—DRAFT MATERIAL—

January 26, 2009

Lecture Notes for CSCI C241/H241

Induction, Recursion and Programming*

Steven D. Johnson
Computer Science Department
Indiana University School of Informatics

NOTICE: *This is draft material for use in the course* C241, Discrete Structures for Computer Science. *Enrolled students are permitted to print copies for use during this course. They should discard their copies on completion of the course. Past students may contact the author for a current revision. Further distribution in any form is not permitted without permission of the author. Contact:* `sjohnson@cs.indiana.edu`.

*Induction, Recursion, and Programming *is a Working title. Some content derives from the book* Induction, Recursion, and Programming *by Mitchell Wand (North-Holland, 1976). All rights to original content are reserved.*

New content © 2008 Steven D. Johnson

Contents

1 Sets **3**
1.1 Set Operations 6
1.2 Words and Languages 10
1.3 A Simple Algorithmic Language 14

2 Propositional Logic and Boolean Algebra **19**
2.1 Propositions and Truth Tables 19
2.1.1 Implication* 19
2.2 Truth Tables 22
2.3 Boolean Algebra 25
2.3.1 Duality 27
2.4 Normal Forms 28
2.5 Application of Boolean Algebra to Hardware Synthesis* 30

3 Counting **37**
3.1 Cardinality 37
3.2 Permutations and Combinations 38

4 Induction **47**
4.1 Numerical Induction 47
4.2 More Examples of Induction 55

5 Countability and Order **63**
5.1 Cardinality and Countability 63
5.2 Order Notation and Order Arithmetic 67
5.3 Complexity 70
5.3.1 The Halting Problem 70
5.3.2 Infeasible Problems. 72
5.3.3 Orders of Infinity. 72
5.4 Additional Problems 73

6 Relations **77**
6.1 Functions 78
6.1.1 Infix Notation 84

6.2 Relations on a Single Set 87
6.2.1 Attaching Information to Graphs 90
6.3 Trees 92
6.4 DAGs 98
6.5 Equivalence Relations 99
6.6 Partial Orders* 102
6.7 Decision Diagrams* 105

7 Induction II 111
7.1 Introduction 111
7.1.1 The Problem of Self Reference 113
7.2 Inductively Defined Sets 114
7.3 The Principle of Structural Induction. 117
7.4 Validity of the Induction Principle* 120
7.5 Defining Functions with Recursion 127
7.6 Evaluation of Recursive Functions 129
7.7 Reasoning about Recursive Functions 132

8 Languages and Meanings 139
8.1 Language Definitions 139
8.2 Defining How Languages are Interpreted 141
8.3 Specifying Precedence 144
8.4 Environments 146
8.5 Backus-Naur Form 149
8.6 Propositional Formulas 150
8.7 Substitution 153
8.8 The Programming Language of Statements 157
8.9 *Discussions 162
8.9.1 Parenthesized Expressions 162

9 Formal Logic 165
9.1 Propositional Logic 165
9.2 Formal Proofs 170
9.2.1 Deducability and Validity 172
9.2.2 A More Useful Propositional Calculus 174
9.3 First-order Predicates 177
9.4 Predicate Calculus 180

10 Proving Programs 183
10.1 The Language of Statements 183
10.1.1 Operational Interpretation of Statements 184
10.1.2 Axiomatic Interpretation of Statements 185
10.1.3 Reasoning Rules for Statements 185
10.1.4 The Compound Rule 187
10.1.5 The Conditional Rule 187
10.1.6 The Repetition Rule 188

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

10.1.7 The Relaxation Rule 188
10.2 Using the Rules 188

Chapter 1

Sets

The concepts of *set* and *set membership* are fundamental. A set is determined by its *elements* (or *members*) and to say that x is an element of the set S, we write:

$$x \in S$$

To say that y is *not* a member of S we write:

$$y \notin S$$

Simple sets are specified by listing all of their elements between braces. A set A of five numbers is specified:

$$A = \{1, 2, 3, 4, 5\}$$

A set B of three colors is specified:

$$B = \{red,\ blue,\ green\}$$

Example

Ex 1.1 *List the set S of whole numbers between 1 and 15 which are evenly divisible by either 2 or 3.*

Solution: *We might begin by listing the numbers divisible by 2:* $\{2, 4, 6, 8, 10, 12, 14, \ldots$ *and then the numbers divisible by 3:* $\ldots, 3, 6, 9, 12, 15\}$*. So the set we are looking for could be written*

$$S = \{2, 4, 6, 8, 10, 12, 14, 3, 6, 9, 12, 15\}$$

This listing contains duplications and the numbers are listed in a strange order. Neither of these problems makes the description incorrect, but it is less confusing to list each element just once. A better description for S is

$$\{2, 3, 4, 6, 8, 9, 10, 12, 14, 15\}$$

One way to abbreviate a long list of elements is to use *ellipses* to indicate a large, possibly infinite, range of values. For example, the set of lower-case letters ranging from ‘a’ to ‘z’ could be expressed as follows:

$$\{\text{‘a’},\ \text{‘b’},\ \ldots,\ \text{‘z’}\}$$

The set of numbers ranging from 1 to 10,000 could be specified:

$$\{1, 2, 3, \ldots, 10000\}$$

The following definition uses ellipses to describe some infinite sets that are used throughout this book.

Definition 1.1

(a) *The set of* whole numbers *is* $\mathbb{W} = \{1, 2, 3, \ldots\}$.

(b) *The set of* natural numbers *is* $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$

(c) *The set of* integers *is* $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$

Ellipses are useful when it is natural to list a set’s elements in some consecutive order, but care is needed in their use. Consider the set specification:

$$B = \{2, 4, \ldots, 64\}$$

It is not *completely* obvious from this set definition whether the elements of B are:

- the even numbers from 2 to 64:

$$B \stackrel{?}{=} \{\ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64\ \},$$

- or the powers-of-2 from 2 to 64:

$$B \stackrel{?}{=} \{2, 4, 8, 16, 32, 64\},$$

- or something else.

The rule-of-thumb is to take the simplest sequence that clearly describes the set exactly, but “simplest” can be a matter of judgment involving assumptions about the knowledge of the Reader. One way to avoid confusion is to include a formula representing a typical element of the list; for example:

$$B = \{2, \ldots, 2i, \ldots, 64\}$$

This description says that the elements of B have the form $2i$; they are even numbers. Although even this definition relies on the reader to understand that i refers to a whole number, the specification is better determined.

Copyright © 2008 Steven D. Johnson

A more general way to specify the elements of a set is to write down a property that is satisfied exactly by the elements of the set. The notation for doing this is called *set builder notation*:

$$\{x \in U \mid P[x]\}$$

U is the *universe* or *domain* from which prospective elements x originate, and $P[x]$ stands for some property that the members must satisfy. The property P must "make sense" with respect to U; that is, is, $P[u]$ must be either *true* or *false* for every element of U. The set specified contains *all* of the elements for which $P[x]$ is *true*. Mention of U may be omitted if either the surrounding context or P make clear what U is.

For example, the second version of B above might be specified

$$B = \{x \in \mathbb{W} \mid x = 2^i \text{ for } i \in \mathbb{W} \text{ such that } 1 \leq i \leq 6\}$$

The property P in this case is

$$P[x] \equiv \text{“}x = 2^i \text{ for some } i \in \mathbb{W} \text{ such that } 1 \leq i \leq 6\text{”}$$

Even though $P[x]$ contains two variables, x and i, it is nevertheless a statement about x *only;* the statement itself *quantifies* i by saying that $i \in \{1, 2, 3, 4, 5, 6\}$.

REMARK: You may have noticed that property P is defined using '[', ']' and '≡', rather than '(', ')' and '='. There is no difference in meaning, but throughout this book the $F[x] \equiv$ "—" notation is used for the purpose of defining *syntax*, or *how* to formulate something.

END REMARK

In this instance, the set-builder description is not much of an an improvement over simply listing the elements, although if i ranged from 1 to 100 it would be. We can do a little better by writing a formula in place of the simple variable x:

$$B = \{2i \mid i \in \mathbb{W} \text{ and } 1 \leq i \leq 6\}$$

we have omitted the declaration "$2i \in \mathbb{W}$" because it is clear from the context of the example that B is a set of whole numbers.

Example

Ex 1.2 *Specify the infinite set E of nonnegative even numbers using ellipses and then again using set-builder notation*

SOLUTION: Using ellipses, one could write $E = \{0, 2, 4, 6, \ldots\}$

It would be clearer to include a representative element $E = \{0, 2, \ldots, 2i, \ldots\}$. Using set-builder notation, we would write $\{2n \mid n \in \mathbb{N}\}$.

Copyright © 2008 Steven D. Johnson

Here are some other sets used in this book.

Definition 1.2

(a) *The set of* integers modulo n *is* $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$.

(b) *The set of* rational numbers*,* $\mathbb{Q}$*, consists of all fractions:*

$$\mathbb{Q} = \{\tfrac{n}{m} \mid n \in \mathbb{N} \textit{ and } m \in \mathbb{W}\}$$

(c) *The* real numbers*.*

REMARK: Despite the fact that most of our mathematical education has dealt with real numbers, it is hard to find a concise property $P_{\mathbb{R}}$ characterizing the real numbers, $\mathbb{R} = \{r \mid P_{\mathbb{R}}[r]\}$. END REMARK

1.1 Set Operations

There is that set which contains no elements. One way to express this set is to place nothing between braces: $\{\}$. We also use the symbol $\emptyset$ for this set.

Definition 1.3 *The* empty set*, denoted by* $\emptyset$*, has no elements:* $\emptyset = \{\}$.

A common mistake is writing "$\{\emptyset\}$" for the empty set. However, $\{\emptyset\}$ is *not* the "really empty" set but rather a set with a single element, namely, $\emptyset$. In set-builder notation, $\emptyset = \{x \mid \mathit{false}(x)\}$, where *false* represents a property that nothing satisfies.

Sets are compared by asking what elements they have in common.

Definition 1.4 *Let* A *and* B *be two sets.*

(a) A equals B*, written* $A = B$*, if* A *and* B *contain exactly the same elements.*

(b) A contains B*, written* $B \subseteq A$*, if every element of* B *is also an element of* A*.* A *is said to* properly contain B *when* $B \subseteq A$ *and* $B \neq A$*. This is sometimes written as* $B \subsetneq A$*.*

(c) A *and* B *are* disjoint *when they have no elements in common, that is,* $A \cap B = \emptyset$*.*

To prove that two sets, A and B are equal, one often shows that each contains the other. The following proposition follows immediately from Definition 1.4.

Fact 1.1 $A = B$ *iff* $A \subseteq B$ *and* $B \subseteq A$*.*

There are numerous ways to build new sets from sets which are given. The most common of these have names and special symbols associated with them as defined below.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Definition 1.5 *Let A and B be two sets.*

(a) *The* intersection *of A and B, written $A \cap B$, is the collection of elements that A and B have in common. That is,*

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

(b) *The* union *of A and B, written $A \cup B$, is the collection of all those elements in either set or both. That is,*

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

(c) *The* (set) difference *of A and B, written $A \setminus B$ is the collection of those elements of A which are not in B. That is,*

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

(d) *The* power set *of A written $\mathcal{P}(A)$, is the collection of A's subsets. That is,*

$$\mathcal{P}(A) = \{S \mid S \subseteq A\}$$

(e) *The* product *of A and B, written $A \times B$ is the collection of all* ordered pairs *whose first elements come from A and whose second elements come from B. That is,*

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

Example

Ex 1.3 *Let A be the set $\{1, 5\}$; and let B be the set $\{1, 2, 3\}$. Describe the sets $A \cap B$, $A \cup B$, $A \setminus B$, $\mathcal{P}(A)$, $A \times B$, $B \times A$, and $A \times \mathbb{N}$.*

SOLUTION: The sets are

$$\begin{aligned}
A \cap B &= \{1\} \\
A \cup B &= \{1, 2, 3, 5\} \\
A \setminus B &= \{5\} \\
\mathcal{P}(A) &= \{\emptyset, \{1\}, \{5\}, \{1, 5\}\} \\
A \times B &= \{(1, 1), (1, 2), (1, 3), (5, 1), (5, 2), (5, 3)\} \\
B \times A &= \{(1, 1), (1, 5), (2, 1), (2, 5), (3, 1), (3, 5)\} \\
A \times \mathbb{N} &= \{(1, n) \mid n \in \mathbb{N}\} \cup \{(5, n) \mid n \in \mathbb{N}\} \\
&= \{(1, 0), (5, 0), (1, 1), (5, 1), \ldots (1, i), (5, i), \ldots\}
\end{aligned}$$

Copyright © 2008 Steven D. Johnson

In listing the elements of $A \cup B$, each distinct element is written just once. Consult Definition 1.4 to verify that each element of $\mathcal{P}(A)$ is, in fact, a subset of A.

