

Now we are able to say how closed programs evaluate. Let

$$\mathbf{Step} = \{\mathbf{value}, \mathbf{error}\} \cup \{\, \mathbf{next}\, pr \mid pr \in \mathbf{CP} \,\}.$$

We define a function $\mathbf{step} \in \mathbf{CP} \to \mathbf{Step}$ using a case analysis on the form of its argument, pr. It returns: **value**, if pr is a value; **error**, if pr isn't a value, but can't be run a single step; and $\mathbf{next}\, pr'$, for $pr' \in \mathbf{CP}$, if pr' is the result of running pr for one step. We proceed as follows:

- ($\mathbf{const}(con)$, where $con \in \mathbf{Const}$) Return **value**.
- ($\mathbf{int}(n)$, where $n \in \mathbb{Z}$) Return **value**.
- ($\mathbf{sym}(a)$, where $a \in \mathbf{Sym}$) Return **value**.
- ($\mathbf{str}(x)$, where $x \in \mathbf{Str}$) Return **value**.
- ($\mathbf{pair}(pr_1, pr_2)$, where $pr_1, pr_2 \in \mathbf{CP}$) Use case analysis on the form of $\mathbf{step}\, pr_1$:
  - (**error**) Return **error**.
  - (**value**) Use case analysis on the form of $\mathbf{step}\, pr_2$:
    - (**error**) Return **error**.
    - (**value**) Return **value**.
    - ($\mathbf{next}\, pr_2'$, where $pr_2' \in \mathbf{CP}$) Return $\mathbf{pair}(pr_1, pr_2')$.
  - ($\mathbf{next}\, pr_1'$, where $pr_1' \in \mathbf{CP}$) Return $\mathbf{next}(\mathbf{pair}(pr_1', pr_2))$.
- ($\mathbf{calc}(oper, pr)$, where $oper \in \mathbf{Oper}$ and $pr \in \mathbf{CP}$) Use case analysis on the form of $\mathbf{step}\, pr$:
  - (**error**) Return **error**.
  - (**value**) Use case analysis on the form of $\mathbf{calculate}(oper, pr)$:
    - (**none**) Return **error**.
    - ($\mathbf{some}\, pr'$, where $pr' \in \mathbf{Val}$) Return $\mathbf{next}\, pr'$.
  - ($\mathbf{next}\, pr'$, where $pr' \in \mathbf{CP}$) Return $\mathbf{next}(\mathbf{calc}(oper, pr'))$.
- ($\mathbf{cond}(pr_1, pr_2, pr_3)$, where $pr_1, pr_2, pr_3 \in \mathbf{CP}$) Use case analysis on the form of $\mathbf{step}\, pr_1$:
  - (**error**) Return **error**.
  - (**value**) Use case analysis on the form of pr_1 (which is a value):
    - ($\mathbf{const}(\mathsf{true})$) Return $\mathbf{next}\, pr_2$.
    - ($\mathbf{const}(\mathsf{false})$) Return $\mathbf{next}\, pr_3$.

    - $*$ (anything else) Return **error**.
  - (**next** pr_1', where $pr_1' \in$ **CP**) Return **next**(**cond**(pr_1', pr_2, pr_3)).

- (**lam**(v, pr), where $v \in$ **Var**, $pr \in$ **Prog** and **free** $pr \subseteq \{v\}$) Return **value**.

- (**letSimp**$(v, pr_1, pr2)$, where $v \in$ **Var**, $pr_1 \in$ **CP** and **free** $pr_2 \subseteq \{v\}$) Use case analysis on the form of **step** pr_1:

  - (**error**) Return **error**.
  - (**value**) Return **next**(**subst**(pr_1, v, pr_2)).
  - (**next** pr_1') Return **letSimp**(v, pr_1', pr_2).

- (**letRec**(v_1, v_2, pr_1, pr_2), where $v_1, v_2 \in$ **Var**, $pr_1, pr_2 \in$ **Prog**, **free** $pr_1 \subseteq \{v_1, v_2\}$ and **free** $pr_2 \subseteq \{v_1\}$) Let $pr \in$ **CP** be

  $$\textbf{subst}(\textbf{letRec}(v_1, v_2, pr_1, \textbf{var}(v_1)), v_1, \textbf{lam}(v_2, pr_1)).$$

  Return **next**(**subst**(pr, v_1, pr_2)).

The module `Prog` defines the following datatype and function, which correspond to the set **Step** and function **step**:

```
datatype step = Value
              | Error
              | Next of cp
val step : cp -> step
```

For example, here is an example of how a nonterminating recursive function can be evaluated:

```
- val pr = Prog.input "";
@ letRec(x, y,
@        app(var(x), calc(plus, pair(var(y), int(1)))),
@        app(var(x), int(0)))
@ .
val pr = - : prog
- val pr1 = Prog.toClosed pr;
val pr1 = - : cp
- Prog.step pr1;
val it = Next - : Prog.step
- val Prog.Next pr2 = it;
val pr2 = - : cp
- Prog.output("", Prog.fromClosed pr2);
app(lam(y,
        app(letRec(x, y,
                   app(var(x), calc(plus, pair(var(y), int(1)))),
                   var(x)),
```

```
            calc(plus, pair(var(y), int(1)))),
    int(0))
val it = () : unit
- Prog.step pr2;
val it = Next - : Prog.step
- val Prog.Next pr3 = it;
val pr3 = - : cp
- Prog.output("", Prog.fromClosed pr3);
app(letRec(x, y, app(var(x), calc(plus, pair(var(y), int(1)))),
           var(x)),
    calc(plus, pair(int(0), int(1))))
val it = () : unit
- Prog.step pr3;
val it = Next - : Prog.step
- val Prog.Next pr4 = it;
val pr4 = - : cp
- Prog.output("", Prog.fromClosed pr4);
app(lam(y,
        app(letRec(x, y,
                   app(var(x), calc(plus, pair(var(y), int(1)))),
                   var(x)),
            calc(plus, pair(var(y), int(1)))),
    calc(plus, pair(int(0), int(1))))
val it = () : unit
- Prog.step pr4;
val it = Next - : Prog.step
- val Prog.Next pr5 = it;
val pr5 = - : cp
- Prog.output("", Prog.fromClosed pr5);
app(lam(y,
        app(letRec(x, y,
                   app(var(x), calc(plus, pair(var(y), int(1)))),
                   var(x)),
            calc(plus, pair(var(y), int(1)))),
    int(1))
val it = () : unit
```

Proposition 5.1.1
For all $pr \in \mathbf{CP}$, $\mathbf{step}\, pr = \mathbf{value}$ *iff* $pr \in \mathbf{Val}$.

