Chapter 2

Preliminaries

Contents

2.1 General Notation 4
2.2 Decision Problems and Algorithms 4
2.2.1 Decision Problems 4
2.2.2 Algorithms 5
2.3 Complexity Classes 6
2.4 Sets, Multisets and Mappings 7
2.4.1 Sets 7
2.4.2 Mappings 8
2.5 Binary Relations and Orders 9
2.5.1 Binary Relations 9
2.5.2 Orders 10
2.6 Graphs, Dags, and Trees 11
2.6.1 Directed Graphs 11
2.6.2 Directed Acyclic Graphs 12
2.6.3 Trees 12
2.7 Inductive Definitions of Expressions and Parse Trees 14
2.7.1 Inductive Definitions, Expressions, and Subexpressions 14
2.7.2 Parse Trees and Precedences 16
2.7.3 Positions 17
2.8 Rewrite Rule Systems 18
Exercises 21

In this chapter we

(1) define notation used throughout this book, for example notation for equality, sets, mappings, and algorithms.

(2) introduce general mathematical concepts, such as binary relations and systems of rewrite rules, orders, graphs, trees and dags.

(3) informally overview concepts that are mentioned in this book but not formally defined, such as algorithms, decision problems, and complexity classes.

If the reader has a general knowledge of mathematical notions, she is still recommended to read such topics as systems of rewrite rules and inference systems, because the related notions we introduce here do not belong to standard mathematical curricula.

Note that this book contains an index, so all definitions and notation introduced here can be found in the index.

2.1 General Notation

Throughout this book, we use the following symbols. The symbol $\stackrel{\text{def}}{=}$ means "equal by definition". For example, $x \stackrel{\text{def}}{=} E$ means that x is defined to be E. The *symbol* $=$ denotes identity of two expressions. For example, $x = E$ means that x and E coincide. The *symbol* $\Leftrightarrow$ means "if and only if". For example, we can write $i > j \Leftrightarrow j < i$.

The symbol ❑ denotes the end of proof, definition, example, or exercise. If this symbol ends the assertion of a lemma or a theorem, then no proof will be given for this lemma or theorem.

To make the text shorter, we will usually abbreviate expressions such as "for all $i \in \{1, \ldots, n\}$" or "for all $1 \leq i \leq n$" to simply "for all i" when the range of i is clear from the context. For example, when we introduce $a_1, \ldots, a_n$, we can say "for all a_i" instead of "for all a_i such that $1 \leq i \leq n$". Note that we do not want to lose mathematical rigor and use this notation only when the range of i is unambiguously known from the context.

By $\mathbb{N}$ we denote the set of *natural numbers*, that is, non-negative integers $\{0, 1, 2, \ldots\}$.

2.2 Decision Problems and Algorithms

This book is not intended to be a text on computability, algorithms, or decision problems. However, we will occasionally introduce decision problems. For thorough discussion of computability, complexity or decision problems the reader is referred to the standard textbooks in the areas, for example [Rogers 1967, Garey and Johnson 1979, Hopcroft and Ullmann 1979, Odifreddi 1989, Balcázar, Diaz and Gabarró 1990, Papadimitriou 1994].

2.2.1 Decision Problems

Roughly speaking, a *decision problem* is characterized by an infinite set of *instances*. Each instance is a finite expression written in a countable language. Each instance has an answer,

which is either *yes* or *no*. Intuitively, the answer is *yes* if the corresponding instance holds. A simple example of a decision problem is the *addition of natural numbers*: given three natural numbers a, b, c, does $a + b = c$ hold? For this decision problem, an instance is a triple of natural numbers (a, b, c). The answer is *yes*, if $a+b = c$, and *no* otherwise. The set of instances of a decision problem is also called the set of possible *inputs*. Usually, we can write the input in several different forms. For example, we can consider the alphabet consisting of two symbols "|" and ",", denote a natural number n by a sequence of n symbols "|", and use "," as a separator between a, b, c. Then the instance $(2, 2, 4)$ of the addition problem will be written as $||, ||, ||||$. This instance has the answer *yes*. Alternatively, we can write the input triple of natural numbers using the binary notation in the alphabet consisting of three symbols "0", "1", and ",", and denote the above instance by $10, 10, 100$. When we formalize a mathematical problem as a decision problem, we must fix a particular way of representing instances by strings.

The following problem is *not* a decision problem: given two integers a, b, find the integer c such that $a + b = c$, because answers for this problem are integers, while a decision problem must have a yes-no answer. Another example of a problem which is not a decision problem is the following: "is Fermat's last conjecture true?". Although it has a yes-no answer, there is only one input, so the set of all inputs is finite. A relevant decision problem would be: given a positive integer n, does there exist positive integers a, b, c such that $a^n + b^n = c^n$? It has an infinite set of instances (the set of positive integers), and the yes-no answer for every instance.

A problem is called *decidable* if there exists an algorithm which computes the correct yes-no answers for all given inputs. Since nearly all modern programming languages are *Turing complete*, i.e., every algorithm is expressible in any such language, we can say that a problem is decidable if there exists a (say) C++ or Java program which computes the yes-no answer. There are some decision problems which are *undecidable*, one of them is the validity problem for first-order logic, which is a central problem for mathematical logic and will be discussed in the second volume of this book.

If we say "given x one can *effectively find* y such that...", then we mean that there exists an algorithm that computes such a y from every given x.

2.2.2 Algorithms

We will present algorithms in this book using an imperative C-like language. But unlike C, we will try to avoid using many semicolons and braces. We will separate statements by putting them on different lines, and groups of statements by aligning them vertically, i.e. using indentation. This idea is taken over from the programming language Python. The reserved words of the language are boldfaced and underlined, for example **if**, **then**, **case**.