Ordered pairs $(1,3)$ and $(3,1)$ are unequal, for while they contain the same numbers, these numbers are in a different order. Thus, $A \times B$ and $B \times A$ are distinct sets because, for instance, $(1,3) \in A \times B$ but $(1,3) \notin B \times A$. However $A \times B$ and $B \times A$ are not disjoint; they share the element $(1,1)$.

Example

Ex 1.4 *Let $A = \{a, b\}$; let $B = \{0, 1\}$; and let $C = \{1, 3\}$. Compare the sets $(A \times B) \times C$ and $A \times (B \times C)$*

SOLUTION: First,

$$A \times B = \{(a,0), (a,1), (b,0), (b,1)\}$$

Now, the set $(A \times B) \times C$ is a set of ordered pairs, each of which has an ordered pair in its first position:

$$\begin{aligned}(A \times B) \times C = \{&((a,0),1), ((a,1),1), ((b,0),1), ((b,1),1),\\ &((a,0),3), ((a,1),3), ((b,0),3), ((b,1),3)\ \}\end{aligned}$$

Elements of the set $A \times (B \times C)$ have the same "information content" but a different structure:

$$\begin{aligned}A \times (B \times C) = \{&(a,(0,1)), (a,(1,1)), (b,(0,1)), (b,(1,1)),\\ &(a,(0,3)), (a,(1,3)), (b,(0,3)), (b,(1,3))\ \}\end{aligned}$$

Each of the ordered pairs in $A \times (B \times C)$ has its first element coming from A and its second element coming from $B \times C$.

As Example 1.4 illustrates, compound set products may introduce structure that is not wanted. The next definition extends the notion of "product" to an arbitrary number of sets, as well as other variations of the "set-$\times$" operation.

Definition 1.6 *The* product of n sets*, A_1, A_2, $\ldots$, A_n, is*

$$A_1 \times A_2 \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i,\ 1 \leq i \leq n\}$$

The elements of $A_1 \times A_2 \times A_3$ are called ordered n-tuples*. The n-fold product A^n is*

$$A^n = \overbrace{A \times \cdots \times A}^{n \text{ times}}$$

By convention, A^0 is the empty set.

Copyright © 2008 Steven D. Johnson

Exercises 1.1

1. List the following sets:

(a) $\{2^i \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq 8\}$

(b) $\{i^2 \mid i \in \mathbb{N} \text{ and } 0 \leq i \leq 8\}$

(c) $\{2k + 1 \mid k \in \mathbb{N}\}$

(d) $\{m \mid 23 < m < 29 \text{ and } m \text{ is a prime number}\}$

2. Let $A = \{a, b\}$; let $B = \{1, 2, 3\}$; let $C = \emptyset$; and let $D = \{a, b, c, d\}$. List the following sets:

(a)	$A \cup B$	(f)	$A \cup C$
(b)	$A \cap B$	(g)	$A \cap D$
(c)	$A \times B$	(h)	A^3
(d)	$\mathcal{P}(A)$	(i)	$\mathcal{P}(\emptyset)$
(e)	$B \times \emptyset$	(j)	$(D \cap A) \times B$

3. Let $A = \{a, b\}$; let $B = \{1, 2, 3\}$; and let $E = A \times B$. List the following sets:

(a) $\{(x, y, y) \mid (x, y) \in E\}$

(b) $\{(x, x) \mid x \in E\}$

(c) $\{(y, z) \mid (x, y) \in E \text{ and } z \in B\}$

4. In this book, the set $S = \{1, 2, 2, 3, 3\}$ denotes a three-element set in which 2 and 3 have both been listed twice. So we understand that this another way to describe the set $\{1, 2, 3\}$. Some other books interpret $\{1, 2, 2, 3, 3\}$ as a *five* element *mulitset* allowing multiple occurances of equal elements. Write a version of Definition 1.5 for multisets.

5. Define a set P_n representing the prime divisors of a number n.

Comment: A simple set of numbers does not work because the convention is to allow listing redundant elements. So if one were to define P_{72} to be $\{2, 2, 2, 3, 3\}$, the specified set is actually just $\{2, 3\}$.

The question is asking you to devise a way, using just sets and set operations, to "represent" a number's prime decomposition. Unless you know more about how you are going to *use* this representation, there is no best way to do it. Nevertheless, you should take "elegance," (economy of expression, utility of notation) into consideration.

Copyright © 2008 Steven D. Johnson

1.2 Words and Languages

Any set can be used as an *alphabet* of *symbols*, from which we can build the words, phrases, and sentences of a . For example, if one thinks of how to express decimal numbers in a typical programming language, the alphabet includes decimal digits, a plus-sign, a minus-sign, a decimal point, and an exponentiation symbol for scientific notation. From the alphabet:

$$A = \{\texttt{0},\ \texttt{1},\ \ldots,\ \texttt{9},\ \texttt{+},\ \texttt{-},\ \texttt{.},\ \texttt{e}\}$$

a vocabulary of *numerals* can be specified to express numbers. You can think of a numeral as the "name" of a number. Some examples of numerals accepted by most programming languages are shown below:

```
1126      -65385    1.20    0.314159

1.06e12   -3.5e-2   0       0.0
```

If we think of numerals as corresponding to words in a natural language, then we might think of numeric terms as phrases. Some examples of terms are

```
12 * 3.0        5 * 6 mod 4        12.0 + -3.5

2 + 3 * 6     3 / 4.0 / 1.3e-22      log(0.0)
```

Now, the alphabet for forming terms might consist of

> *all the letters used to make numerals plus* `*` , `/` , `m` , `o` , `d` , `+` , `l` , `g` , `(` , `)` , *and so on.*

Or, we could use numerals as basic symbols, so that the alphabet contains all possible numerals, together with the new symbols `*` , `/` , `(` , `)` , `mod` , `+` , `log` , and so on.
We might even want numbers themselves to serve as symbols, so that the alphabet becomes

> $\mathbb{R}$, *together with the new symbols* `*` , `/` , `(` , `)` , `mod` , `+` , `log` , *and so on.*

Thus, the concept of "alphabet" is general enough to include any collection of elementary symbols, and our notion of "symbol" may include not only textual objects but also other kinds of objects as well.

In this book, `teletype` font is reserved for *concrete syntax*, literal textual symbols. A set of such symbols designating colors might be

$$\mathit{Colors} = \{\texttt{red}, \texttt{orange}, \texttt{yellow}, \texttt{green}, \texttt{blue}, \texttt{violet}\}$$

This is an alphabet of *names* identifying colors, not the colors themselves.

Copyright © 2008 Steven D. Johnson

A set of punctuation symbols used in a programming language is

$$Punctuation = \{\, \texttt{(}\,,\ \texttt{)}\,,\ \texttt{.}\,,\ \texttt{,}\,,\ \texttt{;}\,\}$$

The following definitions provide us the basic mathematical notions used in talking about words.

Definition 1.7 *Let V be any set. The set of* words over V*, denoted by V^+, consists of all finite sequences of symbols from V. V is called the* alphabet *for V^+.*

Example

Ex 1.5 *Let $V = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ and list V^+.*

SOLUTION: The set of words over V is

$$\begin{aligned} V^+ = \{\ & \texttt{a}, \texttt{b}, \texttt{c},\\ & \texttt{aa}, \texttt{ab}, \texttt{ac}, \texttt{ba}, \texttt{bb}, \texttt{bc}, \texttt{ca}, \texttt{cb}, \texttt{cc},\\ & \texttt{aaa}, \texttt{aab}, \texttt{aac}, \texttt{aba}, \texttt{abb}, \texttt{abc}, \texttt{aca}, \texttt{acb}, \texttt{acc},\\ & \texttt{baa}, \texttt{bab}, \texttt{bac}, \texttt{bba}, \texttt{bbb}, \texttt{bbc}, \texttt{bca}, \texttt{bcb}, \texttt{bcc},\\ & \texttt{caa}, \texttt{cab}, \texttt{cac}, \texttt{cba}, \texttt{cbb}, \texttt{cbc}, \texttt{cca}, \texttt{ccb}, \texttt{ccc},\\ & \texttt{aaaa}, \texttt{aaab}, \texttt{aaac}, \texttt{aaba}, \ldots, \texttt{cccc},\\ & \vdots \end{aligned}$$

This listing shows all the one-letter words in the first line, all the two-letter words in the second line, and so forth. Within each line, the words listed systematically, in "alphabetic" order.

Example

Ex 1.6 Let $W = \{5, 17\}$. List W^+.

SOLUTION: In the description below, an explicit *concatenation* mark is used to set individual symbols apart.

$$W^+ = \{5,\ 17,\ 5\hat{}5,\ 5\hat{}17,\ 17\hat{}5,\ 17\hat{}17,\ 5\hat{}5\hat{}5,\ 5\hat{}5\hat{}17, 5\hat{}17\hat{}5,\ 5\hat{}17\hat{}17,\ \ldots\}$$

For instance the word $17\hat{}5\hat{}5$ is composed of the three symbols 17, 5, and 5 from W.

The concatenation symbol is omitted—unless to do so causes confusion—just as the multiplication symbol is omitted in arithmetic formulas. It should be clear that the concatenation of two words is a word.

Definition 1.8 *The* concatenation *of words u and v, is the word formed by juxtaposing u and v, that is, first spelling u and then spelling v; This new word is denoted by $u\hat{}v$, or where possible, simply by uv. We write u^n for*

$$\overbrace{u\hat{}u\hat{}\cdots\hat{}u}^{n\text{ times}}$$

Copyright © 2008 Steven D. Johnson

Example

Ex 1.7 *Let* $V = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ *and consider* V^+*, as defined in Example 1.5. Let* $u = \texttt{aab}$*,* $v = \texttt{cc}$*, and* $w = \texttt{abc}$*.*

$$\begin{aligned} \texttt{cca}\hat{}\texttt{abb} &= \texttt{ccaabb} \\ \texttt{a}\hat{}\texttt{bbbb}\hat{}\texttt{a} &= \texttt{abbbba} \\ (\texttt{a}\hat{}\texttt{c})u &= \texttt{acaab} \\ v\,\texttt{b}\,v &= \texttt{ccbcc} \\ uvw &= \texttt{aabccabc} \\ uuu &= u^3 = \texttt{aabaabaab} \end{aligned}$$

If u, v, and w are words, then

$$u\hat{}(v\hat{}w) = (u\hat{}v)\hat{}w$$

In other words, when three or more words are concatenated, it does not matter whether the concatenation is done from left to right or from right to left (or in any other way) so long as the order is preserved.

It is sometimes useful to include a “word” containing no letters. The following definition provides a symbol for this word.