Let

$$\begin{aligned}\mathbf{Run} = &\{\, \mathbf{ans}\, pr \mid pr \in \mathbf{Val} \,\} \\ &\cup \{\, \mathbf{fail}\, pr \mid pr \in \mathbf{CP} \,\} \\ &\cup \{\, \mathbf{intermed}\, pr \mid pr \in \mathbf{CP} \,\},\end{aligned}$$

and define a function $\mathbf{run} \in \mathbf{CP} \times \mathbb{N} \to \mathbf{Run}$ by structural recursion on its second argument. Given $pr \in \mathbf{CP}$ and $n \in \mathbb{N}$, $\mathbf{run}(pr, n)$ is computed using a case analysis of the form of n:

- (0) Return **intermed** pr.
- (n+1, where $n \in \mathbb{N}$) We proceed by a case analysis of the form of **step** pr:
  - (**error**) Return **fail** pr.
  - (**value**) Return **ans** pr.
  - (**next** pr', where $pr' \in$ **CP**) Return **run**(pr', n).

The module `Prog` defines the following datatype and functions:

```
datatype run = Ans      of cp
             | Fail     of cp
             | Intermed of cp
val run      : cp * int -> run
val evaluate : prog * int -> unit
```

The function `run` corresponds to **run**, except that it issues an error message when its second argument is negative. The function `evaluate` issues an error message if its first argument, pr, isn't closed, or its second argument, n, is negative. Otherwise, `evaluate` explains what results from running

`run`(`Prog.toClosed` pr, n).

Here are some examples of how `run` can be used:

```
- val pr = Prog.toClosed(Prog.input "");
@ app(lam(x, calc(plus, pair(var(x), int(3)))),
@     int(4))
@ .
val pr = - : cp
- Prog.run(pr, 0);
val it = Intermed - : Prog.run
- val Prog.Intermed pr' = it;
val pr' = - : cp
- Prog.output("", Prog.fromClosed pr');
app(lam(x, calc(plus, pair(var(x), int(3)))), int(4))
val it = () : unit
- Prog.run(pr, 1);
val it = Intermed - : Prog.run
- val Prog.Intermed pr' = it;
val pr' = - : cp
- Prog.output("", Prog.fromClosed pr');
calc(plus, pair(int(4), int(3)))
val it = () : unit
- Prog.run(pr, 2);
val it = Intermed - : Prog.run
- val Prog.Intermed pr' = it;
val pr' = - : cp
- Prog.output("", Prog.fromClosed pr');
```

```
int(7)
val it = () : unit
- Prog.run(pr, 3);
val it = Ans - : Prog.run
- val Prog.Ans pr' = it;
val pr' = - : cp
- Prog.output("", Prog.fromClosed pr');
int(7)
val it = () : unit
- val pr = Prog.toClosed(Prog.fromString "calc(plus, int(4))");
val pr = - : cp
- Prog.run(pr, 1);
val it = Fail - : Prog.run
- val Prog.Fail pr' = it;
val pr' = - : cp
- Prog.output("", Prog.fromClosed pr');
calc(plus, int(4))
val it = () : unit
```

Suppose that the file `even-prog` contains the following definition of a function for testing whether a natural number is even:

```
letRec(even, n,
       cond(calc(isZero, var(n)), const(true),
            letSimp(m, calc(minus, pair(var(n), int(1))),
                    cond(calc(isZero, var(m)), const(false),
                         app(var(even),
                             calc(minus,
                                  pair(var(m), int(1)))))))),
       var(even))
```

Here are some examples of how we can test this function using `evaluate`:

```
- val even = Prog.input "even-prog";
val even = - : prog
- fun test(n, m) =
=       let val pr = Prog.app(even, Prog.int n)
=       in Prog.evaluate(pr, m) end;
val test = fn : IntInf.int * int -> unit
- test(3, 100);
terminated with value "const(false)"
val it = () : unit
- test(4, 100);
terminated with value "const(true)"
val it = () : unit
- test(5, 100);
terminated with value "const(false)"
val it = () : unit
- test(6, 100);
terminated with value "const(true)"
```

```
val it = () : unit
```

And here are some other uses of `evaluate`:

```
- val pr1 = Prog.input "";
@ calc(consSym, int(12))
@ .
val pr1 = - : prog
- Prog.evaluate(pr1, 2);
terminated with value "sym(b)"
val it = () : unit
- val pr2 = Prog.input "";
@ calc(consSym,
@      pair(sym(9),
@           pair(const(nil),
@                pair(sym(<hi>), const(nil)))))
@ .
val pr2 = - : prog
- Prog.evaluate(pr2, 2);
terminated with value "sym(<9,<hi>>)"
val it = () : unit
- val pr3 = Prog.input "";
@ calc(deconsSym, sym(c))
@ .
val pr3 = - : prog
- Prog.evaluate(pr3, 2);
terminated with value "int(13)"
val it = () : unit
- val pr4 = Prog.input "";
@ calc(deconsSym, sym(<on,,to>))
@ .
val pr4 = - : prog
- Prog.evaluate(pr4, 2);
terminated with value
"pair(sym(o),
      pair(sym(n),
           pair(const(nil),
                pair(const(nil),
                     pair(sym(t), pair(sym(o), const(nil)))))))"
val it = () : unit
- val pr5 = Prog.input "";
@ calc(symListToStr,
@      pair(sym(9),
@           pair(sym(<hi>),
@                const(nil))))
@ .
val pr5 = - : prog
- Prog.evaluate(pr5, 2);
terminated with value "str(9<hi>)"
val it = () : unit
```

```
- val pr6 = Prog.input "";
@ calc(strToSymList, str(ab<hi><>))
@ .
val pr6 = - : prog
- Prog.evaluate(pr6, 2);
terminated with value
"pair(sym(a),
      pair(sym(b), pair(sym(<hi>), pair(sym(<>), const(nil)))))"
val it = () : unit
```

Proposition 5.1.2

1. *For all* $pr \in \mathbf{CP}$, $n \in \mathbb{N}$ *and* $pr' \in \mathbf{Val}$, *if* $\mathbf{run}(pr, n) = \mathbf{ans}\, pr'$, *then, for all* $m \in \mathbb{N}$, *if* $m \geq n$, *then* $\mathbf{run}(pr, m) = \mathbf{ans}\, pr'$.