We illustrate our way of presenting algorithms by an algorithm for addition of natural numbers presented in the binary notation. The algorithm for this problem is given in Figure 2.1. The reader can note that different statements are put on different lines. To save space, we can also put statements on the same line, in this case we separate them

```
procedure add(x_1, x_2)
input: natural numbers (x_1, x_2) written as 0-1 strings
output: natural number x_1 + x_2 written a 0-1 string
begin
  let r = 0 be an integer
  let sum =″ be a string
  while at least one of x_1, x_2 is non-empty do
    if x_1 =″ then x_1 := 0
    if x_2 =″ then x_2 := 0
    let u_i, b_i be strings such that u_i b_i = x_i and b_i is the last bit of x_i, for i = 1, 2
    case b_1 + b_2 + r of
      0 ⇒ sum := 0sum; r := 0
      1 ⇒ sum := 1sum; r := 0
      2 ⇒ sum := 0sum; r := 1
      3 ⇒ sum := 1sum; r := 1
    (x_1, x_2) := (u_1, u_2)
  return sum
end
```

Figure 2.1: Algorithm for binary addition

by a semicolon. An example is the line which contains two assignments $sum := 0sum$; $r := 0$. Several assignment statements can also be represented as a single simultaneous assignment, for example $(x_1, x_2) := (u_1, u_2)$ or $(x_1, x_2) := (x_2, x_1)$. There are several groups of statements which can be recognized by the indentation: statements in a group are left-aligned. For example, in Figure 2.1 there is a group of four statements: two **let** statements followed by a **while** and a **return**. Another group of five statements consists of two **if** statements, followed by a **let**, a **case**, and an assignment.

2.3 Complexity Classes

Decisions problems can be classified by their *complexity*. Intutitively, complexity formalizes the notion of resources (time and space) required to compute an answer as a function of the size of the input. In order to formalize complexity, one can use an abstract computational devise, such as *Turing machine*, which allows one to formalise the notion of computation and also time and space taken by a computation.

The notions of nondeterministic and deterministic time and space complexity that we use are standard [Hopcroft and Ullmann 1979]. In particular, we use the following acronyms for the corresponding standard complexity classes. We want to note that in some cases there is no unanimous notation for a class in the literature. We refer the reader to Papadimitriou

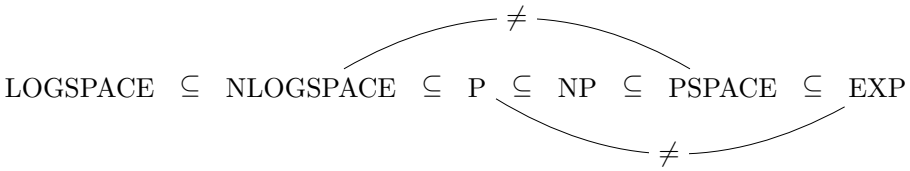


Figure 2.2: Known relationships between the complexity classes. None of the inclusions are known to be proper.

[1994] or Garey and Johnson [1979] in those cases when a complexity class is not treated in [Hopcroft and Ullmann 1979].

- LOGSPACE is the class of problems that can be solved deterministically within logarithmic space.
- NLOGSPACE is the class of problems that can be solved nondeterministically within logarithmic space.
- P is the class of problems that can be solved deterministically within polynomial time.
- NP is the class of problems that can be solved nondeterministically within polynomial time.
- PSPACE is the class of problems that can be solved within polynomial space.
- EXP is the class of problems that can be solved deterministically within exponential time, i.e., within time $2^{p(n)}$, where p is a polynomial and n the size of the input.
- NEXP is the class of problems that can be solved nondeterministically within exponential time.

2.4 Sets, Multisets and Mappings

We will use the standard mathematical notation for sets, with some exceptions. The readers of this book are assumed to have some background in mathematics, so we do not define the standard mathematical notions here.

2.4.1 Sets

We use the standard notation $\cup$, $\cap$ and $\setminus$ for the standard set operations of union, intersection, and set difference.

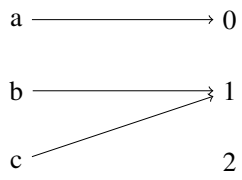


Figure 2.3: A finite mapping

2.4.2 Mappings

Mappings, and especially finite mappings, will be used in this book extensively. Every mapping $f : A \to B$ can be identified with its *graph*, i.e. the set of pairs (a, b) such that $f(a) = b$. When we discuss mappings, we will often denote such pairs by $(a \mapsto b)$.[1] In view of this notation, we denote by $f + (a \mapsto b)$ the mapping obtained by defining the value of f on a to be b. More precisely, $f + (a \mapsto b)$ is the mapping f' such that

$$f'(x) \stackrel{\text{def}}{=} \begin{cases} b, & \text{if } x = a; \\ f(x), & \text{otherwise.} \end{cases}$$

Let $a_1, \ldots, a_n$ are distinct elements. We use $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ to denote the finite mapping f defined as follows:

$$f(x) \stackrel{\text{def}}{=} \begin{cases} b_1, & \text{if } x = a_1; \\ \ldots & \ldots \\ b_n, & \text{if } x = a_n; \\ \mathit{undefined}, & \text{otherwise.} \end{cases}$$

Consider, for example, Figure 2.3 showing a finite mapping from the set $\{a, b, c\}$ to the set $\{0, 1, 2\}$. Then this mapping can be written as

$$\{a \mapsto 0, b \mapsto 1, c \mapsto 1\}.$$

If $f : A \to B$ and $g : B \to C$ are mappings, then we denote by $f \circ g$ their *composition*, that is a mapping from A to C defined as follows:

$$(f \circ g)(x) \stackrel{\text{def}}{=} g(f(x)).$$

[1] Users of the programming language Perl should be used to the fact that $(a \Rightarrow b)$ can be considered as syntactic sugar for the pair (a, b).

2.5 Binary Relations and Orders

2.5.1 Binary Relations

In this book we will often denote binary relations by an arrow-like symbol, for example, $\rightarrow$, or $\Rightarrow$. This notation is normally used when the binary relation $\Rightarrow$ denotes steps in a process, that is, $s_1 \Rightarrow s_2$ denotes that the process state can change in one step from s_1 into s_2. It is also often used to denote arcs in a directed graph (see below). Instead of writing $(x, y) \in \Rightarrow$ we will write $x \Rightarrow y$. If $\Rightarrow$ is a binary relation, we also define the following binary relations derived from $\Rightarrow$.