Definition 1.9 *The* empty word*, over any alphabet, is denoted by* ε*. Given alphabet* V*, for any word* $w \in V^+$*,*

$$\varepsilon\hat{}w = w \quad \text{and} \quad w\hat{}\varepsilon = w$$

The language V^* *includes all words in* V^+ *together with* ε*,*

$$V^* = V^+ \cup \{\varepsilon\}$$

Usually, we are interested in some particular subset of words over an alphabet. For example, not every word over

$$W = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{+}, \texttt{-}, \texttt{.}, \texttt{e}\}^+$$

is a legal numeral. Only those words that satisfy certain spelling laws are legal. A numeral can have at most one decimal point so while `55.27.33` is a word in W, it is not a numeral. By *language* we simply mean a specific subset of words over a given alphabet.

Definition 1.10 *A* language *over alphabet* V *is any subset of* V^+*.*

Copyright © 2008 Steven D. Johnson

Example

Ex 1.8 *Describe the language of* decimal numerals.

SOLUTION: Let

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

represent the set of digits. We will also need punctuation symbols from the set

$$P = \{\texttt{.},\ \texttt{+},\ \texttt{-}\}$$

The components of a numeral are

(a) A *sign*, which may be omitted for a positive number. Hence the sign comes from the set

$$S = \{\texttt{+},\ \texttt{-},\ \varepsilon\}$$

(b) the integer part, a string of one or more digits, D^+.

(c) the fractional part, if present, is a period followed by a string of zero or more digits,

$$E = \{\varepsilon\} \cup \{\texttt{.}\hat{}f \mid f \in D^*\}$$

Putting these together, the language of decimal numerals is described by

$$Numerals = \{s\hat{}m\hat{}f \mid s \in S,\ m \in D^+, \text{ and } f \in E\}$$

Here are some word instances in *Numeral*:

(a) The word `+2.731` is a valid numeral. It breaks down into a sign, integral part, and factional part according to the specification expression

$$\underset{S}{\boxed{\texttt{+}}}\hat{}\underset{D^*}{\boxed{\texttt{2}}}\hat{}\underset{E}{\boxed{\texttt{.735}}}$$

(b) The word `42` is also valid, having an empty sign and fractional part

$$\boxed{\varepsilon}\boxed{\texttt{42}}\boxed{\varepsilon}$$

(c) The words `55.126.99`, `++5` and ε do not satisfy the description and hence are not in the specified language

Copyright © 2008 Steven D. Johnson

Exercises 1.2

1. Let $V = \{\texttt{a}, \texttt{b}, \texttt{\$}\}$. For each of the following languages $L_i \subseteq V^+$, list enough elements to make it clear what each contains.

(a) In language L_1 each word has exactly one `$` and equally many `a` s as `b` s.

(b) In each word of language L_2, `a` s and `b` s alternate with any number of `$` s mixed in.

(c) In each word of language L_3, no `a` occurs next to a `b` .

(d) $L_4 = \{u\hat{}\texttt{\$}\hat{}v \mid u \in \{\texttt{a}\}^+ \text{ and } v \in \{\texttt{\$}, \texttt{b}\}^+\}$

(e) $L_5 = \{\texttt{a}^k\hat{}\texttt{\$}\hat{}\texttt{b}^k \mid k \in \mathbb{N}\}$

2. In Exercise 1.5 the listing of $\{\texttt{a}, \texttt{b}, \texttt{c}\}^+$ shows that there are 3 one-letter words and 9 two-letter words. The third and fourth rows of the listing do not show all the possible words. How many words would there be in the third row; that is, how many three-letter words are there in $\{\texttt{a}, \texttt{b}, \texttt{c}\}^+$? How many four-letter words? How many n-letter words?

1.3 A Simple Algorithmic Language

A very simple programming language is used later this book. We introduce the language informally in this section; in Chapter 10, once we have the needed mathematical foundations, we examine this language in more detail. It is a structured, sequential *language of statements*, similar in form to many languages that exist today, such as C and Java. It is assumed that you have some experience with, and intuition about, programming in this kind of language. There are just four kinds of statements.

1. The *assignment statement*, has the form

$$v := E$$

The object to the left of the assignment symbol is called an *identifier*, or sometimes *program variable* (but never just *"variable"*). To the right is an expression, E, whose value is calculated and then associated with the program variable from that point on. Program variables can be simple names, such as `x` and `answer` , or array references, such as `a[i]` and `b[5, j]` .

2. A *conditional statement* has the form

$$\texttt{if } T \texttt{ then } S_1 \texttt{ else } S_2$$

Where S_1 and S_2 are, themselves, statements. If the test T holds then statement S_1 is executed; otherwise, statement S_2 is executed;

3. A *repetition statement* has the form

$$\texttt{while } T \texttt{ do} S$$

   Statement S is repeatedly executed so long as the test T remains true.

4. Finally, a *compound statement* has the form

$$\texttt{begin} S_1 \texttt{;} S_2 \texttt{;} \; \ldots \; \texttt{;} S_n \texttt{end}$$

   Statements S_1, S_2, ..., S_n are executed in order.

Figure 1.3 shows an equivalent specification of the *Statement* language. It uses *Backus-Naur* notation, or "BNF," a form often seen in programming manuals. Unlike the description above, Figure 1.3 says nothing about what statements *mean*, only what they look like. BNF only describes what sentences are syntactically correct.

Both descriptions are *self referencing*, meaning that they refer to the language in the process of defining it. Conditional, repetition, and compound statements *contain* statements. Although self reference used in this way may seem natural and intuitive, *not all self-referencing definitions are meaningful.* For example, consider the language described by items 1–3 above, leaving out the assignment statement. Can you give an example of a program in that language?

The statement language has no input/output operations. We can talk about the result of a program in terms of the final values of its identifiers. Here is an example of a program in the language of statements:

```
      { A,  B ∈  ℕ }
𝒫:    begin
      x := A;
      y := B;
      z := 0;
ℓ1:   while  x ≠ 0 do   { This is the loop ℓ1 }
         begin
         x := x − 1;
ℓ2:      z := z + y
         end
      end
      { z = AB }
```

The program labels, $\mathcal{P}$, ℓ_1 and ℓ_2 are not part of the language—there is no `goto` statements so labels aren't needed—but are used in discussions to refer to points of the program.

Comments written between braces, $\{ \cdots \}$, are called *assertions*. For now, comments are optional, but in Chapter 10 they become a formal part of the language, used to reason logically about a program's data state. After some staring perhaps, it should be clear that program $\mathcal{P}$ computes the product of natural numbers A and B, as the comments assert.

Exercises 1.3

1. In the programming language just described, write programs for each of the following specifications.

(a) Assume that program variables x and y have been initialized with values in $\mathbb{N}$. Compute the sum of these values, leaving the result in program variable z, using only the operations of adding or subrtracting 1 to (from) a program variable.

(b) Assume that program variables x and y have been initialized with values in $\mathbb{N}$. Compute the product of x and y using only the operations of addition and subtraction.

(c) Assume that program variables x and y have been initialized with values, A and B respectively, in $\mathbb{N}$. Compute the value A^B using only addition and multiplication.

(d) Write a program to compute the quotient, q, and remainder, r of two values initially held in variables x (the dividend) and y, the divisor. Assume that you have only addition and subtraction.

(e) Write a program to compute the *greatest common divisor*, $\gcd(x, y)$, of $A,\ B \in \mathbb{N}$ held in program variables x and y respectively, using only addition and subtraction.

(f) Assume that in addition to '+' and '−' you also have an operation, *half*(v) that divides its operand, v by two. Use this operation to improve the performance of the gcd program of the previous exercise.

Copyright © 2008 Steven D. Johnson

⟨STMT⟩	::=	⟨IVS⟩ `:=` ⟨TERM⟩	(*assignment*)
	|	`if` ⟨QFF⟩ `then` ⟨STMT⟩ `else` ⟨STMT⟩	(*conditional*)
	|	`while` ⟨QFF⟩ `do` ⟨STMT⟩	(*repetition*)
	|	`begin` ⟨STMT⟩ `;` ⟨STMT⟩ `end`	(*compound*)
⟨IVS⟩	::=	⋯	(*identifier*)
⟨TERM⟩	::=	⋯	(*expression*)
⟨QFF⟩	::=	⋯	(*test*)

Figure 1.1: Partial description of the *Statement* programming language STMT, expressed in Backus-Naur form (Sec. 8.5)

Copyright © 2008 Steven D. Johnson

Chapter 2

Propositional Logic and Boolean Algebra

2.1 Propositions and Truth Tables

A *proposition* is a statement of fact, a sentence to which a value of *true* or *false* can be assigned. Compound propositions are built from simpler propositions using *logical connectives*, such as "and," "or," and "implies." A *propositional formula* is an expression involving simple propositions and logical connectives. Suppose that P and Q stand for propositional formulas. Then the following are also propositional formulas:

$\neg P$	*negation*	$P \Rightarrow Q$	*implication*
$P \wedge Q$	*conjunction*	$P \Leftrightarrow Q$	*coincidence*
$P \vee Q$	*disjunction*	$P \not\Leftrightarrow Q$	*exclusive-or*

Figure 2.1 shows some of the ways these connectives are expressed in English. The following definition tells what the connectives mean.

Definition 2.1 *The tables below define how the propositional connectives are interpreted.*

P	$\neg P$
F	T
T	F

P	Q	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$	$P \not\Leftrightarrow Q$
F	F	F	F	T	T	F
F	T	F	T	T	F	T
T	F	F	T	F	F	T
T	T	T	T	T	T	F

2.1.1 Implication*

The definition of *implication*, $P \Rightarrow Q$, sometimes becomes confusing when considered in isolation. Since explaining it may simply compound the confusion,

$\neg P$	It is not the case that P. P does not hold. Not P.
$P \wedge Q$	P and Q. P but Q.
$P \vee Q$	P or Q. either P or Q or both at least one of P and Q
$P \Rightarrow Q$	P implies Q. if P then Q. Q whenever P. Q if P. P only if Q. P is sufficient for Q. Q is necessary for P. Q follows from P.
$P \Leftrightarrow Q$	P if and only if Q. P iff Q. P exactly when Q. P is necessary and sufficient for Q. Whenever P then Q and conversely.
$P \not\Leftrightarrow Q$	Either P or Q but not both. Exactly one of P and Q. P exclusive-or Q.

Figure 2.1: The logical connectives and some corresponding English utterances.

Copyright © 2008 Steven D. Johnson

you may wish to defer reading this section until a question like *"What does ⇒ really mean?"* comes to mind and motivates you to read about it.

We are not accustomed to thinking about what "P implies Q" should mean when the *antecedent* P is known to be false or when the *consequent* Q is known to be true. In most mathematical arguments, there is a connection between the two. P must be *used* to carry the argument through. The following examples illustrate why the definition of implication is natural.

Example

Ex 2.1 Proposition *If A is any set, then $\emptyset \subseteq A$.*

PROOF: According to Definition 1.4, $\emptyset \subset A$ is true provided, "every element of $\emptyset$ is also an element of A." This means that the statement $x \in \emptyset \Rightarrow x \in A$ must be valid, no matter what element is chosen for x. But no matter what x is, "$x \in \emptyset$" is a false statement. In other words, "$\emptyset \subseteq A$" logically reduces to $F \Rightarrow x \in A$, and by Definition 2.1, this proposition is *true* whether or not $x \in A$. ∎

Another way to put it is that no choice of x exists that can be used falsify "$x \in \emptyset \Rightarrow x \in A$." It cannot be false, so it must be true. We say that the proposition holds *vacuously*, since the antecedent is *logically false*.

Example

Ex 2.2 Proposition *If A is any set, then $A \subseteq A$.*

PROOF: According to Definition 1.4, $A \subseteq A$ means that the statement $x \in A \Rightarrow x \in A$ must be valid no matter what x is. But if "$x \in A$" is true, then the statement reduces to $T \Rightarrow T$, and if $x \notin A$ we have $F \Rightarrow F$. In either case, the proposition is *true*. ∎

We say that the proposition is *tautologically valid* because it reduces to a purely logical trueism. See Definition 2.2 later in this chapter.