2. *For all* $pr \in \mathbf{CP}$, $n \in \mathbb{N}$ *and* $pr' \in \mathbf{CP}$, *if* $\mathbf{run}(pr, n) = \mathbf{fail}\, pr'$, *then, for all* $m \in \mathbb{N}$, *if* $m \geq n$, *then* $\mathbf{run}(pr, m) = \mathbf{fail}\, pr'$.

Now we can define the mathematical meaning of closed programs. Let

$$\mathbf{Eval} = \{\mathbf{nonterm}, \mathbf{error}\} \cup \{\, \mathbf{norm}\, pr \mid pr \in \mathbf{Val} \,\}.$$

We define a mathematical function—*not* an algorithm—$\mathbf{eval} \in \mathbf{CP} \to \mathbf{Eval}$, as follows. Suppose $pr \in \mathbf{CP}$. There are two main cases to consider:

- Suppose, for all $n \in \mathbb{N}$, there is a $pr' \in \mathbf{CP}$ such that $\mathbf{run}(pr, n) = \mathbf{intermed}\, pr'$. Then $\mathbf{eval}\, pr = \mathbf{nonterm}$.
- Suppose there is an $n \in \mathbb{N}$ such that there is no $pr' \in \mathbf{CP}$ such that $\mathbf{run}(pr, n) = \mathbf{intermed}\, pr'$. Let n be the smallest natural number such that there is no $pr' \in \mathbf{CP}$ such that $\mathbf{run}(pr, n) = \mathbf{intermed}\, pr'$. There are two subcases to consider:
  - Suppose $\mathbf{run}(pr, n) = \mathbf{ans}\, pr'$ for some $pr' \in \mathbf{Val}$. Then $\mathbf{eval}\, pr = \mathbf{norm}\, pr'$.
  - Suppose $\mathbf{run}(pr, n) = \mathbf{fail}\, pr'$ for some $pr' \in \mathbf{CP}$. Then $\mathbf{eval}\, pr = \mathbf{error}$.

For example:

- Let pr be

  $$\mathbf{letRec}(\mathsf{x}, \mathsf{y}, \mathbf{app}(\mathbf{var}(\mathsf{x}), \mathbf{var}(\mathsf{y})), \mathbf{app}(\mathbf{var}(\mathsf{x}), \mathbf{int}(0))).$$

  Then $\mathbf{eval}\, pr = \mathbf{nonterm}$.
- $\mathbf{eval}(\mathbf{app}(\mathbf{int}(0), \mathbf{int}(1))) = \mathbf{error}$.
- $\mathbf{eval}(\mathbf{calc}(\mathsf{plus}, \mathbf{pair}(\mathbf{int}(1), \mathbf{int}(2)))) = \mathbf{norm}(\mathbf{int}(3))$.

5.1.3 Programs as Data

In Section 5.3, we will be concerned with programs that process programs. Because programs are described by strings, these program processing programs could work on strings. But that would be complicated and cumbersome. It's far better to represent programs using pairs.

The set **Rep** of *program representations* is the least subset of **Val** such that:

(variable) for all $v \in \mathbf{Var}$, $\mathbf{pair}(\mathbf{str}(\mathsf{var}), \mathbf{str}(v)) \in \mathbf{Rep}$;

(constant) for all $con \in \mathbf{Const}$, $\mathbf{pair}(\mathbf{str}(\mathsf{const}), \mathbf{const}(con)) \in \mathbf{Rep}$;

(integer) for all $n \in \mathbb{Z}$, $\mathbf{pair}(\mathbf{str}(\mathsf{int}), \mathbf{int}(n)) \in \mathbf{Rep}$;

(symbol) for all $a \in \mathbf{Sym}$, $\mathbf{pair}(\mathbf{str}(\mathsf{sym}), \mathbf{sym}(a)) \in \mathbf{Rep}$;

(string) for all $x \in \mathbf{Str}$, $\mathbf{pair}(\mathbf{str}(\mathsf{str}), \mathbf{str}(x)) \in \mathbf{Rep}$;

(pair) for all $pr_1, pr_2 \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{pair}), \mathbf{pair}(pr_1, pr_2)) \in \mathbf{Rep};$$

(calculation) for all $oper \in \mathbf{Oper}$ and $pr \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{calc}), \mathbf{pair}(\mathbf{str}(oper), pr)) \in \mathbf{Rep};$$

(conditional) for all $pr_1, pr_2, pr_3 \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{cond}), \mathbf{pair}(pr_1, \mathbf{pair}(pr_2, pr_3))) \in \mathbf{Rep};$$

(function application) for all $pr_1, pr_2 \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{app}), \mathbf{pair}(pr_1, pr_2)) \in \mathbf{Rep};$$

(anonymous function) for all $v \in \mathbf{Var}$ and $pr \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{lam}), \mathbf{pair}(\mathbf{str}(v), pr)) \in \mathbf{Rep};$$

(simple let) for all $v \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{letSimp}), \mathbf{pair}(\mathbf{str}(v), \mathbf{pair}(pr_1, pr_2))) \in \mathbf{Rep};$$

(recursive let) for all $v_1, v_2 \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Rep}$,

$$\mathbf{pair}(\mathbf{str}(\mathsf{letRec}), \mathbf{pair}(\mathbf{str}(v_1), \mathbf{pair}(\mathbf{str}(v_2), \mathbf{pair}(pr_1, pr_2)))) \in \mathbf{Rep}.$$

We define a function $\overline{\cdot} \in \mathbf{Prog} \to \mathbf{Rep}$ by structural recursion:

- for all $v \in \mathbf{Var}$, $\overline{\mathbf{var}(v)} = \mathbf{pair}(\mathbf{str}(\mathsf{var}), \mathbf{str}(v))$;
- for all $con \in \mathbf{Const}$, $\overline{\mathbf{const}(con)} = \mathbf{pair}(\mathbf{str}(\mathsf{const}), \mathbf{const}(con))$;
- for all $n \in \mathbb{Z}$, $\overline{\mathbf{int}(n)} = \mathbf{pair}(\mathbf{str}(\mathsf{int}), \mathbf{int}(n))$;
- for all $a \in \mathbf{Sym}$, $\overline{\mathbf{sym}(a)} = \mathbf{pair}(\mathbf{str}(\mathsf{sym}), \mathbf{sym}(a))$;
- for all $x \in \mathbf{Str}$, $\overline{\mathbf{str}(x)} = \mathbf{pair}(\mathbf{str}(\mathsf{str}), \mathbf{str}(x))$;
- for all $pr_1, pr_2 \in \mathbf{Prog}$,

$$\overline{\mathbf{pair}(pr_1, pr_2)} = \mathbf{pair}(\mathbf{str}(\mathsf{pair}), \mathbf{pair}(\overline{pr_1}, \overline{pr_2}));$$

- for all $oper \in \mathbf{Oper}$ and $pr \in \mathbf{Prog}$,

$$\overline{\mathbf{calc}(oper, pr)} = \mathbf{pair}(\mathbf{str}(\mathsf{calc}), \mathbf{pair}(\mathbf{str}(oper), \overline{pr}));$$

- for all $pr_1, pr_2, pr_3 \in \mathbf{Prog}$,

$$\overline{\mathbf{cond}(pr_1, pr_2, pr_3)} = \mathbf{pair}(\mathbf{str}(\mathsf{cond}), \mathbf{pair}(\overline{pr_1}, \mathbf{pair}(\overline{pr_2}, \overline{pr_3})));$$

- for all $pr_1, pr_2 \in \mathbf{Prog}$,

$$\overline{\mathbf{app}(pr_1, pr_2)} = \mathbf{pair}(\mathbf{str}(\mathsf{app}), \mathbf{pair}(\overline{pr_1}, \overline{pr_2}));$$

- for all $v \in \mathbf{Var}$ and $pr \in \mathbf{Prog}$,

$$\overline{\mathbf{lam}(v, pr)} = \mathbf{pair}(\mathbf{str}(\mathsf{lam}), \mathbf{pair}(\mathbf{str}(v), \overline{pr}));$$

- for all $v \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\overline{\mathbf{letSimp}(v, pr_1, pr_2)} = \mathbf{pair}(\mathbf{str}(\mathsf{letSimp}), \mathbf{pair}(\mathbf{str}(v), \mathbf{pair}(\overline{pr_1}, \overline{pr_2})));$$

- for all $v_1, v_2 \in \mathbf{Var}$ and $pr_1, pr_2 \in \mathbf{Prog}$,

$$\overline{\mathbf{letRec}(v_1, v_2, pr_1, pr_2)} = \mathbf{pair}(\mathbf{str}(\mathsf{letRec}), \mathbf{pair}(\mathbf{str}(v_1), \mathbf{pair}(\mathbf{str}(v_2), \mathbf{pair}(\overline{pr_1}, \overline{pr_2})))).$$

If $pr \in \mathbf{Prog}$, we say that $\overline{pr}$ *represents* pr, and that pr *is represented by* $\overline{pr}$.

Each program is represented by a unique program representation, and every program representation represents a unique program:

Proposition 5.1.3
$\overline{\cdot}$ *is a bijection from* **Prog** *to* **Rep**.

The module `Prog` also defines the functions:

```
val toRep   : prog -> prog
val fromRep : prog -> prog
val isRep   : prog -> bool
```

The function `toRep` converts a program to the program representation that represents it. The function `fromRep` issues an error message if its argument isn't a program representation; otherwise it returns the program represented by its argument. And the function `isRep` tests whether a program is a program representation.

For example:

```
- val pr = Prog.input "";
@ app(lam(x, app(var(x), int(1))),
@     lam(y, var(y)))
@ .
val pr = - : prog
- val pr' = Prog.toRep pr;
val pr' = - : prog
- Prog.output("", pr');
pair(str(app),
     pair(pair(str(lam),
               pair(str(x),
                    pair(str(app),
                         pair(pair(str(var), str(x)),
                              pair(str(int), int(1))))))),
          pair(str(lam), pair(str(y), pair(str(var), str(y))))))
val it = () : unit
- val pr'' = Prog.fromRep pr';
val pr'' = - : prog
- Prog.equal(pr'', pr);
val it = true : bool
- Prog.isRep pr;
val it = false : bool
- Prog.isRep pr';
val it = true : bool
- Prog.isRep(Prog.fromString "pair(str(x), str(y))");
val it = false : bool
```

It is easy to write a program that tests whether a program representation represents a closed program, as well to write a program that tests whether a program representation represents a value.

It is possible to write a function in our programming language that acts as an interpreter:

- It takes in a value $\overline{pr}$, representing a closed program pr.
- It begins evaluating pr, using the representation $\overline{pr}$.
- If this evaluation results in an error, then the interpreter returns **const**(nil).

- Otherwise, if it results in a value $\overline{pr'}$ representing a value pr', then it returns $\overline{pr'}$.
- Otherwise, it runs forever.

E.g., $\overline{\mathbf{cond}(\mathbf{const}(\mathsf{true}), \mathbf{const}(\mathsf{false}), \mathbf{const}(\mathsf{nil}))}$ evaluates to $\overline{\mathbf{const}(\mathsf{false})}$.

We can also write a function in our programming language that acts as an *incremental* interpreter:

- At each stage of its evaluation of a closed program, it carries out some fixed number of steps of the evaluation.
- If during the execution of those steps, an error is detected, then it returns $\mathbf{const}(\mathsf{nil})$.
- Otherwise, if a value $\overline{pr'}$ representing a value pr' has been produced, then it returns this value.
- But otherwise, it returns an anonymous function that when called will continue this process.

5.1.4 Recursive and Recursively Enumerable Languages

A *string predicate program* pr is a closed program such that, for all strings w, $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \in \{\mathbf{norm}(\mathbf{const}(\mathsf{true})), \mathbf{norm}(\mathbf{const}(\mathsf{false}))\}$.

A string w is *accepted by* a closed program pr iff $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. We write $L(pr)$ for the set of all strings accepted by a closed program pr. When this set is a language, then we refer to $L(pr)$ as the *language accepted by* pr. (E.g., if $pr = \mathbf{lam}(\mathsf{x}, \mathbf{const}(\mathsf{true}))$, then $L(pr) = \mathbf{Str}$, and so is not a language.)