- The *inverse* of $\Rightarrow$, denoted $\Leftarrow$, is defined by $(x \Leftarrow y) \stackrel{\text{def}}{=} (y \Rightarrow x)$, for all x, y.
- The *symmetric closure* of $\Rightarrow$, denoted $\Leftrightarrow$, is defined as $\Rightarrow \cup \Leftarrow$. Equivalently, we can define $\Leftrightarrow$ by saying that $(x \Leftrightarrow y)$ if and only if $(x \Rightarrow y)$ or $(y \Rightarrow x)$, for all x, y.
- The *transitive closure* of $\Rightarrow$, denoted $\Rightarrow^+$, is defined by: $x \Rightarrow^+ y$ if and only if there exist $n \geq 0$ and elements $x_1, \ldots, x_n$, such that $x \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_n \Rightarrow y$.
- The *reflexive and transitive closure* of $\Rightarrow$, denoted $\Rightarrow^*$, is defined by: $x \Rightarrow^* y$ if and only if $x = y$ or $x \Rightarrow^+ y$, for all x, y.

For example, suppose that $\Rightarrow$ is defined as a finite relation $\{1 \Rightarrow 2, 2 \Rightarrow 3\}$ on the set $\{1, 2, 3\}$. Then

- the inverse of $\Rightarrow$ is the relation $\{2 \Rightarrow 1, 3 \Rightarrow 2\}$;
- the symmetric closure of $\Rightarrow$ is the relation $\{1 \Rightarrow 2, 2 \Rightarrow 1, 2 \Rightarrow 3, 3 \Rightarrow 2\}$;
- the transitive closure of $\Rightarrow$ is the relation $\{1 \Rightarrow 2, 2 \Rightarrow 3, 1 \Rightarrow 3\}$;
- the reflexive and transitive closure of $\Rightarrow$ is the relation

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

Note that the reflexive and transitive closure depends on the set on which $\Rightarrow$ is defined. For example, if $\Rightarrow$ in this example was a relation over the set of natural numbers, then the reflexive and transitive closure of $\Rightarrow$ would be an infinite relation

$$\{1 \Rightarrow 2, 2 \Rightarrow 3, 1 \Rightarrow 3\} \cup \{n \mapsto n \mid n \in \mathbb{N}\}.$$

Let $\Rightarrow$ be a binary relation on S. We call $x \in S$ *irreducible* w.r.t. $\Rightarrow$, if there is no y such that $x \Rightarrow y$. We write $x \Rightarrow^! y$ if $x \Rightarrow^* y$ and y is irreducible. Then y is called a *normal form* of x w.r.t. $\Rightarrow$.

For example, suppose that $\Rightarrow$ is defined as a finite relation

$$\{1 \Rightarrow 2, 2 \Rightarrow 3, 2 \Rightarrow 4, 5 \Rightarrow 5\}$$

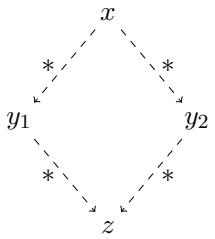


Figure 2.4: Confluence

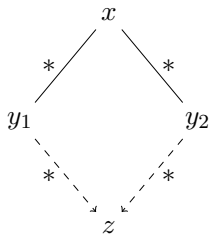


Figure 2.5: Local confluence

on the set $\{1, 2, 3, 4, 5\}$. Then 3 and 4 are irreducible w.r.t. $\Rightarrow$ while 1, 2, and 5 are reducible. The element 1 has two normal forms (3 and 4), while 3 has a unique normal form (3 itself). The element 5 has no normal form.

The relation $\Rightarrow$ is called *well-founded* or *terminating* if there exists no infinite sequence $x_1 \Rightarrow x_2 \Rightarrow \ldots$ and it is called *confluent* if the relation $\Leftarrow^* \circ \Rightarrow^*$ is contained in $\Rightarrow^* \circ \Leftarrow^*$, i.e., for every x, y_1, y_2 such that $x \Rightarrow^* y_1$ and $x \Rightarrow^* y_2$ there exists z such that $y_1 \Rightarrow^* z$ and $y_2 \Rightarrow^* z$ (see Figure 2.4). The relation $\Rightarrow$ is called *locally confluent* if $\Leftarrow \circ \Rightarrow \subseteq \Rightarrow^* \circ \Leftarrow^*$ (see Figure 2.5). A confluent and terminating relation is also called *convergent*.

It is known that there are locally confluent relations that are not confluent. However, every terminating locally confluent relation is confluent.

2.5.2 Orders

An *order*, or a *partial order*, is a binary relation which is irreflexive and transitive. This means that any order $>$ satisfies the following properties:

(1) (*irreflexivity*) there exists no x such that $x > x$;

(2) (*transitivity*) if $x > y$ and $y > z$, then $x > z$.

An order $>$ on a set S is called a *total order*, or *linear order*, if for every $x, y \in S$, if $x \neq y$, then either $x > y$ or $y > x$. For every order $>$ we denote by $\geq$ the *non-strict version of* $>$, defined by

$$x \geq y \stackrel{\text{def}}{=} (x > y \text{ or } x = y).$$

We will use a similar notation for the non-strict versions of all orders, for example if the order is denoted by $\succ$, then its non-strict version is denoted by $\succeq$.

Conversely, for every reflexive and transitive relation $\geq$, we denote by $>$ its *strict part* defined by:

$$x > y \stackrel{\text{def}}{=} x \geq y \text{ and } y \not\geq x,$$

and similar for other notations for reflexive and transitive relations, for example $\succ$ denotes a strict version of $\succeq$. In order for the strict part to be an order, the relation $\geq$ must satisfy *anti-reflexivity*: if $x \geq y$ and $y \geq x$, then $x = y$.