Example

Ex 2.3 Proposition *For all $n, m \in \mathbb{Z}$, if $a > 0$ and $b > 0$ then $(a+b)^1 \geq a^1+b^1$.*

PROOF: By definition, raising any number to the "first power" yields the same number. In other words, for any $z \in \mathbb{Z}$, $z^1 = z$. Thus,

$$(a+b)^1 = a+b = a^1+b^1$$

so it follows immediately that $(a+b)^1 \geq a^1+b^1$, as desired. ∎

Copyright © 2008 Steven D. Johnson

The antecedent "$a > 0$ and $b > 0$" is irrelevant, the truth of the proposition does not depend on whether or not a and b is positive. (Had the proposition been $(a+b)^2 \geq a^2 + b^2$, the antecedent *would* be relevant.)

We say that this proposition holds *trivially*, that is, the consequent holds independently of the antecedent.

A good way to think about $P \Rightarrow Q$ is that it says, *"either P is false or Q is true, or possibly both."* Or, *"it is never the case that Q is true and P is false.* Or, Q *(is true) only if* P *(is also true)."*

2.2 Truth Tables

Definition 2.1 gives us the means to evaluate complex propositions. We do so by first evaluating the innermost terms and then working outward. We keep track of intermediate results in a *truth table*, similar to the table shown in the definition.

Example

Ex 2.4 *Evaluate the formula* $(P \vee R) \Rightarrow Q$*.*

A truth table for this formula includes one row for each combination of truth values that might be assigned to its sub-formulas. In this case there are eight possibilities. To the right is the evaluation of the formula. In this example, sub-terms $P \vee R$ and Q are evaluated first, and then the '$\Rightarrow$' is evaluated.

P	Q	R	$(P \vee R)$	$\Rightarrow$	Q
F	F	F	F	T	F
F	F	T	T	F	F
F	T	F	F	T	T
F	T	T	T	T	T
T	F	F	T	F	F
T	F	T	T	F	F
T	T	F	T	T	T
T	T	T	T	T	T
			(1)	(3)	(2)

According to the table, there are three cases in which the proposition $(P \vee R) \Rightarrow Q$ is false.

Truth tables are sometimes used to analyze "story problems." Consider the following example:

Example

Ex 2.5 *Dick, Jane, and Sally are working together on a programming project. Dick says, "Sally's routine is correct; but my routine is correct only if Jane's is." Sally says, "If my routine has a bug, so does Dick's; but my routine is correct." Jane says, "Either Dick's routine has a bug or Sally's does; but not both." Assuming all three are telling the truth, whose routine has a bug? Whose is correct? Assuming all the routines are correct, who's not telling the truth?*

Let us identify the atomic propositions. Define[1] D, J and S as follows:

$$\begin{aligned} D &\equiv \text{"Dick's routine is correct."} \\ J &\equiv \text{"Jane's routine is correct."} \\ S &\equiv \text{"Sally's routine is correct."} \end{aligned}$$

The assertions made by the three programmers are

$$\begin{aligned} \text{Dick:} \quad & S \wedge (D \Rightarrow J) \\ \text{Jane:} \quad & D \not\Leftrightarrow S \\ \text{Sally:} \quad & (\neg S \Rightarrow \neg D) \wedge S \end{aligned}$$

and we are interested in the truth of the proposition $D \wedge J \wedge S$. Here is a truth table:

D	J	S	$S \wedge (D \Rightarrow J)$	$D \not\Leftrightarrow S$	$(\neg S \Rightarrow \neg D) \wedge S$	
F	F	F	F	F	F	
F	F	T	T	T	T	⋆
F	T	F	F	F	F	
F	T	T	T	T	T	⋆
T	F	F	F	T	F	
T	F	T	F	F	T	
T	T	F	F	T	F	
T	T	T	T	F	T	

For both cases in which Dick's, Jane's, and Sally's statements are true, D is false and S is true. Thus, we can conclude that, Dick's routine has a bug and Sally's does not, provided all three programmers are telling the truth. We cannot draw any conclusion about Jane's routine. The last row of the truth table is the case where all three routines are correct; and in that row, Jane's statement is false.

Propositions may be characterized and compared using truth tables. Our study of mathematical reasoning in later sections involves truth-table analysis of the assertions made in proofs. The next two definitions provide a basic vocabulary for classifying propositions.

[1] Throughout this book a triple-equals sign is used when names are assigned to formulas. There is more about this in Chapter 6.

Copyright © 2008 Steven D. Johnson

Definition 2.2 *A proposition is called a* tautology *when all rows of its truth table evaluate to T. A proposition is called a* contradiction *when all rows of its truth table evaluate to F. A proposition which is neither a tautology nor a contradiction is called a* contingency.

Definition 2.3 *Two propositions are said to be* logically equivalent *when their truth tables are identical.*

How many logical connectives do we need? As the following proposition says, all the connectives we have defined can be implemented using only negation and disjunction.

Proposition 2.1 *If P and Q are propositions then the following pairs of formulas are logically equivalent:*

$$\begin{array}{lrcl} \text{(a)} & P \Rightarrow Q & \text{and} & (\neg P) \vee Q \\ \text{(b)} & P \wedge Q & \text{and} & \neg((\neg P) \vee (\neg Q)) \\ \text{(c)} & P \Leftrightarrow Q & \text{and} & (P \Rightarrow Q) \wedge (Q \Rightarrow P) \\ \text{(d)} & P \not\Leftrightarrow Q & \text{and} & \neg(P \Leftrightarrow Q) \end{array}$$

PROOF:

In each case logical equivalence is established by a comparison of truth tables, as specified in Definition 2.3. The tables for part (a) are shown below and the rest of the proof is left as an exercise.

P	Q	$P \Rightarrow Q$	$(\neg P)$	$\vee$	Q
F	F	T	T	T	F
F	T	T	T	T	T
T	F	F	F	F	F
T	T	T	F	T	T

∎

Other sufficient sets of connectives are developed as exercises. There is much more to say about propositions. But now we have enough information about them to consider the next topic of this chapter.

Exercises 2.2

1. Let P stand for the proposition "Sue says it." Let Q stand for the proposition "Sam saw it." Let R stand for the proposition "Sid did it." Express the following sentences as formulas involving the logical connectives. If there is more than one way to translate a sentence, use truth tables to explain any differences in the meaning among these translations.

(a) Sid did it, Sam saw it, and Sue says it.

(b) If Sid did it, Sam saw it.

Copyright © 2008 Steven D. Johnson

(c) Sid did it only if Sam saw it.

(d) Sue says it only if Sid did it, and Sam saw it.

(e) If Sue says it implies Sam saw it, Sid did it.

2. Definition 2.1 gives the meaning of five logical operations of two arguments. How many distinct logical connectives of two arguments are there?

3. Consider the logical operation defined below:

P	Q	$P \downarrow Q$
F	F	T
F	T	T
T	F	T
T	T	F

Show that '↓' can be used to implement (in the sense of Prop. 2.1) all of the operations of Definition 2.1.

4. Determine another logical operation, different than ↓, which can be used to implement all of the operations of Definition 2.1.

2.3 Boolean Algebra

Digital computers are based on systems in which there are just two values. Electronically the two values are realized as voltage levels (*voltage* is a measure of electrical force). All the components of a digital system are carefully designed to produce and respond to just these two levels. Mathematically, the binary values of a digital system are represented as a two-element set of binary digits, or *bits*, $\{0, 1\}$. The basic operations on bits are defined below.

Definition 2.4 *The operations of* inversion*,* addition*, and* multiplication*, defined on bits, are given by the following tables.*

$\overline{x}$	
0	1
1	0

$\cdot$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	1

The multiplication sign is dropped where possible, so that the expression

$$(\overline{x} \cdot y) + \overline{(x \cdot \overline{y})}$$

is usually written

$$\overline{x}\, y + \overline{x\, \overline{y}}$$

The next proposition states a number of algebraic laws that are valid for the operations just defined.

Proposition 2.2 *Assume variables x, y, and z range over bits, and let the operations of negation, addition, and multiplication be as defined in Definition 2.4. These operations obey the following algebraic identities.*

BOOLEAN IDENTITIES		
Negation	$\overline{\overline{x}} = x$	
Identity	$0 + x = x$	$1 \cdot x = x$
Dominance	$1 + x = 1$	$0 \cdot x = 0$
Idempotence	$x + x = x$	$x \cdot x = x$
Cancellation	$x + \overline{x} = 1$	$x \cdot \overline{x} = 0$
Commutativity	$x + y = y + x$	$x \cdot y = y \cdot x$
Associativity	$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Distributivity	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$
DeMorgan	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	$\overline{(x \cdot y)} = \overline{x} + \overline{y}$

PROOF: Each of the laws can be verified by comparing truth tables according to Definitions 2.4 and 2.3. ∎

Definitions 2.4 and **??** form a system in which we can reason about equality. As the exercises at the end of this section illustrate, we can generalize this binary *algebra*, or system of laws, to other structures.

Definition 2.5 *A set B containing distinguished elements 1 and 0, and having operations '$\cdot$', '$+$', and '$\overline{\ }$' which satisfy the laws of Proposition 2.2 for x, y, and z ranging over B, is called a* Boolean Algebra.

Example

Ex 2.6 *Use the boolean identities to show that $a(a + b) = a$.*

We shall show this by performing a derivation starting from the left-hand side.

$$\begin{aligned} a(a+b) &= (a+0)(a+b) && (\textit{identity}) \\ &= a + 0b && (\textit{distributivity}) \\ &= a + 0 && (\textit{dominance}) \\ &= a && (\textit{identity}) \end{aligned}$$

Example

Ex 2.7 *Use the boolean identities to show that $a + ab = a$.*

Compare this equation with the one in Example 2.6. It is the *dual* of the one shown here, obtained by interchanging addition and multiplication. We can already conclude that it is valid by the previous derivation, since there must be a dual derivation to prove it. Here it is:

$$\begin{aligned} a + ab &= a1 + ab && (identity) \\ &= a(1 + b) && (distributivity) \\ &= a1 && (dominance) \\ &= a && (identity) \end{aligned}$$

2.3.1 Duality

The laws dealing with bit addition and bit multiplication come in pairs. For every identity that holds for addition, there is a corresponding identity for multiplication, and conversely. This property is called *duality.*

We have already suggested a correlation between the boolean values and operations, and truth values with logical operations. For if we think of the bit 0 as meaning *false* and the bit 1 as meaning *true,* then inversion, addition, and multiplication, implement negation, disjunction, and conjunction, respectively. Compare the following tables with those in Definition 2.1:

$\neg x$	
F	T
T	F

$\wedge$	F	T
F	F	F
T	F	T

$\vee$	F	T
F	F	T
T	T	T

But there is another way to associate truth values with bits. Let bit 0 represent *true* and bit 1 represent *false.* In this case, bit addition implements logical conjunction and bit multiplication implements disjunction:

$\neg x$	
T	F
F	T

$\vee$	T	F
T	T	T
F	T	F

$\wedge$	T	F
T	T	F
F	F	F

Associating *true* with bit 1 is called the *positive logic interpretation.* and associating *false* with 1 is called the *negative logic interpretation.* Digital designers commonly use both interpretations. Even so, the '+' sign is called "or" and the '·' sign "and."

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

2.4 Normal Forms

Consider the truth table, given below, for the proposition $((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Leftrightarrow (P \Rightarrow R)$

P	Q	R	$((P \Rightarrow Q)$	$\wedge$	$(Q \Rightarrow R))$	$\Leftrightarrow$	$(P \Rightarrow R)$
F	F	F	T	$T^\bullet$	T	T	T
F	F	T	T	$T^\bullet$	T	T	T
F	T	F	T	F	F	F	T
F	T	T	T	$T^\bullet$	T	T	T
T	F	F	F	F	T	T	F
T	F	T	F	F	T	F	T
T	T	F	T	F	F	T	F
T	T	T	T	$T^\bullet$	T	T	T

Comparing the truth tables for $A \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow R)$ and $B \equiv (P \Rightarrow R)$ one can see it is *not* the case that A eq B.