The `Prog` module also includes:

```
val accepted : prog -> str * int -> unit
```

The function `accepted` takes in a program pr, and issues and error message if pr is not closed. Otherwise, it returns a function f that behaves as follows, when called with a pair (x, n). If n is negative, it issues an error message. Otherwise, it proceeds by a case analysis of the result of running

$$pr' = \texttt{Prog.run}(\texttt{Prog.app}(pr,\ \texttt{Prog.str}\ x), n)\ :$$

- $(\mathbf{ans}(\mathbf{const}(\mathsf{true})))$ It explains that x was accepted by pr.
- $(\mathbf{ans}(\mathbf{const}(\mathsf{false})))$ It explains that x was rejected by pr, because the application of pr to $\mathbf{str}(x)$ resulted in $\mathbf{const}(\mathsf{false})$.
- $(\mathbf{ans}\ pr''$, where $pr'' \notin \{\mathbf{const}(\mathsf{true}), \mathbf{const}(\mathsf{false})\})$ It explains that x was rejected by pr, since the application of pr to $\mathbf{str}(x)$ resulted in some value other than $\mathbf{const}(\mathsf{true})$ or $\mathbf{const}(\mathsf{false})$.

- (**fail** pr'', where $pr'' \in \mathbf{CP}$) It explains that x was rejected by pr, since the application of pr to $\mathbf{str}(x)$ resulted in failure.
- (**intermed** pr'', where $pr'' \in \mathbf{CP}$) It explains that (based on running pr' for n steps) it is unknown whether x is accepted by pr.

Suppose the file `equal-prog` contains the text:

```
lam(p,
    calc(isZero,
         calc(compare, var(p))))
```

Suppose the file `succ-prog` contains the text:

```
lam(p,
    calc(isZero,
         calc(compare, var(p))))
```

Suppose the file `count-prog` contains the text:

```
lam(equal, lam(succ, lam(a,
  letRec(f, xs,
         cond(calc(isNil, var(xs)),
              pair(int(0), const(nil)),
              cond(app(var(equal),
                       pair(calc(fst, var(xs)), var(a))),
                   letSimp(res,
                           app(var(f),
                               calc(snd, var(xs))),
                           pair(app(var(succ),
                                    calc(fst, var(res))),
                                calc(snd, var(res)))),
                   pair(int(0), var(xs)))),
         var(f)))))
```

And suppose the file `zeros-ones-twos-prog` contains the text:

```
lam(equal, lam(count, lam(x,
  letSimp(xs, calc(strToSymList, var(x)),
  letSimp(zeros,
          app(app(var(count), sym(0)), var(xs)),
  letSimp(ones,
          app(app(var(count), sym(1)), calc(snd, var(zeros))),
  letSimp(twos,
          app(app(var(count), sym(2)), calc(snd, var(ones))),
    cond(calc(isNil, calc(snd, var(twos))),
         cond(app(var(equal),
                  pair(calc(fst, var(zeros)),
                       calc(fst, var(ones)))),
              app(var(equal),
                  pair(calc(fst, var(ones)),
```

```
                        calc(fst, var(twos)))),
                const(false)),
            const(false)))))))))
```

We can construct—and experiment with—a program for testing whether a string is an element of $\{\, 0^n 1^n 2^n \mid n \in \mathbb{N} \,\}$, as follows:

```
- val equal = Prog.input "equal-prog";
val equal = - : prog
- val succ = Prog.input "succ-prog";
val succ = - : prog
- val count =
=       Prog.app
=       (Prog.app(Prog.input "count-prog", equal),
=        succ);
val count = - : prog
- val zerosOnesTwos =
=       Prog.app
=       (Prog.app(Prog.input "zeros-ones-twos-prog", equal),
=        count);
val zerosOnesTwos = - : prog
- val accepted = Prog.accepted zerosOnesTwos;
val accepted = fn : str * int -> unit
- accepted(Str.fromString "%", 10);
unknown if accepted or rejected
val it = () : unit
- accepted(Str.fromString "%", 100);
accepted
val it = () : unit
- accepted(Str.fromString "012", 100);
accepted
val it = () : unit
- accepted(Str.fromString "000111222", 1000);
accepted
val it = () : unit
- accepted(Str.fromString "00011122", 1000);
rejected with false
val it = () : unit
- accepted(Str.fromString "00111222", 1000);
rejected with false
val it = () : unit
- accepted(Str.fromString "00011222", 1000);
rejected with false
val it = () : unit
- accepted(Str.fromString "021", 100);
rejected with false
val it = () : unit
- accepted(Str.fromString "112200", 1000);
rejected with false
```

```
val it = () : unit
```

We say that a language L is:

- *recursive* iff $L = L(pr)$, for some string predicate program pr; and
- *recursively enumerable* (*r.e.*) iff $L = L(pr)$, for some closed program pr.

We define

$$\mathbf{RecLan} = \{\, L \in \mathbf{Lan} \mid L \text{ is recursive} \,\}, \text{ and}$$
$$\mathbf{RELan} = \{\, L \in \mathbf{Lan} \mid L \text{ is recursively enumerable} \,\}.$$

Hence $\mathbf{RecLan} \subseteq \mathbf{RELan}$. Because $\mathbf{CP}$ is countably infinite, we have that $\mathbf{RecLan}$ and $\mathbf{RELan}$ are countably infinite, so that $\mathbf{RELan} \subsetneq \mathbf{Lan}$. Later we will see that $\mathbf{RecLan} \subsetneq \mathbf{RELan}$.

Proposition 5.1.4
For all $L \in \mathbf{Lan}$, L is recursive iff there is a closed program pr such that, for all $w \in \mathbf{Str}$:

- *if $w \in L$, then* $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$*; and*
- *if $w \notin L$, then* $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{false}))$.

Proof.

("only if") Since L is recursive, $L = L(pr)$ for some string predicate program pr. Suppose $w \in \mathbf{Str}$. There are two cases to show.

- Suppose $w \in L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.
- Suppose $w \notin L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \neq \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. But pr is a string predicate program, and thus $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{false}))$.

("if") To see that pr is a string predicate program, suppose $w \in \mathbf{Str}$. Since $w \in L$ or $w \notin L$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \in \{\mathbf{const}(\mathsf{true}), \mathbf{const}(\mathsf{false})\}$. We will show that $L = L(pr)$.