DEFINITION 2.1 (Minimal, Maximal) Let $>$ be an order on a set S and $S' \subseteq S$. If $x > y$, we say that x is *greater* than y and y is *less* than x (w.r.t. $>$). An element $x \in S$ is called *minimal w.r.t.* S' (respectively, *maximal w.r.t.* S') if there exists no $y \in S'$ such that $x > y$ (respectively $y > x$). If x is minimal (respectively, maximal) w.r.t. S' and $x \in S'$, then we say that x *minimal* (respectively *maximal*) *in* S'. If for all $y \in S'$ we have $y \geq x$ (respectively, $x \geq y$), then x is said to be the *least* (respectively, the *greatest*) element w.r.t. S'. If x is the least (respectively, the greatest) element of S' and $x \in S'$ we also say that x is the least (respectively, the greatest) element *in* S'. ❑

With every order $>$ we associate its *inverse*, always denoted by $<$, defined by

$$x < y \stackrel{\text{def}}{=} y > x.$$

It is easy to see that the inverse of an order is also an order. We will use a similar notation for the inverse of all orders, for example if the order is denoted by $\succ$, then its inverse is denoted by $\prec$. Naturally, the non-strict version of $<$ is denoted by $\leq$.

For the inverse versions $<$ of orders we will use the notions dual to those used for $>$ in Definition 2.1. For example, if $x < y$ we say that x is less than y.

2.6 Graphs, Dags, and Trees

In this section we overview directed graphs, graphs, and trees.

2.6.1 Directed Graphs

DEFINITION 2.2 (Directed Graph) A *directed graph* $G = (N, A)$ consists of a set N and a binary relation A on N. Elements of N are called *nodes*, elements of A are called *arcs*. If $(n_1, n_2) \in A$, we say that there is an *arc from* n_1 *to* n_2 *in* G and denote it by $n_1 \to n_2$. In this case we call n_1 a *predecessor* of n_2 and n_2 a *successor* of n_1. A directed graph is *finite* if its set of nodes is finite. ❑

An example directed graph is shown in Figure 2.6. The set of nodes is $\{a, b, c, d, e\}$ and the set of arcs is

$$\{(a, a),\ (a, b),\ (b, c),\ (b, d),\ (c, b),\ (c, d),\ (e, a),\ (e, d)\}.$$

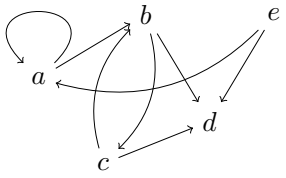


Figure 2.6: A directed graph

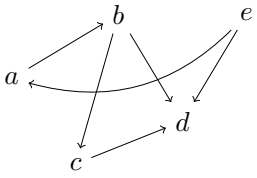


Figure 2.7: A dag

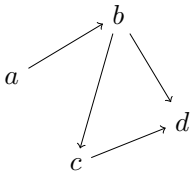


Figure 2.8: Its subdag rooted at a

DEFINITION 2.3 (Path, Cycle) A *path* in a directed graph is a finite or infinite sequence of nodes $n_0, n_1, \ldots$ such that $n_i \to n_{i+1}$ for all i. If the path is finite, that is, of the form $n_1, \ldots, n_k$ for some k, we say that the path is *from* n_1 *to* n_k. The number $k-1$ is the *length* of the path. If there exists a path from a node n to a node n', we say that n' is *accessible*, or *reachable* from n. Such a path is called a *cycle* if $k > 1$ and $n_1 = n_k$. ❑

For example, in Figure 2.6, a and a, a, b, d are paths, and a, a and b, c, b are some of the cycles. Evidently, a finite graph has an infinite path if and only if it has a cycle.

2.6.2 Directed Acyclic Graphs

DEFINITION 2.4 (Dag) A directed graph with no cycles is called a *directed acyclic graph*, or *dag*. If a dag has a node n so that every node in the dag is reachable from n, then n is called the *root* of this dag, and we say that the dag is *rooted at* n. ❑

For example, the graph in Figure 2.6 is not a dag, since it has cycles. By removing from this graph the arcs $a \to a$ and $c \to b$ we obtain the dag shown in Figure 2.7. This dag is rooted at e.

Let $d = (N, A)$ be a dag with the set of nodes N and the set of arcs A, and $n \in N$ be a node. Consider the subset $N' \subseteq N$ of nodes reachable from n and the subset $A \subseteq A'$ of arcs connecting the nodes in N'. Then $d' \stackrel{\text{def}}{=} (N', A')$ is also a dag, moreover n is the root of this dag. We will call d' the *subdag of* d *rooted at* n. For example, the subdag of the dag of Figure 2.7 rooted at a is shown in Figure 2.8.

Since any dags has no cycles, all paths in a finite dag are finite.

2.6.3 Trees

DEFINITION 2.5 (Tree) A *tree* is a dag with the following properties. (i) it has a root; (ii) each node other than the root has exactly one predecessor; and (iii) the successors of every node are ordered. ❑

We shall usually draw trees with the root at the top and arcs pointing downward. Then successors of each node are ordered left-to-right. An example tree is shown in Figure 2.9.

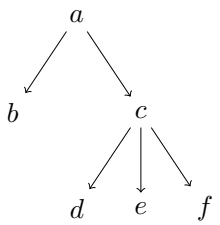
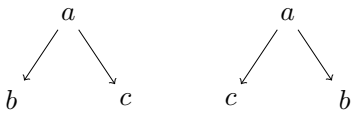


Figure 2.9: A tree

The root of this tree is a.

The condition that the successors of a node be ordered means, for example, that the following two trees are different.



There is a special terminology associated with trees.

DEFINITION 2.6 We call a *child* of a node a successor of this node. Likewise, the *parent* of a node is the predecessor of this node. A node with no children is called a *leaf*, otherwise it is called an *internal node*. A node n_1 is called a *descendant* (respectively, *ascendant*) of a node n_2 if there exists a path from n_2 to n_1 (respectively from n_1 to n_2). A *branch* is any path (maybe infinite) from the root which cannot be extended. This means that either a branch either infinite, or the last node of it is a leaf. ❑

For example, the tree of Figure 2.9, has the leaves b, d, e, f and internal nodes a, c. An example of a branch is a, c, e. The sequence a, c is a path but not branch. The tree of Figure 2.10 has exactly two branches, the branch a_0, b_1, b_2 is finite, while the branch $a_0, a_1, a_2, \ldots$ is infinite. This tree has in infinite number of paths.