Look now at the truth table for formula A. It is true in four cases and false in the other four. We can construct a formula describing just the *true* cases by recording the truth values of P, Q and R.

A is true just when

$P = F$, $Q = F$ and $R = F$, or
$P = F$, $Q = F$ and $R = T$, or
$P = F$, $Q = T$ and $R = T$, or
$P = T$, $Q = T$ and $R = T$.

In each case, a formula can be written singling out exactly that case

A is true just when

$(\neg P) \wedge (\neg Q) \wedge (\neg R)$ is true, or
$(\neg P) \wedge (\neg Q) \wedge R$ is true, or
$(\neg P) \wedge Q \wedge R$ is true, or
$P \wedge Q \wedge R$ is true.

Equivalently,

A is true just when

$(\neg P \wedge \neg Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge R) \vee (P \wedge Q \wedge R)$ is true

That is,

A eq $(\neg P \wedge \neg Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge R) \vee (P \wedge Q \wedge R)$

It should be evident that one can extract such a formula from any truth table, and is, in essence, just a way of describing the truth table. The result is always disjunction of *and-clauses*, each of which contains every variable just once in either positive or negated instance.

The formula thus derived is called the *disjunctive normal form* (*DNF*) of the original expression, A in this case. Since every proposition has a DNF, we have shown that for any logical expression, there is an equivalent one expressed in terms of '$\wedge$', '$\vee$' and '$\neg$', as suggested by Proposition 2.1.

We have now seen two essentially equivalent ways to represent propositions in a way a way that makes them easier to analyze or compare: truth tables and DNFs. We will see others later in this book. Both of them suggest computer encodings that could be used in automating the analyses.

(a) An array of 1s and 0s could be used to represent the truth table. For the proposition A

1	1	0	1	0	0	0	1

(b) A list of 3-bit quantities, one for each clause in the DNF. For the proposition A the list would be

```
(000 001 011 111)
```

In both cases, the encodings contain the essential information characterizing A. However, to interpret the encodings, one needs additional information about *how many* variables are used in the formula and *in what order* those variables are used in developing the truth table. We will see when we look at other representations, later in this book, that more compact computer representations always entail prior knowledge about the number and order of the variables.

Exercises 2.4

1. Prove Proposition 2.2

2. Reduce the following boolean expressions to simpler terms

(a) $xy + (x + y)\overline{z} + y$

(b) $x + y + \overline{(\overline{x} + y + z)}$

(c) $yz + wx + z + [wz(xy + wz)]$

3. Define $x \oplus y$ to be $x\overline{y} + \overline{x}y$. Use boolean algebra to prove

(a) $x \oplus y = \overline{x} \oplus \overline{y}$

(b) $x(y \oplus z) = xy \oplus xz$

(c) $\overline{(x \oplus y)} = \overline{x} \oplus y$

Copyright © 2008 Steven D. Johnson

4. Let $A = \{a, b, c\}$ and define the following correspondence for $\mathcal{P}(A)$:

$$\begin{array}{rcl} 1 & \mapsto & A \\ 0 & \mapsto & \emptyset \\ \overline{X} & \mapsto & A \setminus X \\ X \cdot Y & \mapsto & X \cap Y \\ X + Y & \mapsto & X \cup Y \end{array}$$

Show that this correspondence forms a Boolean algebra, according do Definition 2.5.

5. Let D be the set of numbers that divide 30, $D = \{1, 2, 3, 5, 6, 10, 15, 30\}$, and define the following correspondence.

$$\begin{array}{rcl} 1 & \mapsto & 30 \\ 0 & \mapsto & 1 \\ \overline{x} & \mapsto & 30 \div x \\ x \cdot y & \mapsto & \text{the greatest common divisor of } x \text{ and } y \\ x + y & \mapsto & \text{the least common multiple of } x \text{ and } y \end{array}$$

Show that this correspondence forms a Boolean algebra.

6. Show that the connectives '$\wedge$', '$\vee$', and '$\neg$' form a Boolean algebra under a notion of equality that says, $P = Q$ iff P is logically equivalent to Q.

7. Construct truth tables and DNFs for the following propositional formulas

(a) $(P \wedge (P \Rightarrow Q)) \Rightarrow Q$

(b) $((P \Rightarrow R) \wedge (Q \Rightarrow R)) \Leftrightarrow ((P \wedge Q) \Rightarrow R)$

(c) $((P \Rightarrow R) \vee (Q \Rightarrow R)) \Rightarrow ((P \vee Q) \Rightarrow R)$

(d) $((P \Rightarrow R) \vee (Q \Rightarrow S)) \Rightarrow ((P \vee Q) \Rightarrow (R \vee S))$

8. The term *disjunctive normal form* suggests that there might be such a thing as *conjunctive normal form* (*CNF*), and there is. What would the CNF of a formula look like? Devise a systematic way to synthesize a CNF from the truth table of a propositional formula.

2.5 Application of Boolean Algebra to Hardware Synthesis*

The Boolean algebra for $\{1, 0\}$ has applications to the description of digital hardware. Addition of binary numbers involves the same column-by-column

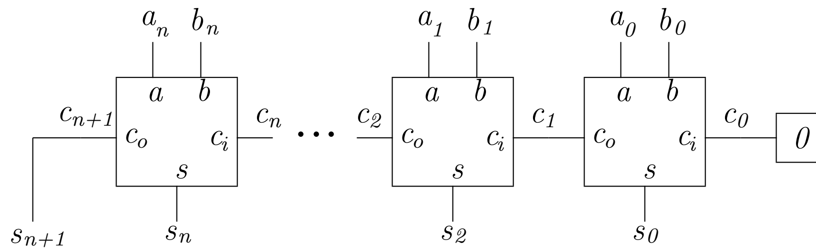
Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

procedure as decimal addition, except that the arithmetic is base 2. For instance to add the numbers 1111001_2 and 101010_2,

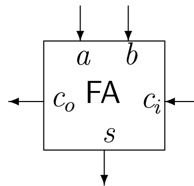
1	1	1	1	0	0	0	(0)	*carry*
	1	1	1	1	0	0	1	*a*
+		1	0	1	0	1	0	+ *b*
1	0	1	0	0	0	1	1	*sum*

start at the least significant position adding the two right-most digits. If the result is 10_2 or 11_2, the "2's place" is carried into the next column. To make the algorithm uniform, we start out with a *carry-in* of 0; and if the most-significant *carry-out* is 1, another place is given to the sum.

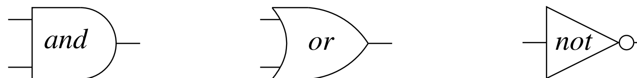
The procedure is implemented in hardware by connecting a series of identical single-bit *full adders*, one for each bit of the operands.



So building an n-bit adder requires designing a full adder and replicating it n times.



Digital hardware is built using a set of devices, or *logic gates*, that operate on two distinct voltage levels, called *high* (H) and *low* (L). The actual voltage values depend on the technology used to make the devices[2] The simplest of these devices realize the functions *and*, *or* and *not*, which are represented by the schematic symbols



[2]In integrated circuits, H is 3-5 volts and L is 0 volts relative to a reference voltage called *ground*. Of course, in physical devices the voltages are not exact, and may range a bit from their ideal values.

Copyright © 2008 Steven D. Johnson Draft Material (v08r0) January 26, 2009

The truth table below specifies what the Full Adder does. It has two outputs, s for the *sum* and c_o for the carry-out, so two truth tables are needed. required

a	b	c_i	s	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

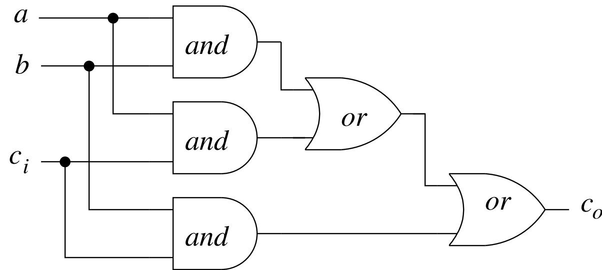
If we express these tables in disjunctive normal form (sometimes called *sum-of-products* form in this context), a naive implementation is obtained, since we have logic gates to realize each operation.

$$\begin{aligned} s &= \bar{a}\,\bar{b}\,c + \bar{a}\,b\,\bar{c} + a\,\bar{b}\,\bar{c} + a\,b\,c \\ c_o &= \bar{a}\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + a\,b\,c \end{aligned}$$

However, it would be better to reduce these formulas to something smaller, in order to use fewer gates. One way to do this is to reduce the expressions algebraically to equivalent but smaller formulas. The c_o output can be reduced to

$$\begin{aligned}
c_o &= \bar{a}\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + a\,b\,c && \text{(truth table)} \\
&= \bar{a}\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + (a\,b\,c + a\,b\,c) && \text{(idempotence)} \\
&= \bar{a}\,b\,c + a\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + a\,b\,c && \text{(commutativity, associativity)} \\
&= \bar{a}\,b\,c + a\,(\bar{b}\,c + b\,\bar{c} + b\,c) && \text{(distributivity)} \\
&= \bar{a}\,b\,c + a\,b\,c + a\,(\bar{b}\,c + b\,\bar{c} + b\,\bar{c} + b\,c) && \text{(idempotence)} \\
&= \bar{a}\,b\,c + a\,b\,c + a\,((\bar{b} + b)\,c + b\,(\bar{c} + c) && \text{(distributivity, twice)} \\
&= \bar{a}\,b\,c + a\,b\,c + a\,(c + b) && \text{(idempotence)} \\
&= (\bar{a} + a)\,b\,c + a\,(c + b) && \text{(distributivity)} \\
&= 1\,b\,c + a\,(c + b) && \text{(cancellation)} \\
&= b\,c + a\,(c + b) && \text{(identity)} \\
&= b\,c + a\,c + a\,b && \text{(distributivity)}
\end{aligned}$$

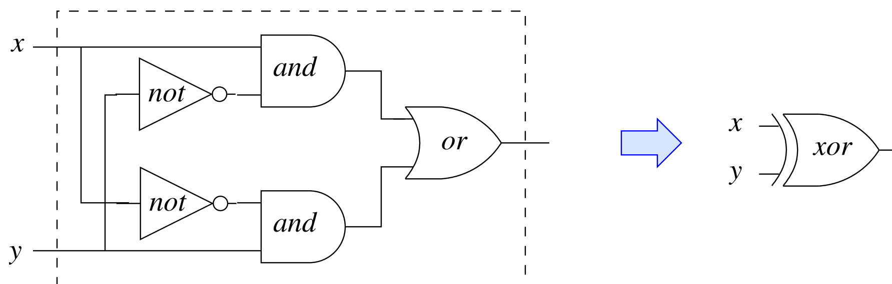
Hence, c_o is implemented with



The derivation of c_o followed the typical pattern of enlarging the formula so that it could later be simplified. Such algebraic manipulations are often done with a goal in mind, and reaching that goal may involve expansion, even if the ultimate objective is reduction.