- Suppose $w \in L$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, so that $w \in L(pr)$.
- Suppose $w \in L(pr)$, so that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. If $w \notin L$, then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{false}))$—contradiction. Thus $w \in L$.

□

Proposition 5.1.5
*For all $L \in$ **Lan**, L is recursively enumerable iff there is a closed program pr such that, for all $w \in$ **Str**,*

$$w \in L \text{ iff } \mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true})).$$

Proof.

("only if") Since L is recursively enumerable, $L = L(pr)$ for some closed program pr. Suppose $w \in$ **Str**.

- Suppose $w \in L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.
- Suppose $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. Thus $w \in L(pr) = L$.

("if") It suffices to show that $L = L(pr)$.

- Suppose $w \in L$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, so that $w \in L(pr)$.
- Suppose $w \in L(pr)$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, so that $w \in L$.

□

Theorem 5.1.6
The context-free languages are a proper subset of the recursive languages: **CFLan** $\subsetneq$ **RecLan**.

Proof. 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 string predicate program pr 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 in the preceding subsection, we wrote a string predicate program pr that tests whether a string is in L. Thus L is recursive. □

5.1.5 Notes

Neil Jones [Jon97] pioneered the use of a programming language with structured data as an alternative to Turing machines for studying the limits of what is computable. In contrast to Jones's approach, however, our programming language is functional, not imperative (assignment-oriented), and it has explicit support for the symbols and strings of formal language theory.

5.2 Closure Properties of Recursive and R.E. 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.

5.2.1 Closure Properties of Recursive Languages

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 string predicate programs pr_1 and pr_2 that test whether strings are in L_1 and L_2, respectively. We write a program pr with form

$$\begin{array}{l}\textbf{lam}(\mathsf{w},\\ \quad \textbf{letSimp}(\mathsf{f1},\\ \qquad pr_1,\\ \qquad \textbf{letSimp}(\mathsf{f2},\\ \qquad\quad pr_2,\\ \qquad\quad \cdots))),\end{array}$$

which tests whether its input is an element of L_1L_2.

The elided part of pr generates all of the pairs of strings (x_1, x_2) such that x_1x_2 is equal to the value of w. Then it works though these pairs, one by one. Given such a pair (x_1, x_2), pr calls $\mathsf{f1}$ with x_1 to check whether $x_1 \in L_1$. If the answer is $\textbf{const}(\mathsf{false})$, then it goes on to the next pair. Otherwise, it calls $\mathsf{f2}$ with x_2 to check whether $x_2 \in L_2$. If the answer is $\textbf{const}(\mathsf{false})$, then it goes on to the next pair. Otherwise, it returns $\textbf{const}(\mathsf{true})$. If pr runs out of pairs to check, then it returns $\textbf{const}(\mathsf{false})$.

We can check that pr is a string predicate program testing whether $w \in L_1L_2$. Thus L_1L_2 is recursive. □

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. □

5.2.2 Closure Properties of Recursively Enumerable Languages

Theorem 5.2.3
If L, L_1 and L_2 are recursively enumerable languages, then so are $L_1 \cup L_2$, $L_1 L_2$, 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 pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L_1$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, and for all $w \in \mathbf{Str}$, $w \in L_2$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. (Remember that pr_1 and pr_2 may fail to terminate on some inputs.)

To show that $L_1 L_2$ is recursively enumerable, we will construct a program pr such that, for all $w \in \mathbf{Str}$, $w \in L_1 L_2$ iff $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.

When pr is called with $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, it behaves as follows. First, it generates all the pairs of strings (x_1, x_2) such that $w = x_1 x_2$. Let these pairs be $(x_{1,1}, x_{2,1}), \ldots, (x_{1,n}, x_{2,n})$. Now, pr uses our *incremental* interpretation function to run a fixed number of steps of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ and $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ (working with $\overline{\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))}$ and $\overline{\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))}$), for all $i \in [1 : n]$, and then repeat this over and over again.

- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ returns $\overline{\mathbf{const}(\mathsf{true})}$, then $x_{1,i}$ is marked as being in L_1.
- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ returns $\overline{\mathbf{const}(\mathsf{true})}$, then the $x_{2,i}$ is marked as being in L_2.
- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ returns something other than $\overline{\mathbf{const}(\mathsf{true})}$, then the i'th pair is marked as discarded.
- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ returns something other than $\overline{\mathbf{const}(\mathsf{true})}$, then the i'th pair is marked as discarded.
- If, at some stage, $x_{1,i}$ is marked as in L_1 and $x_{2,i}$ is marked as in L_2, then Q returns $\mathbf{const}(\mathsf{true})$.
- If, at some stage, there are no remaining pairs, then pr returns $\mathbf{const}(\mathsf{false})$.

□

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 pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, and for all $w \in \mathbf{Str}$, $w \in \Sigma^* - L$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.

We construct a string predicate program pr that tests whether its input is in L. Given $\mathbf{str}(w)$, for $w \in \mathbf{Str}$, pr proceeds as follows. If $w \notin \Sigma^*$, then pr returns $\mathbf{const}(\mathsf{false})$. Otherwise, pr alternates between incrementally interpreting $\mathbf{app}(pr_1, \mathbf{str}(w))$ (working with $\overline{\mathbf{app}(pr_1, \mathbf{str}(w))}$) and $\mathbf{app}(pr_2, \mathbf{str}(w))$ (working with $\overline{\mathbf{app}(pr_2, \mathbf{str}(w))}$).

- If, at some stage, the first incremental interpretation returns $\overline{\mathbf{const}(\mathsf{true})}$, then pr returns $\mathbf{const}(\mathsf{true})$.
- If, at some stage, the second incremental interpretation returns $\mathbf{const}(\mathsf{true})$, then pr returns $\mathbf{const}(\mathsf{false})$.
- If, at some stage, the first incremental interpretation returns anything other than $\mathbf{const}(\mathsf{true})$, then pr returns $\mathbf{const}(\mathsf{false})$.
- If, at some stage, the second incremental interpretation returns anything other than $\mathbf{const}(\mathsf{true})$, then pr returns $\mathbf{const}(\mathsf{true})$.

□

5.2.3 Notes

Our approach to this section is standard, except for our using programs instead of Turing machines.

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.

5.3.1 Diagonalization

To find a non-r.e. language, we can use diagonalization, a technique we used in Section 1.1 to show the uncountability of $\mathcal{P}\,\mathbb{N}$.