Note that every node is an ascendant (and a descendant) of itself, and every node is a descendant of the root.

Sometimes we shall draw arcs in trees pointing upward, then they will resemble ordinary trees. We shall also display arcs by lines rather than arrows, as illustrated in Figure 2.11.

Let d be a tree and d' the subdag of d rooted at a node n. It is not hard to argue that d' is a tree whose root is n. Therefore, we will call d' the *subtree of* d *rooted at* n.

A dag or a tree is called *binary* if every internal node in it has exactly two successors. These two successors will be called the *left* and the *right* successor, respectively. For example, none of the trees in Figures 2.10 and 2.11 are binary, but the trees of Figure 2.11 can be made into binary by removing the node e.

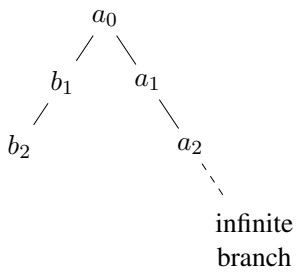


Figure 2.10: Branches

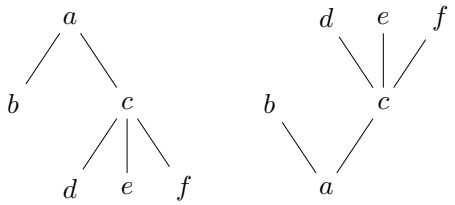


Figure 2.11: Two representations of trees

2.7 Inductive Definitions of Expressions and Parse Trees

In this book we introduce several kinds of expressions, for example, formulas. A suitable notion of expression will normally be defined by induction. When we have an *inductive definition* of expressions, it will be convenient for us to use trees for representing them. In this section we discuss the notions of inductive definition, expression, and parse tree. As an example, we will use arithmetical expressions over natural numbers.

2.7.1 Inductive Definitions, Expressions, and Subexpressions

Before defining arithmetical expressions formally, we will explain them informally. A natural number n will be represented by the expression $\underbrace{|\ldots|}_{n \text{ times}}\varepsilon$. For example, $|||||\varepsilon$ represents the number 5, while ε represents the number 0. In this section we call such expressions *primitive*. More complex expressions are built from the primitive expressions using the symbols $|$ (to denote the addition of 1, or the successor operation), $+$ (to denote addition) and $\cdot$ (to denote multiplication).

The following definitions are simply illustrating an inductive definition of expressions, they will not be used in further chapters of this book. Nevertheless, the reader is advised to read through this example, as the definitions introduced here will be used for many important notions, such as formula.

DEFINITION 2.7 (Arithmetical Expressions) Arithmetical expressions are defined inductively as follows.

(1) ε is an arithmetical expression.

(2) If e is an expression, then $|e$ is an arithmetical expression

(3) If e_1, e_2 are arithmetical expressions, then $(e_1 + e_2)$ and $(e_1 \cdot e_2)$ are arithmetical expressions. ❑

Let us analyze this definition. Each clause of the definition says how an arithmetical expression is built from other expressions using *constructors*. The constructors here are ε, $|$, $+$ and $\cdot$. We also use parentheses to disambiguate complex expressions. Without parentheses, the expression $\varepsilon + \varepsilon \cdot \varepsilon$ would have two readings: $(\varepsilon + \varepsilon) \cdot \varepsilon$ and $\varepsilon + (\varepsilon \cdot \varepsilon)$. We will always define various kinds of expressions such that every expression have a unique reading. In the definitions of this book we will use various names for constructors. For example, the constructors of propositional formulas are called *connectives*, the constructors of quantified boolean formulas are called *quantifiers*, and the constructors of temporal formulas are called *temporal operators*.

Inductive Definition 2.7 of arithmetical expressions induces two important notions: those of a *subexpression* and a *parse tree*. Each clause of this definition introduces an expression e built using a constructor from other expressions, these expressions are called the *immediate subexpressions* of e.

DEFINITION 2.8 (Subexpressions) *Subexpressions* are defined inductively as follows. First, let us define immediate subexpressions. The notion of immediate subexpression is obtained directly by inspecting the inductive definition of expression: every clause in the definition, except for the simplest ones, yields a new kind of immediate subexpressions. For the arithmetical expressions the definition is as follows. For every expressions e_1, e_2, e_3 we have

(1) e is the immediate subexpression of $|e$.

(2) e_1 and e_2 are the immediate subexpressions of $(e_1 + e_2)$ and $(e_1 \cdot e_2)$.

The notion of subexpression is defined as the reflexive and transitive closure of the notion of immediate subexpression:

(3) e_1 is a subexpression of e_1;

(4) if e_1 is an immediate subexpression of e_2 and e_2 is a subexpression of e_3, then e_1 is a subexpression of e_3. ❑

Note the following.

(1) ε has no immediate subexpressions.

(2) The notion of subexpression can be obtained from the inductive definition of expressions in a completely automatic way: every clause which defines how a new expression is built from one or more expressions using a constructor, gives us a clause in the definition of a subexpression.

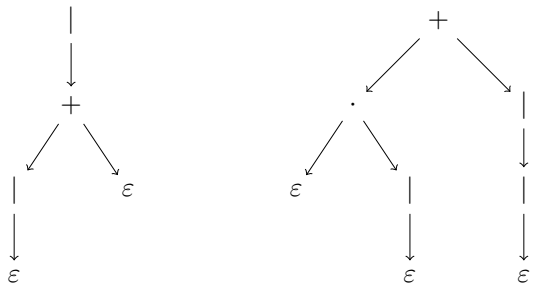


Figure 2.12: Parse Trees

2.7.2 Parse Trees and Precedences

Many kinds of inductively defined expressions of this book are best illustrated when the expressions are represented by trees. These trees are called *parse trees* and can be obtained by analyzing how an expression is built from its immediate subexpressions. The nodes of parse trees are labeled by constructors. The parse tree for an expression is built using a simple rule: if an expression e is built from its immediate subexpressions $e_1, \ldots, e_n$ using a constructor c, then the parse tree for e has c as the root and the parse trees for $e_1, \ldots, e_n$ as its children. Applying this rule to the arithmetical expressions we obtain

DEFINITION 2.9 (Parse Tree for Arithmetical Expressions) The *parse tree* for an arithmetical expression e is defined by induction on e as follows.