"Simplification" of s is even more subtle. Suppose we have a device that realizes the the *exclusive-or* operation,

$$x \oplus y \stackrel{\text{def}}{=} \bar{x}\, y + x\, \bar{y}$$



The goal now is to derive and equivalent expression that uses instances of '*oplus*'

. . .

$$
\begin{aligned}
s &= \bar{a}\,\bar{b}\,c + \bar{a}\,b\,\bar{c} + a\,\bar{b}\,\bar{c} + a\,b\,c && \text{(truth table)} \\
&= \bar{a}\,(\bar{b}\,c + b\,\bar{c}) + a\,(\,\bar{b}\,\bar{c} + b\,c\,) && \text{(distributivity, twice)}
\end{aligned}
$$

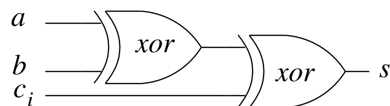
$$
\boxed{
\begin{aligned}
&\bar{b}\,\bar{c} + b\,c \\
&\quad= \overline{\overline{\bar{b}\,\bar{c} + b\,c}} && \text{(negation)} \\
&\quad= \overline{\overline{(\bar{b}\,\bar{c})}\;\overline{(b\,c)}} && \text{(DeMorgan's Law)} \\
&\quad= \overline{(\bar{\bar{b}} + \bar{\bar{c}})\;(\bar{b} + \bar{c})} && \text{(DeMorgan's Law)} \\
&\quad= \overline{(b + c)\,(\bar{b} + \bar{c})} && \text{(negation)} \\
&\quad= \overline{(b + c)\bar{b} + (b + c)\,\bar{c}} && \text{(distributivity)} \\
&\quad= \overline{b\,\bar{b} + c\bar{b} + b\,\bar{c} + c\,\bar{c}} && \text{(distributivity)} \\
&\quad= \overline{0 + c\bar{b} + b\,\bar{c} + 0} && \text{(cancellation)} \\
&\quad= \overline{\bar{b}c + b\,\bar{c}} && \text{(identity, commutativity)}
\end{aligned}
}
$$

$$
\begin{aligned}
&= \bar{a}\,(\bar{b}\,c + b\,\bar{c}) + a\,\overline{(\bar{b}\,c + b\,\bar{c})} && \text{(boxed derivation)} \\
&\ \ \vdots
\end{aligned}
$$

A *subsidiary* derivation is used to refine the sub-formula $\bar{b}\,\bar{c} + b\,c$ to $\bar{b}\,c + b\,\bar{c}$. Continuing with with the derivation,

$$
\begin{aligned}
&\ \ \vdots \\
&= \bar{a}\;(\bar{b}\,c + b\,\bar{c}) + a\;\overline{(\bar{b}\,c + b\,\bar{c})} \\
&= \bar{a}\,(b \oplus c) + a\,\overline{(b \oplus c)} && \text{(definition `}\oplus\text{')} \\
&= a \oplus (b \oplus c) && \text{(definition `}\oplus\text{' (!))}
\end{aligned}
$$

The implementation of s becomes



The second instance of '$\oplus$' involves sub-expressions rather than simple variables. So even in boolean algebra, derivations can become very complex, as you already know from ordinary algebra.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Chapter 3

Counting

One often wants to know how many elements a set contains. In fact, knowing this number is often more important than knowing just what the elements are! Definition 3.1, introduces notation and terminology relating to a set's size.

Section 3.2 lays a foundation for *counting* the elements in a set. This is usually done by posing a kind of experiment in which the question, "How many elements are in set S?" is decomposed into a sequence of simpler decisions for which the number of outcomes is already known. One then tallies all the outcomes to get a final answer.

3.1 Cardinality

Definition 3.1 *$|S|$ denotes the number of elements in S. $|S|$ is called the* size *or* cardinality *of S.*

For example, $|\{a, b, c, d, e\}| = 5$, $|\emptyset| = 0$; and

$$|\{p \mid p \text{ is a prime number less than } 30\}| = 10$$

Fact 3.1 *Let A and B be finite sets.*

(a) $|A \cup B| = |A| + |B| - |A \cap B|$

(b) $|A \cap B| = |A| + |B| - |A \cup B|$

(c) $|A \times B| = |A| \cdot |B|$

(d) $|A \setminus B| = |A| - |A \cap B|$

Comparing this list with the set operations defined in Definition 1.5, $|\mathcal{P}(A)|$ is missing. We will calculate $|\mathcal{P}(A)|$ later.

3.2 Permutations and Combinations

Let $A = \{a, b, c\}$ and consider the different orders that A's elements can be listed. Try it yourself, and compare your answer with

$$\begin{array}{c} abc \\ acb \\ \\ bac \\ bca \\ \\ cab \\ cba \end{array}$$

Let $A = \{a, b, c, d\}$ and consider the different orders that A's elements can be listed. Try it yourself, and check your answer with

$abcd$	$bacd$	$cabd$	$dabc$
$abdc$	$badc$	$cadb$	$dacb$
$acbd$	$bcad$	$cbad$	$dbac$
$acdb$	$bcda$	$cbda$	$dbca$
$adbc$	$bdac$	$cdab$	$dcab$
$adcb$	$bdca$	$cdba$	$dcba$

The listing is organized according to the choice of which letter is first. Having made that choice, all orderings for the remaining three letters are listed. This is a problem we have already solved: there are six possibilities. Figure 3.1(a) shows a tree, labeled to show how the listing is organized. Each path *through* the tree (i.e., from the root to a leaf) represents one letter ordering, as determined by reading the edge labels in order, Trees used in this way are called *decision trees*. along the path.

If we are interested only in the number of solutions, and not what they are, the decision tree can be reduced in size by taking symmetries into account. In Figure 3.1(a) the subtrees at any level are isomorphic, differing only in their labels. So instead of drawing all of them, we just keep track of how many of them there are. Figure 3.1(b) depicts a decision tree in which the edges are labeled with the number of isomorphic subtrees they represent. The product of the numbers along a path gives the number of solutions represented.

How many letter-orderings does a five-letter alphabet have? There are five choices for the first letter, and we have already shown that there are 24 ways to order the remaining four letters. Hence, the answer to the five-letter question is $5 \times 24 = 120$.

Let us generalize this discussion.

Fact 3.2 *Let S be a set of size n. There are*

$$1 \times 2 \times \cdots \times n$$

Copyright © 2008 Steven D. Johnson

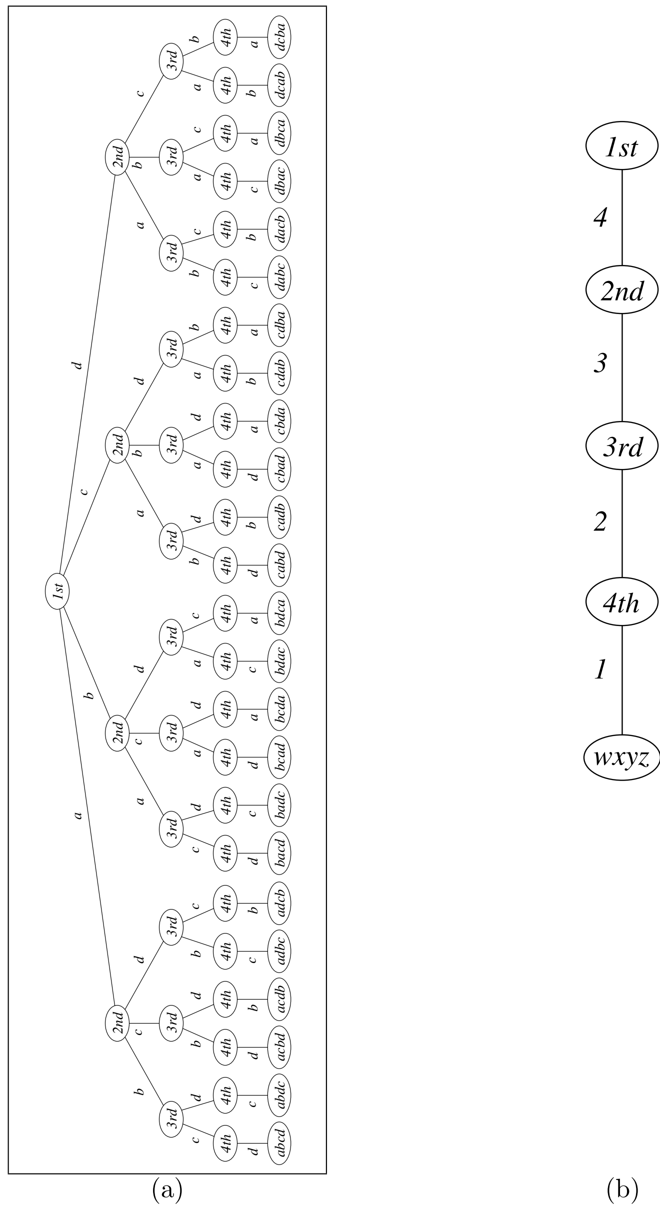


Figure 3.1: A decision tree (a) and counting tree (b) for ordering a 4-letter alphabet

different orders in which the elements of S can be listed. Such a listing, in which each $s \in S$ occurs exactly once, is called permutation of S.

The "product of whole numbers from 1 to n" is useful enough to be given its own notation:

Definition 3.2 *The product*

$$\prod_{k=1}^{n} k = 1 \times 2 \times \cdots \times n$$

is called n factorial*, written $n!$. By convention (Recall Example* **??***), $0!$ is defined to be 1.*

Thus, if $|S| = n$ it has $n!$ permutations.

How many ways are there to list two distinct elements from $A = \{a, b, c, d\}$? Without actually doing it, we could reason as before that

1. There are four possibilities for the first letter.
2. Once the first letter is chosen, three possibilities remain for the second.

So the number of two-letter orderings is $4 \times 3 = 12$. Counting by taking the product of the successive outcomes is called the *Rule of Products* in some textbooks, and the *Principle of Choice* in others.

Another way to look at the problem problem is to start with something you already know—the number of permutations of a set of four elements—and eliminate rudundant representatives. The listing below strikes out all but one of the permuations whose first two letters are the same.

abcd	*bacd*	*cabd*	*dabc*
~~*abdc*~~	~~*badc*~~	~~*cadb*~~	~~*dacb*~~
acbd	*bcad*	*cbad*	*dbac*
~~*acdb*~~	~~*bcda*~~	~~*cbda*~~	~~*dbca*~~
adbc	*bdac*	*cdab*	*dcab*
~~*adcb*~~	~~*bdca*~~	~~*cdba*~~	~~*dcba*~~

So the number of 2-letter permutations is equal to the number of 4-letter permutations divided by the number which represent the same 2-letter outcome. Again, we are taking advantage of a symmetry in the problem, knowing that the partitioning is uniformly independent of what the first two letters actually are. Definition 3.3, below, summarizes this discussion.

Definition 3.3 *Let $\forall$ be a set with $|A| = n$. For $m \leq n$, the number of distinct ways to list m distinct elements from A is*

$$\frac{n!}{(n-m)!}$$

*Such a listing is called an m-*permutation.

Example

Ex 3.1 *You have 26 trophies you would like to display across your fireplace mantel, but there is for only room 10 of them. How many different ways can you do this?*

SOLUTION 1: There are 10 positions at which a trophy can be placed, so the question is simply asking how many 10-permutations are there for a set with 26 elements. By Definition 3.3 this number is

$$\frac{26!}{16!}$$

or $26 \times 25 \times \cdots \times 17$. Unless you have a computer handy and the time, don't bother to calculate this number; it is 19,275,223,968,000.

Suppose you've placed 10 trophies on the mantel, and you decide it is better to place the largest one in the middle. Does swapping two of the selected trophies result in a "different" display? The problem statement fails to make this clear, but the solution given presumes that the order of the trophies matters.

Now suppose we are interested in selecting two letters from $A = \{a, b, c, d\}$ but don't care about the order. In other words we are asking how many distinct *subsets of size 2* are there in a set of four elements. As before, we can consider the set of A's 2-permutations and strike out the redundant ones—those having the same two first letters in either order.

abcd	~~*bacd*~~	~~*cabd*~~	~~*dabc*~~
~~*abdc*~~	~~*badc*~~	~~*cadb*~~	~~*dacb*~~
acbd	*bcad*	~~*cbad*~~	~~*dbac*~~
~~*acdb*~~	~~*bcda*~~	~~*cbda*~~	~~*dbca*~~
adbc	*bdac*	*cdab*	~~*dcab*~~
~~*adcb*~~	~~*bdca*~~	~~*cdba*~~	~~*dcba*~~

Definition 3.4 *The* chose *function*[1] $\binom{n}{k}$ *is the number of ways to choose k unordered elements from a set of size n. This number is given by the formula*

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!}$$

$\binom{n}{k}$ is calculated by dividing the number of permutations of an n-element set by $k!$, the number of ways to permute the first k elements times $(n-k)!$ the number of ways to permute the remaining elements beyond the kth.