Let Σ be the alphabet used to describe programs: the digits and letters, plus the elements of $\{\langle\mathsf{comma}\rangle, \langle\mathsf{perc}\rangle, \langle\mathsf{tilde}\rangle, \langle\mathsf{openPar}\rangle, \langle\mathsf{closPar}\rangle, \langle\mathsf{less}\rangle, \langle\mathsf{great}\rangle\}$. Every element of Σ^* either describes a unique closed program, or describes no closed programs. Given $w \in \Sigma^*$, we write $L(w)$ for:

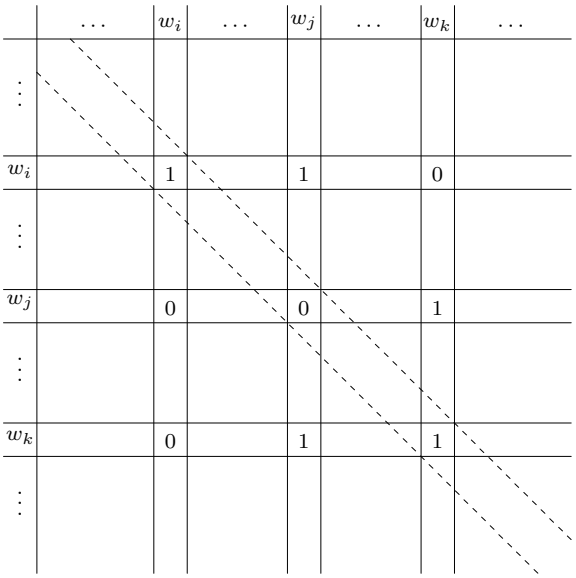


Figure 5.1: Example Diagonalization Table

- $\emptyset$, if w doesn't describe a closed program; and
- $L(pr)$, where pr is the unique closed program described by w, if w does describe a closed program.

Thus $L(w)$ will always be a set of strings, even though it won't always be a language.

Consider the infinite table of 0's and 1's in which both the rows and the columns are indexed by the elements of Σ^*, listed in ascending order according to our standard total ordering, and where a cell (w_n, w_m) contains 1 iff $w_n \in L(w_m)$, and contains 0 iff $w_n \notin L(w_m)$. Each recursively enumerable language is $L(w_n)$ for some (non-unique) n, but not all the $L(w_n)$ are languages. Figure 5.1 shows how part of this table might look, where w_i, w_j and w_k are sample elements of Σ^*. Because of the table's data, we have that $w_i \in L(w_j)$ and $w_j \notin L(w_i)$.

To define a non-r.e. Σ-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 $w_n \notin L(w_n)$. With our example table:

- $L(w_i)$ is not our language, since $w_i \in L(w_i)$, but w_i is not in our language;
- $L(w_j)$ is not our language, since $w_j \notin L(w_j)$, but w_j is in our language; and
- $L(w_k)$ is not our language, since $w_k \in L(w_k)$, but w_k is not in our language.

In general, there is no $n \in \mathbb{N}$ such that $L(w_n)$ is our language. Consequently our language is not recursively enumerable.

We formalize the above ideas as follows. Define languages L_d ("d" for "diagonal") and L_a ("a" for "accepted") by:

$$L_d = \{\, w \in \Sigma^* \mid w \notin L(w) \,\}, \text{ and}$$
$$L_a = \{\, w \in \Sigma^* \mid w \in L(w) \,\}.$$

Thus $L_d = \Sigma^* - L_a$. We have that, for all $w \in \Sigma^*$, $w \in L_a$ iff $w \in L(pr)$, for some closed program (which will be unique) described by w. (When w doesn't describe a closed program, $L(w) = \emptyset$.)

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 closed program pr such that $L_d = L(pr)$. Let $w \in \Sigma^*$ be the string describing pr. Thus $L(w) = L(pr) = L_d$.

There are two cases to consider.

- Suppose $w \in L_d$. Then $w \notin L(w) = L_d$—contradiction.
- Suppose $w \notin L_d$. Since $w \in \Sigma^*$, we have that $w \in L(w) = L_d$—contradiction.

Since we obtained a contradiction in both cases, we have an overall contradiction. Thus L_d is not recursively enumerable. □

Theorem 5.3.2
L_a is recursively enumerable.

Proof. Let *acc* be the program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows. First, it attempts to parse $\mathbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, *acc* returns $\mathbf{const}(\mathsf{false})$. If pr is not closed, then *acc* returns $\mathbf{const}(\mathsf{false})$. Otherwise, it uses our interpreter function to evaluate $\mathbf{app}(pr, \mathbf{str}(w))$, using $\overline{\mathbf{app}(pr, \mathbf{str}(w))}$. If this interpretation returns $\overline{\mathbf{const}(\mathsf{true})}$, then *acc* returns $\mathbf{const}(\mathsf{true})$. If it returns anything other than $\overline{\mathbf{const}(\mathsf{true})}$, then *acc* returns $\mathbf{const}(\mathsf{false})$. (Thus, if the interpretation never returns, then *acc* never terminates.)

We can check that, for all $w \in \mathbf{Str}$, $w \in L_a$ iff $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. Thus L_a is recursively enumerable. □

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. $L_a \subseteq \Sigma^*$ is recursively enumerable, but $\Sigma^* - L_a = L_d$ is not recursively enumerable. □

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. □

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 $L_a \subseteq \Sigma^*$, we have that $L_d = \Sigma^* - L_a$ is recursive—contradiction. Thus L_a is not recursive. □

Since $L_a \in$ **RELan** and $L_a \notin$ **RecLan**, we have:

Theorem 5.3.6
The recursive languages are a proper subset of the recursively enumerable languages: **RecLan** $\subsetneq$ **RELan**.

Combining this result with results from Sections 4.8 and 5.1, we have that

$$\textbf{RegLan} \subsetneq \textbf{CFLan} \subsetneq \textbf{RecLan} \subsetneq \textbf{RELan} \subsetneq \textbf{Lan}.$$

5.3.2 Undecidability of the Halting Problem

We say that a closed program *pr halts* iff $\textbf{eval}\, pr \neq \textbf{nonterm}$.

Theorem 5.3.7
There is no value halts such that, for all closed programs pr,

- *If pr halts, then* $\textbf{eval}(\textbf{app}(\mathit{halts}, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathsf{true}))$*; and*
- *If pr does not halt, then* $\textbf{eval}(\textbf{app}(\mathit{halts}, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathsf{false}))$.