(1) The parse tree for ε consists of one leaf ε.

(2) The parse tree for $|e$ has the root $|$. The root has a single child that is the parse tree for e.

(3) The parse tree for $(e_1 + e_2)$ (respectively, for $(e_1 \cdot e_2)$) has the root $+$ (respectively, $\cdot$). The root has two children that are the parse trees for e_1 and e_2. ❑

For example, the parse trees for the expressions $|(|\varepsilon + \varepsilon)$ and $(\varepsilon \cdot |\varepsilon) + ||\varepsilon$ are given in Figure 2.12.

To avoid writing too many parentheses in expressions, we will sometimes omit parentheses. To disambiguate the reading of expressions without parentheses, we will assign a numeric *precedence* to all constructors. In the case of an ambiguous expression the precedences will be used to resolve the ambiguity.

DEFINITION 2.10 (Precedence) A *precedence* is a natural number. A *precedence function* assigns a precedence to every constructor having at least one argument. Let P be a precedence function. We say that a constructor c_1 has a *higher precedence* (respectively, *equal precedence*) than a constructor c_2 if $P(c_1) > P(c_2)$ (respectively, $P(c_1) = P(c_2)$). ❑

Given a precedence function P, we can resolve some ambiguous expressions with omitted parentheses as follows. If the expression has several constructors which can be used as the root of the parse tree, then the constructors with the highest precedence is selected. If there are several such constructors, then the expression is still considered ambiguous.

EXAMPLE 2.11 (Precedences) Consider the following arithmetical expression $||\varepsilon + \varepsilon \cdot |\varepsilon$ with omitted parentheses. Without precedences, this expression is ambiguous. For example, we can disambiguate this expression using any of the following expressions:

$$|(|(\varepsilon + \varepsilon) \cdot |\varepsilon)$$
$$||((\varepsilon + \varepsilon) \cdot |\varepsilon)$$
$$(||\varepsilon + (\varepsilon \cdot |\varepsilon))$$

Consider now the following precedence function P:

$$P(|) = 2; \quad P(\cdot) = 1; \quad P(+) = 0.$$

Note that we do not assign any precedence to the constructor ε, since it has no arguments. Using this precedence function, this expression is unambiguous and can only be read as $(||\varepsilon + (\varepsilon \cdot |\varepsilon))$.

Even when we use this precedence function, some expressions remain ambiguous, for example $\varepsilon + \varepsilon + \varepsilon$, which can be disambiguated by adding one pair of parentheses, for example $(\varepsilon + \varepsilon) + \varepsilon$ or $\varepsilon + (\varepsilon + \varepsilon)$. ❑

2.7.3 Positions

Sometimes we wish to distinguish a particular occurrence of a subexpression in an expression. For example, the expression $(\varepsilon \cdot |\varepsilon) + ||\varepsilon$ corresponding to the parse tree on the right of Figure 2.12 has three different occurrences of the expression ε and two different occurrences of the expression $|\varepsilon$. Parse trees give us a convenient way of speaking about particular occurrences of subexpressions into expressions. This can be done by using the notion of *position*. The notion of position can also be automatically generated from the inductive definition of expressions: each clause in the inductive definition gives us a clause in the definition of position. Let us define position for arithmetical expressions.

DEFINITION 2.12 (Position) A *position* is a sequence of integers. Let us define by induction the notion of a *position in an expression* E. For each position π in an expression E we also define the notion of the *subexpression of* E *at the position* π, denoted $E|_\pi$. The definition for arithmetical expressions is as follows.

(1) For every expression E, the empty string ϵ is a position in E, and $E|_\epsilon \stackrel{\text{def}}{=} E$.

(2) If $E|_\pi = |E_1$, then $\pi 1$ is a position in E and $E|_{\pi 1} = E_1$.

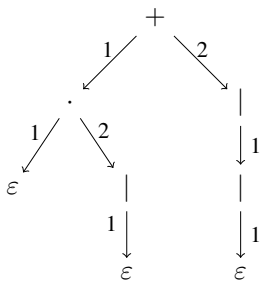


Figure 2.13: Positions

(3) If $E|_\pi = E_1 + E_2$, then $\pi 1$ and $\pi 2$ are positions in E and $E|_{\pi 1} = E_1$, $E|_{\pi 2} = E_2$.

(4) If $E|_\pi = E_1 \cdot E_2$, then $\pi 1$ and $\pi 2$ are positions in E and $E|_{\pi 1} = E_1$, $E|_{\pi 2} = E_2$. ❑

The best way to illustrate positions is to consider the parse tree of an expression and, for every node, label all arcs from this node left-to-right by $1, 2, \ldots$. For example, Figure 2.13 shows a labeled tree corresponding to the tree of Figure 2.12. To find the subexpression at a position π, we follow the path of the tree starting at the root choosing the arcs numbered as the numbers in π. For example, for the expression E of Figure 2.13 we have $E|_{21} = |\varepsilon$.

The notion of a position also gives us a convenient way of speaking about replacement of subexpressions by other subexpressions. Let E, E' be expressions and π be a position in E. Denote by $E[E']_\pi$ the expression obtained from E by replacing the subexpression at the position π by E'. For example, $((\varepsilon \cdot |\varepsilon) + ||\varepsilon)[\varepsilon]_2 = (\varepsilon \cdot |\varepsilon) + \varepsilon$. Note that the subexpression of $E[E']_\pi$ at the position π is E'. Therefore, we will use the same notation $E[E']_\pi$ to denote an expression E having E' as the subexpression at the position π. We will omit the position π when it is clear from the context or unique. For example, we can say: "let $E[E_1]$ be an expression, then $E[E_2]$ ..." instead of saying: "let E be an expression containing a subexpression E_1 at the position π, then the expression $E[E_2]_\pi$ ...."