[1]Also known as the *binomial coefficient*, and the *combinational number*.

Example

Ex 3.2 *You have 26 trophies you would like to display across your fireplace mantel, but there is for only room 10 of them. How many different ways can you do this?*

This is the same problem as Ex. 3.1, which took the ordering of the ten trophies into account. I the order doesn't matter, we should divide out the number of ways they can be arranged on the mantel:

$$\frac{19,275,223,968,000}{10!} = \frac{19,275,223,968,000}{3,628,800} = 5,311,735 = \frac{26!}{10! \times 16!} = \binom{26}{10}$$

Example

Ex 3.3 *In how many ways can a set six elements, $A = a, b, c, d, ef$, be partitioned into three subsets, X, Y and Z, containing three, two and and elements, respectively?*
Solve this problem by counting the partitions one at a time:

(a) *By Definiton 3.4 there are*

$$\binom{6}{3} = \frac{6!}{3! \times 3!} = \frac{6 \cdot 5 \cdot 4}{3 \cdot 2 \cdot 1} = 20$$

ways to choose three elements from A.

(b) *Once X has been determined, there are three elements left to choose for Y. The number of ways to do that is*

$$\binom{3}{2} = \frac{3!}{2! \times 1!} = 3$$

(c) *Once X and Y have been determined, there is one remaining element to choze for Z*

$$\binom{1}{1} = \frac{1!}{1! \times 0!} = \frac{1}{1 \cdot 1} = 1$$

(d) *The number of partitionings is the product of the numbers of these choices, $20 \cdot 3 \cdot 1 = 60$.*

Remark: It shouldn't matter what order you choose X, Y and Z. Check that

$$\binom{6}{1} \times \binom{5}{2} \times \binom{3}{3} = \binom{6}{2} \times \binom{4}{1} \times \binom{3}{3} = \text{ etc.}$$

End Remark

In Step (b) of Ex. 3.3 calculated that there were $\binom{3}{2}$ ways to choose two elements from 3 for Y. An equivalent problem is to choose one element *not* to include in Y, and there are $\binom{3}{1}$ ways to do that. In general,

Proposition 3.3

$$\binom{n}{k} = \binom{n}{n-k}$$

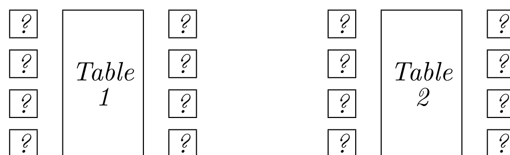
PROOF: By Definition 3.4 and since $n - (n - k) = k$,

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!} = \frac{n!}{(n-k)! \times k!} = \frac{n!}{(n-k)! \times (n-(n-k))!} = \binom{n}{n-k}$$

■

Example

Ex 3.4 *You're having a dinner party for sixteen guests, including yourself, of which half are male and half are female. You have two tables each seating eight, with four places on two sides:*



You want to make sure that both tables have an equal number of guys and gals. Bob is coming, but you don't know about Jane yet. If Jane does come you need to make sure that she and Bob sit at different tables. How many ways are there to do this?

SOLUTION: *There are two cases to consider, according to whether or not Jane comes to the party*

1. *If Jane is not coming, then anyone can sit at either table. You need to choose four guys and four gals to sit at Table 1. The number of ways to do this is*

$$\binom{4}{8} \times \binom{4}{8}$$

   Once the people sitting at Table 1 are determined, all the rest will be assigned to Table 2, and there is only one way to do that.

2. *If Jane comes she must sit at one table and Bob at the other.*

   (a) *Suppose Jane is assigned to Table 1. Then you need to choose:*

i. *3 other gals from the remaining 7 to sit at Table 1. There are* $\binom{7}{3}$ *ways to do that.*

ii. *Four guys from the remaining set of 7—Bob is excluded—will sit at Table 1. There are* $\binom{7}{4}$ *ways to do that.*

iii. *Once the assignments are made to Table 1, all the remaining guests sit at Table 2, so there is only just one choice.*

So there are $\binom{7}{3} \times \binom{7}{4}$ *seating assignments in this case.*

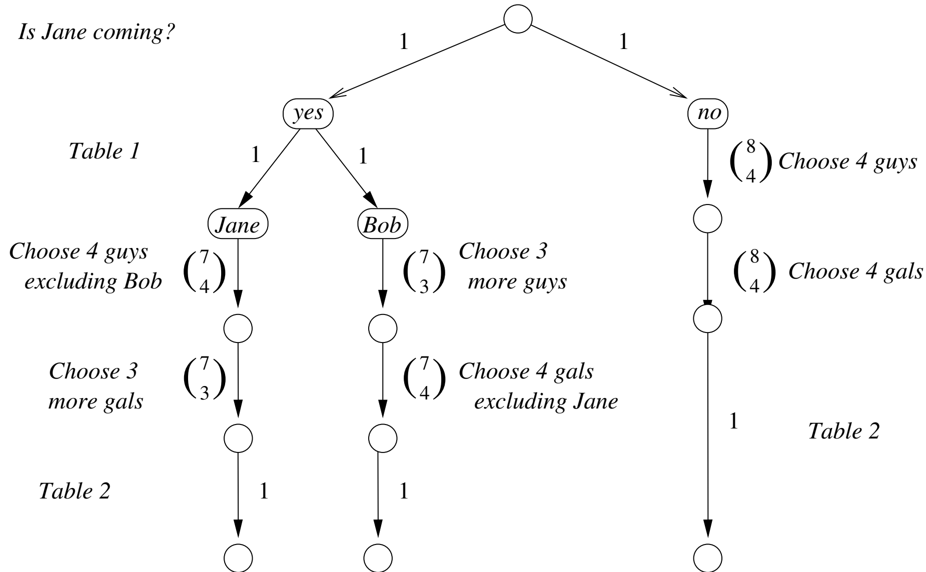
(b) *If Jane is assigned to Table 2, it's the same problem. So the number of assignments in this case is also* $\binom{7}{3} \times \binom{7}{4}$.

The answer, then, is that there are

$$\binom{8}{4} \times \binom{8}{4} + \binom{7}{3} \times \binom{7}{4} + \binom{7}{3} \times \binom{7}{4}$$

ways to assign people to tables.

You may already be thinking that there are many more ways to assign *seating* for this party. Once 8 people are selected for a table, there are 8! ways to arrange them. In the next example we will consider seating assignments. A counting tree for Example 3.4 might look like



The total number of solutions for a problem is the sum, over all the paths *through* (i.e., from the root to a leaf) its counting tree of the product of numbers along each path. More formally,

Fact 3.4 *let $T \subseteq A \times A$ be a decision tree with root r and edge labeling $L\colon T \to \mathbb{N}$. Let $P = \{\langle r, a_1, \ldots, \ell\rangle \mid \ell$ a leaf in $T\}$. The number of solutions represented by T is*

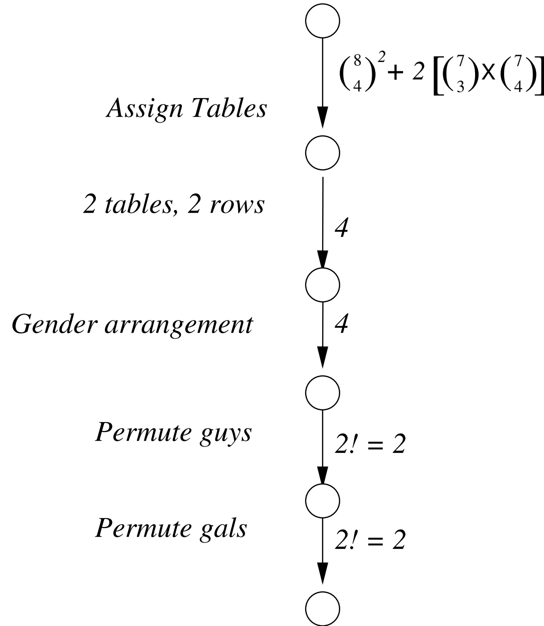
$$\sum_{p \in P} \Big(\prod_{e \in p} L(e) \Big)$$

The formula above takes some liberties with notation. The indexing specifier "$e \in p$" is saying "take all the *edges* in (or along) *path p*." Although in Chapter 6, paths are defined to be sequences, not sets, the meaning is should be clear.

Example

Ex 3.5 *For the same party as Example 3.4, how many ways are there to assign guests to seats in such a way that every guy is sitting next to at least one gal, and* vice versa.

SOLUTION: *Decompose the problem by first assigning guests to tables, as in Example 3.4. Now assign seats along each side of each table. One way to do this is simply to list all the possibilities: guy-gal-guy-gal, gal-guy-gal-guy, guy-gal-gal-guy and gal-guy-guy-gal. So the decision tree looks like*



and since $\binom{7}{3} = \binom{7}{4}$, the number of solutions is

$$64 \left[\binom{8}{4}^2 + 2\binom{7}{4}^2 \right]$$

Exercises 3.2

1. Suppose you want to assign seats for a single row of 4 guys and 4 gals in such a way that each guy is sitting next to *at least* one gal, and *vice versa*. How many ways are there to do this? HINT: Use a decision tree, and practice by solving the 3-guy, 3-gal problem.

2. In Example 3.5, suppose you want to assign seats so that each guy is sitting next to *or across from* at least one gal, and *vice versa*. How many ways are there to do this? HINT: List out all the gender arrangements.

3. A standard deck of cards has 52 cards consisting of 13 cards in each of four *suits*: $\spadesuit$, $\heartsuit$, $\diamondsuit$ $\clubsuit$. In each suit, cards have *face value*s from $\{1, 2, \ldots, 13\}$, each card having a different face value. A *hand* is a set of five cards from the deck. A hand is called a *flush* if all five cards are of the same suit. A hand is called a *straight* if the five cards are sequential in value, for instance, $\{3\heartsuit, 4\spadesuit, 5\diamondsuit, 6\diamondsuit, 7\heartsuit\}$.

(a) How many different flushes are there in a standard deck?

(b) How many different straights are there in a standard deck?

4. *Prove:* For all $n, k \in \mathbb{N}$, $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$

5. If $|A| = n$ guess the value of $|\mathcal{P}(A)|$ by listing a few small examples.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Chapter 4

Induction

The mathematics of programming—indeed, programming itself—is deeply related to inductive reasoning. It is the the most important topic in this book. The word "induction" derives from the notion of learning from seeing, or arriving at a generalization from observation of instances—a fundamental aspect of science, certainly, and perhaps human experience as well. The mathematical notion of induction involves proving some instance (*base cases*) and then showing how to prove the rest (the *induction cases*).

Various styles of inductive reasoning arise in computer science. They are all fundamentally equivalent, so each style can be thought of as an abbreviated form of a basic reasoning principle. In addition, *proof by induction* is just one aspect of a of a collection of schemes for defining sets, functions and relations on these sets, properties of these relations. These are ultimately reflected in the way programs are expressed, starting with recursive programming styles and extending to "loops" in sequential programs.

4.1 Numerical Induction

The *induction principle* for the natural numbers gives a method for proving that a property holds for *all* natural numbers. Let H be any predicate on $\mathbb{N}$ (Recall Definition 6.7). In words, the induction principle says,

> *If you can prove*
>
> (BASE CASE) the assertion $H(0)$ holds, and
>
> (INDUCTION STEP) for an arbitrary $k \in \mathbb{N}$, $H(k)$ implies $H(k+1)$,
>
> *then you may conclude*
>
> by induction, H *holds for all* $n \in \mathbb{N}$.