Proof. Suppose, toward a contradiction, that such a *halts* does exist. We use *halts* to construct a program *acc* that behaves as follows when run on $\textbf{str}(w)$, for some $w \in \textbf{Str}$. First, it attempts to parse $\textbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, it returns $\textbf{const}(\mathsf{false})$. If pr is not closed, then it returns $\textbf{const}(\mathsf{false})$. Otherwise, it calls *halts* with argument $\overline{\textbf{app}(pr, \textbf{str}(w))}$.

- If *halts* returns $\textbf{const}(\mathsf{true})$ (so we know that $\underline{\textbf{app}(pr, \textbf{str}(w))}$ halts), then *acc* applies the interpreter function to $\underline{\textbf{app}(pr, \textbf{str}(w))}$, using it to evaluate $\textbf{app}(pr, \textbf{str}(w))$. If the interpreter returns $\overline{\textbf{const}(\mathsf{true})}$, then *acc* returns $\textbf{const}(\mathsf{true})$. Otherwise, the interpreter returns some other value, and *acc* returns $\textbf{const}(\mathsf{false})$.

- Otherwise, *halts* returns **const**(false) (so we know that **app**(pr, **str**(w)) does not halt), in which case *acc* returns **const**(false).

Now, we prove that *acc* is a string predicate program testing whether a string is in L_a.

- Suppose $w \in L_a$. Thus $w \in L(pr)$, where pr is the unique closed program described by w. Hence **eval**(**app**(pr, **str**(w))) = **norm**(**const**(true)). It is easy to show that **eval**(**app**(acc, **str**(w))) = **norm**(**const**(true)).
- Suppose $w \notin L_a$. If $w \notin \Sigma^*$, or $w \in \Sigma^*$ but w does not describe a program, or w describes a program that isn't closed, then **eval**(**app**(acc, **str**(w))) = **norm**(**const**(false)). So, suppose w describes the closed program pr. Then $w \notin L(pr)$, i.e., **eval**(**app**(pr, **str**(w))) $\neq$ **norm**(**const**(true)). It is easy to show that **eval**(**app**(acc, **str**(w))) = **norm**(**const**(false)).

Thus L_a is recursive—contradiction. Thus there is no such *halt*. □

We say that a value pr *halts on* a value pr' iff **eval**(**app**(pr, pr')) $\neq$ **nonterm**.

Corollary 5.3.8 (Undecidability of the Halting Problem)
There is no value haltsOn such that, for all values pr and pr′:

- *if pr halts on pr′, then*

$$\textbf{eval}(\textbf{app}(haltsOn, \textbf{pair}(\overline{pr}, \overline{pr'}))) = \textbf{norm}(\textbf{const}(\mathsf{true}));\ \textit{and}$$

- *If pr does not halt on pr′, then*

$$\textbf{eval}(\textbf{app}(haltsOn, \textbf{pair}(\overline{pr}, \overline{pr'}))) = \textbf{norm}(\textbf{const}(\mathsf{false})).$$

Proof. Suppose, toward a contradiction, that such a *haltsOn* exists. Let *halts* be the value that takes in a value $\overline{pr}$ representing a closed program pr, and then calls *haltsOn* with **pair**($\overline{\textbf{lam}(\mathsf{x}, pr)}$, $\overline{\textbf{const}(\mathsf{nil})}$). Then this program satisfies the property of Theorem 5.3.7—contradiction. Thus such a *haltsOn* does not exist. □

5.3.3 Other Undecidable Problems

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.

5.3.4 Notes

Because a closed program can be evaluated by itself, i.e., without being supplied an input, our treatment of the undecidability of the halting problem is nonstandard. First, we prove that there is no program that takes in a program as data and tests whether that program halts. Our version of the undecidability of the halting then follows as a corollary.

Bibliography

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1991.

[BE93] J. Barwise and J. Etchemendy. *Turing's World 3.0 for Mac: An Introduction to Computability Theory*. Cambridge University Press, 1993.

[BLP+97] A. O. Bilska, K. H. Leider, M. Procopiuc, O. Procopiuc, S. H. Rodger, J. R. Salemme, and E. Tsang. A collection of tools for making automata theory and formal languages come alive. In *Twenty-eighth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 15–19. ACM Press, 1997.

[End77] H. B. Enderton. *Elements of Set Theory*. Academic Press, 1977.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, second edition, 2001.

[HR00] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, pages 6–10. ACM Press, 2000.

[Jon97] N. J. Jones. *Computability and Complexity: From a Programming Perspective*. The MIT Press, 1997.

[Koz97] D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1997.

[Lei10] H. Leiß. The Automata Library. `http://www.cis.uni-muenchen.de/~leiss/sml-automata.html`, 2010.

[Lin01] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, 2001.

[LP98] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, second edition, 1998.

[LRGS04] S. Lombardy, Y. Régis-Gianas, and J. Sakarovitch. Introducing vaucanson. *Theoretical Computer Science*, 328:77–96, 2004.

[Mar91] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw Hill, second edition, 1991.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised 1997*. The MIT Press, 1997.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[RHND99] M. B. Robinson, J. A. Hamshar, J. E. Novillo, and A. T. Duchowski. A Java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In *Thirtieth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 105–109. ACM Press, 1999.

[Rod06] S. H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, 2006.

[RW94] D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[RWYC17] D. Raymond, D. Wood, S. Yu, and C. Câmpeanu. Grail+: A symbolic computation environment for finite-state machines, regular expressions, and finite languages. `http://www.csit.upei.ca/~ccampeanu/Grail/`, 2017.

[Sar02] J. Saraiva. HaLeX: A Haskell library to model, manipulate and animate regular languages. In *ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)*, Pittsburgh, October 2002.

[Sto05] A. Stoughton. Experimenting with formal languages. In *Thirty-sixth ACM SIGCSE Technical Symposium on Computer Science Education*, page 566. ACM Press, 2005.

[Sto08] A. Stoughton. Experimenting with formal languages using Forlan. In *FDPE '08: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 41–50, New York, NY, USA, 2008. ACM.

[Sut92] K. Sutner. Implementing finite state machines. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 15, pages 347–363. American Mathematical Society, 1992.

[Ull98] J. D. Ullman. *Elements of ML Programming: ML97 Edition*. Prentice Hall, 1998.

Index