This notation is slightly ambiguous: consider, for example, $E[E]_\pi$. This is a correct expression for all positions π in E, but we cannot say that E contains E at the position π, unless π is the empty position ϵ. However, we will only use this notation in an unambiguous way. For example, we can say: "let $e[(e_1 \cdot e_2)]$ be an expression, then $e[(e_2 + e_1)]$ ...". It is easy to see that in this case the meaning is unambiguous: the first expression contains a subexpression $(e_1 + e_2)$, and the second one is obtained from it by replacing one occurrence of this subexpression by $(e_2 + e_1)$.

2.8 Rewrite Rule Systems

The so-called *rewrite rule systems* will be used for denoting various algorithms on expressions. In this section we introduce the notion of rewrite rule system and define some

associated notions.

When we have defined a suitable notion of expression, we can generalize it to contain *patterns*. Patterns use a special alphabet of *variables* $x_1, x_2, \ldots$ not used in the expressions. The notion of pattern is defined in the same way as that of expression, but with the change that a variable may occur in a pattern. In the case of arithmetical expressions patterns are defined as follows:

(1) x_i is a pattern, for every i.

(2) ε is a pattern.

(3) If p is a pattern, then $|p$ is a pattern.

(4) If p_1, p_2 are patterns, then $(p_1 + p_2)$ and $(p_1 \cdot p_2)$ are patterns.

An example of a pattern over arithmetical expressions is

$$(x_1 + (||| x_1 \cdot x_2)). \tag{2.1}$$

When we substitute concrete arithmetical expressions for the variables in a pattern, we obtain an arithmetical expression. For example, when we substitute $|\varepsilon$ for x_1 and $(||\varepsilon + |\varepsilon)$ for x_2 in (2.1) we obtain the expression

$$(|\varepsilon + (|||| \varepsilon \cdot (||\varepsilon + |\varepsilon))).$$

We call a *substitution* any mapping σ from a finite set of variables to the set of expressions. We denote by $p\sigma$ the pattern obtained from p by replacing every variable x_i on which σ is defined by $\sigma(x_i)$. Note that $p\sigma$ is an expression if and only if σ is defined on every variable of p.

A binary relation R on expressions is called *monotonic* if for every expression $e[e_1]$ and expression e_2, if $(e_1, e_2) \in R$, then $(e[e_1], e[e_2]) \in R$.

DEFINITION 2.13 (Rewrite Rule System) A *rewrite rule* is an ordered pair (l, r), denoted by $l \Rightarrow r$, where (i) l, r are patterns; (ii) all variables of r also occur in l. A *rewrite rule system* is a set R of rewrite rules. The *rewrite relation* with R, denoted $\Rightarrow_R$, is the smallest monotonic relation such that $l\sigma \Rightarrow_R r\sigma$ for all $(l \Rightarrow r) \in R$ and all σ such that σ is defined on all variables of l. If $s \Rightarrow_R t$ then we say that s *rewrites into* t with R. We say that R is *terminating*, *confluent*, *convergent*, etc., if so is $\Rightarrow_R$. ❑

EXAMPLE 2.14 Let us give an example rewrite rule system intended for evaluating arithmetical expressions into primitive ones. The system contains the following rules:

$$\begin{array}{rcl} (\varepsilon + x) & \Rightarrow & x; \\ (|y + x) & \Rightarrow & |(y + x); \\ (\varepsilon \cdot x) & \Rightarrow & \varepsilon; \\ (|y \cdot x) & \Rightarrow & (x + (y \cdot x)). \end{array}$$

$$
\begin{array}{ll}
(|\varepsilon + |\varepsilon) \cdot (|\varepsilon + |\varepsilon) & \Rightarrow \\
||\varepsilon \cdot (|\varepsilon + |\varepsilon) & \Rightarrow \\
(|\varepsilon + |\varepsilon) + |\varepsilon \cdot (|\varepsilon + |\varepsilon) & \Rightarrow \\
(|\varepsilon + |\varepsilon) + ((|\varepsilon + |\varepsilon) + \varepsilon \cdot (|\varepsilon + |\varepsilon)) & \Rightarrow \\
(|\varepsilon + |\varepsilon) + ((|\varepsilon + |\varepsilon) + \varepsilon) & \Rightarrow \\
(|\varepsilon + |\varepsilon) + ((\varepsilon + ||\varepsilon) + \varepsilon) & \Rightarrow \\
(|\varepsilon + |\varepsilon) + (||\varepsilon + \varepsilon) & \Rightarrow \\
(\varepsilon + ||\varepsilon) + (||\varepsilon + \varepsilon) & \Rightarrow \\
||\varepsilon + (||\varepsilon + \varepsilon) & \Rightarrow \\
||\varepsilon + (|\varepsilon + |\varepsilon) & \Rightarrow \\
||\varepsilon + (\varepsilon + ||\varepsilon) & \Rightarrow \\
||\varepsilon + ||\varepsilon & \Rightarrow \\
|\varepsilon + |||\varepsilon & \Rightarrow \\
\varepsilon + ||||\varepsilon & \Rightarrow \\
||||\varepsilon &
\end{array}
$$

Figure 2.14: Computing normal form by rewriting

.

We claim that this system is confluent and terminating, and hence convergent. Moreover, it computes the values of arithmetical expressions (for a precise formulation see Exercise 2.17). ❑

When an expression s_1 is rewritten into s_n using a sequence $s_1 \Rightarrow s_2 \Rightarrow \ldots \Rightarrow s_n$, we will refer to every single rewriting $r_i \Rightarrow r_{i+1}$ as a *rewrite step*.

Every rewrite rule system defines (non-deterministic) computations on expressions. Namely, given an expression E_0 we can repeatedly rewrite it using the system as

$$E_0 \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots$$

Such a computation either reaches a normal form or can continue forever. For example, Figure 2.14 shows how we can compute by rewriting the normal form of the expression $(|\varepsilon + |\varepsilon) \cdot (|\varepsilon + |\varepsilon)$.

Note that the definition of rewrite rule systems and rewriting does not specify the order of rewrite steps. If two or more rewrite steps are applicable to an expression, then every one of them can be used for rewriting. For some rewrite rule systems, the order of rewrite steps can influence the result. For example, for the system on integers consisting of two rules $1 \Rightarrow 2$ and $1 \Rightarrow 3$ the expression 1 can either be rewritten to 2 or to 3. If it is rewritten to 2, then it cannot be rewritten to 3 afterward, so 1 has two different normal forms. If a system is confluent, then every expression has at most one normal form. In a way, every confluent system describes a partial function on expressions: a function mapping

an expression into its normal form. If a system is both confluent and terminating, i.e., convergent, then rewriting of an expression into its normal form can be done in a "don't-care" fashion: every sequence of rewrite steps terminates and computes the normal form.

Exercises

Decision Problems

EXERCISE 2.1 Which of the following problems can be formulated as decision problems?

(1) Given a natural number, find its representation as a product of prime numbers.

(2) Given a natural number, find whether this number is prime.

(3) Given a UK passport number, find whether this number is prime.

(4) Given a positive real number, find whether its integer part is prime.

(5) Given a country, find if its male population is larger than its female population. ❑

Algorithm

EXERCISE 2.2 Using notation similar to those of Figure 2.1, write down a simpler recursive algorithm for binary addition. ❑

Mappings

EXERCISE 2.3 A mapping $f : A \to B$ is called *finitely based* if $f(a)$ is defined only for a finite number of elements $a \in A$. Prove that the composition of two finitely based mappings is finitely based. ❑

EXERCISE 2.4 Let f be the mapping $\{1 \mapsto 2, 2 \mapsto 3\}$. Write down, using the notation $\{\ldots \mapsto \ldots\}$, the mappings $f + (3 \mapsto 1)$ and $f + (2 \mapsto 1)$. ❑

Binary relations

EXERCISE 2.5 Consider the following binary relation $\Rightarrow$ on the set $S = \{0, 1, 2, 3, 4, 5\}$:

$$\{2 \mapsto 1, 2 \mapsto 3, 3 \mapsto 3, 5 \mapsto 2, 5 \mapsto 4\}.$$

(1) Find the inverse of $\Rightarrow$.

(2) Find the symmetric closure of $\Rightarrow$.

(3) Find the transitive closure of $\Rightarrow$.

(4) Find the reflexive and transitive closure of $\Rightarrow$.

(5) Find all elements irreducible w.r.t. $\Rightarrow$.

(6) Build the binary relation $\Rightarrow^!$, that is, the set of all pairs (x, y) such that y is a normal form of x. ❑

EXERCISE 2.6 Let $\Rightarrow$ be a binary relation on a set S and $x \in S$. Prove that x has a normal form w.r.t. $\Rightarrow$ if an only if it has no normal form w.r.t. $\Rightarrow^!$. ❑

EXERCISE 2.7 ★ Prove that every terminating locally confluent relations is confluent. ❑

EXERCISE 2.8 ★ Give an example of a locally confluent relation that is not confluent. ❑

EXERCISE 2.9 Let $\Rightarrow$ be a well-founded relation. Prove that every element has a normal form w.r.t. $\Rightarrow$. ❑

EXERCISE 2.10 Give an example of a non-well-founded relation $\Rightarrow$ such that every element has a unique normal form w.r.t. $\Rightarrow$. ❑

Trees

EXERCISE 2.11 Let t be a tree. Then for every pair of nodes n_1, n_2 in t there exists at most one path from n_1 to n_2. In particular, for every node n there exists exactly one path from the root to n. ❑

EXERCISE 2.12 Let t be a finite tree. Then the number of branches in t is equal to the number of leaves in t. ❑

Expressions and Parse Trees

EXERCISE 2.13 Draw the parse trees for the following arithmetical expressions:

(1) $||\varepsilon + ||\varepsilon$;

(2) $|(||\varepsilon + |\varepsilon \cdot |\varepsilon)$. ❑

EXERCISE 2.14 Extend the notion of arithmetical expressions to define a notion of *comparison expression*. Comparison expressions have the form $e_1 \lambda e_2$, where e_1, e_2 are arithmetical expressions and λ is one of the following *comparison operators*: $>, \geq, =, \leq, <$. Note that to define this notion, we need two kinds of expressions: ordinary arithmetical expressions and comparison expressions, since we would like to avoid meaningless expressions like $(e_1 > e_2) + e_3$. Show how this definition of comparison expression gives rise to extended notions of subexpressions and parse trees. ❑

EXERCISE 2.15 (★) Define a suitable notion of expression for arithmetical expressions over binary integers. For these expressions, define a confluent and terminating rewrite rule system for evaluating expressions similar to that of Example 2.14 on page 19. ❑

Rewrite Rules

EXERCISE 2.16 Compute normal forms of the following expressions w.r.t. the rewrite rules of Example 2.14 on page 19.

(1) $||\varepsilon + ||\varepsilon$;

(2) $|(||\varepsilon + |\varepsilon \cdot |\varepsilon)$. ❑

EXERCISE 2.17 (★) Prove that the system of Example 2.14 is confluent and terminating. Moreover, prove that it evaluates expressions into their values in the following sense. Let us call the *image* of an arithmetical expression e the expression (this time in ordinary mathematical sense), denoted by $i(e)$ obtained by replacing every primitive expression $\underbrace{|\ldots|}_{n \text{ times}}\varepsilon$ by the integer n and every non-primitive expression as follows:

(1) if e is a non-primitive expression, then $i(|e) \stackrel{\text{def}}{=} i(e) + 1$;

(2) $i(e_1 + e_2) \stackrel{\text{def}}{=} i(e_1) + i(e_2)$;

(3) $i(e_1 \cdot e_2) \stackrel{\text{def}}{=} i(e_1) \cdot i(e_2)$;

Then every expression e we have

(1) its normal form e' is a primitive expression;

(2) the values of $i(e)$ and $i(e')$ coincide. ❑

EXERCISE 2.18 (See Exercises 2.14 and 2.17). Extend the comparison expressions by two expressions $false$ and $true$. Extend the rewrite system of Example 2.14 into a convergent rewrite system for evaluating comparison expressions so that every comparison expression evaluates to either $false$ and $true$. ❑