The *base case* is usually proved by some direct argument about 0. In the *induction step*, $H(k) \Rightarrow H(k+1)$, the antecedent is called the *induction hypothesis*. The argument for the induction case must in no way depend on the choice of k We are actually proving: "For all $k \in \mathbb{N}$, $H(k)$ implies $H(k+1)$.

Example

Ex 4.1 *Prove: For all* $n \in \mathbb{N}$, $\displaystyle\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

PROOF: The proof is by induction on $k \in \mathbb{N}$ with hypothesis

$$H(k) \equiv \sum_{i=0}^{k} i = \frac{k(k+1)}{2}$$

BASE CASE: For $k = 0$,

$$\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$$

This proves the base case.
INDUCTION STEP: Assume that $\sum_{i=0}^{k} i = k(k+1)/2$. We want to show that $\sum_{i=0}^{k+1} i = (k+1)(k+2)/2$. Starting from the left-hand side, we have

$$\begin{aligned}
\sum_{i=0}^{k+1} i &= \left(\sum_{i=0}^{k} i\right) + (k+1) && \text{(expand } \textstyle\sum\text{)} \\
&= \frac{k(k+1)}{2} + (k+1) && \text{(\textit{induction hypothesis})} \\
&= \frac{k(k+1)}{2} + \frac{2k+2}{2} && \text{(multiply by } \tfrac{2}{2}\text{)} \\
&= \frac{k^2+3k+2}{2} && \text{(add fractions, arithmetic)} \\
&= \frac{(k+1)(k+2)}{2} && \text{(factor the numerator)}
\end{aligned}$$

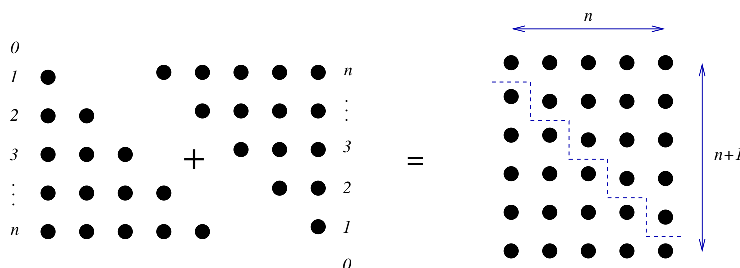
Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

This proves the induction step, and we may now conclude that for all $n \in \mathbb{N}$, $\sum_{i=0}^{n} i = n(n+1)/2$. ∎

REMARK: The sum

$$\sum_{i=0}^{n} i = 0 + 1 + \cdots + n$$

can be thought of as an array of $0, 1, \ldots, n$ dots A diagrammatic argument shows that two such triangles fit together to form an $n \times (n+1)$ rectangle.



The area of the sum is half the area of this rectangle, or $n(n+1)/2$. To most people, this picture is convincing enough to know that the theorem is valid. This diagrammatic proof is often attributed to Eighteenth Century mathematician Karl Friedrich Gauss. The story goes that as a punishment, the teacher assigned Gauss's 3rd grade class to from 1 to 100. Writing this once and then again in reverse:

$$\begin{array}{cccccccc} & 1 & + & 2 & + & \cdots & + & 100 \\ + & 100 & + & 99 & + & \cdots & + & 1 \\ \hline \\ = & 101 & + & 101 & + & \cdots & + & 101 \end{array}$$

Gauss saw that each column has the same partial sum so the final sum is 100×101 divided by 2.

Whether inspired by the diagram or by a manipulation of sums, the intuitive explanation is more compelling than the induction proof. Whether the intuitive explanation is really a proof is debatable, but it can be seen as an application of Theorem 4.2, later in this chapter. END REMARK

The outline of an inductive proof is shown in Figure 4.1. You should include this "boilerplate" when doing the exercises in this section, even though some of it is repetitive. It will help clarify some of the variations on inductive arguments we explore later on.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

THEOREM For all $n \in \mathbb{N}$, $H(n)$.

> [*H is a predicate on natural numbers. Later we do induction on other kinds of sets.*]

PROOF The proof is by induction on $k \in \mathbb{N}$ with hypothesis $H(k)$.

> [*Always announce the induction; it helps orient the reader. The induction hypothesis is the unquantified property $H(k)$. Any variable could be used in place of k—even n—but it is less confusing to introduce a new variable. Always state the variable over which the induction is done and state exactly what $H(k)$ is.*]

BASE CASE

> [*Argue directly for the truth of $H(0)$.*]

This concludes the base case.

INDUCTION STEP Assume $H(k)$.

> [*State the induction hypothesis. The proof of $H(k+1)$ may use this assumption once or several times. Always say exactly where the assumption is used.*]

This concludes the induction. We may conclude that $H(n)$ holds for all $n \in \mathbb{N}$.

Figure 4.1: Outline of an induction proof

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Example

Ex 4.2 *Prove: For given constants, a and $r \neq 1$, and for all natural numbers n,*

$$\sum_{i=0}^{n} ar^i = \frac{ar^{n+1} - a}{r - 1}$$

PROOF: The proof is by induction on k with hypothesis

$$H(k) \equiv \sum_{i=0}^{k} ar^i = \frac{ar^{k+1} - a}{r - 1}$$

BASE CASE: We are to prove that $ar^0 = (ar^{(0+1)} - a)/r - 1$. Reasoning from the right-hand side,

$$\begin{aligned}
&\frac{(ar^{(0+1)} - a)}{r - 1} \\
&= \frac{ar - a}{r - 1} && (r^{(0+1)} = r^1 = r) \\
&= \frac{a(r - 1)}{r - 1} && \text{(distribute } a\text{)} \\
&= a && \text{(cancel } r - 1\text{)} \\
&= a \cdot 1 && \text{(multiply by 1)} \\
&= ar^0 && \text{(substitute, } r^0 = 1\text{)}
\end{aligned}$$

This proves the base case.

INDUCTION STEP: Assume that $\sum_{i=0}^{k} ar^i = (ar^{k+1} - a)/(r - 1)$. To prove $H(k+1)$,

$$\begin{aligned}
\sum_{i=0}^{k+1} ar^i &= \left(\sum_{i=0}^{k} ar^i\right) + ar^{k+1} && \text{(expand } \textstyle\sum\text{)} \\
&= \frac{ar^{k+1} - a}{r - 1} + ar^{k+1} && \textit{induction hypothesis} \\
&= \frac{ar^{k+1} - a}{r - 1} + \frac{ar^{k+1}(r - 1)}{r - 1} && \text{(multiply by } 1 = \tfrac{(r-1)}{(r-1)}\text{)} \\
&= \frac{ar^{k+1} - a + ar^{k+1}(r - 1)}{r - 1} && \text{(add fractions)} \\
&= \frac{ar^{k+1} - a + ar^{k+1}r - ar^{k+1}}{r - 1} && \text{(multiplication)} \\
&= \frac{ar^{k+1}r - a}{r - 1} && \text{(cancel negatives)} \\
&= \frac{ar^{k+2} - a}{r - 1} && \text{(add exponenets)}
\end{aligned}$$

Copyright © 2008 Steven D. Johnson

This completes the induction case. We have shown by induction that for all natural numbers n, $\sum_{i=0}^{n} ar^i = (ar^{n+1} - a)/(r-1)$. ∎

Both the base and induction cases show more details of algebra than would usually be required. From now on, algebraic derivations will be shorter. However, one must *always* show precisely where the induction hypothesis is applied.

Example

Ex 4.3 *Assume that each number M is can be written as a product of prime numbers, $M = q_1 \cdot q_2 \cdots q_k$ and that this decomposition is unique. Prove: There are infinitely many prime numbers.*

PROOF: The proof is by induction on k with hypothesis

$$H(k) \equiv \text{“there are more than } k \text{ prime numbers.”}$$

BASE CASE: $H(0)$ holds because there are more than 0 prime numbers. In particular, 2 is prime. This concludes the base case.
INDUCTION STEP: Assume $H(k)$; that is, assume there are more than k prime numbers. To prove $H(k+1)$, that there are more than $k+1$ prime numbers, we will assume there are *not* and derive a contradiction.

If there are more than k but not more than $k+1$ prime numbers, then there must be exactly $k+1$ primes. Let these numbers be denoted by p_1, p_2, ..., p_{k+1}. Now consider the number

$$N = 1 + p_1 p_2 \cdots p_{k+1}$$

N is not equal to any of the $k+1$ known primes so it cannot be prime. Therefore, N must be evenly divisible by at least one prime number, p_i. But if p_i evenly divides N, since it divides $p_1 p_2 \cdots p_{k+1}$ it must also divide 1. This is impossible and so we have reached a contradiction from the hypothesis that there are just $k+1$ primes. There are more; but this is just what we wanted to prove. This completes the induction step and we may conclude that

$$\text{for all } n \in \mathbb{N}, \text{ there are more than } n \text{ primes.}$$

Another way to state this conclusion is, “there are infinitely many primes.” ∎

Sometimes, inductions start at some number other than zero. This is really just a matter of how the induction hypothesis is stated.

Example

Ex 4.4 *Any amount of postage exceeding 7¢ can be made up using only 3¢ and 5¢ stamps.*

For example, a postage of 29¢ is made up of eight 3¢ stamps and one 5¢ stamp.

Copyright © 2008 Steven D. Johnson

PROOF: The proof is by induction on k with hypothesis

$H(k) \equiv$ *Any postage of up to* $k¢$, $k \geq 8$, *can be made up from* $3¢$ *and* $5¢$ *stamps.*

BASE CASE: One 5¢ and one 3¢ stamps make 8¢ postage.
INDUCTION STEP: Assume $H(k)$ and consider making $(k+1)¢$ postage. There are two cases to consider, depending on whether a 5¢ stamp is used to make $k¢$ postage.

> CASE I If a 5¢ stamp is used to make $k¢$ postage then replacing that stamp with two 3¢ stamps makes $(k+1)¢$ postage.
>
> CASE II If $k¢$ postage is made up of 3¢ stamps only, then there must be at least three 3¢ stamps because $k \geq 8$. Replacing three of the 3¢ stamps with two 5¢ stamps then makes $(k+1)¢$ postage.

The two cases complete the induction case and the proof of the claim. ■

Had the induction hypothesis been, "Postage of $(k+8)¢$ can be made up from 3¢ and 5¢ stamps," then the base case would have been for $k = 0$, but the algebra would have been more complicated. Starting at some number $n > 0$ is called a *base translation.*

Sometimes, it is convenient to use a form of inductive argument in which the hypothesis is assumed to hold for all numbers *up to* k. In words,

> *If you can prove*
>
> (BASE CASE) the assertion $H(0)$ holds, and
>
> (INDUCTION STEP) whenever $H(0)$, $H(1)$, ..., $H(k-1)$ all hold, so does $H(k)$,
>
> *then you may conclude*
>
> by induction, H *holds for all* $n \in \mathbb{N}$.

This principle is equivalent to the first principle because the induction hypothesis is still a proposition about a natural number k. See Exercise 7.

Example

Ex 4.5 *Every natural number greater than 2 is uniquely decomposable into a product of prime numbers.*

Copyright © 2008 Steven D. Johnson

PROOF: The proof is by (course-of-values) induction on k with hypothesis

$H(k) \equiv$ all numbers less than k are uniquely decomposable into a product of primes.

BASE CASE: The base case, for $k = 2$, holds because 2 is the only prime divisor of 2.
INDUCTION STEP: Assume that every number less than k is uniquely decomposable, and suppose that k can be decomposed in two ways,

$$k = p_1 p_2 \cdots p_n$$

and

$$k = q_1 q_2 \cdots q_m$$

such that each of the p's and q's is prime. Since k is decomposable, it can't itself, be prime, so we also know that $k = st$ for two smaller numbers s and t. Without loss of generality, we can assume that the p's and q's are arranged so that

$$k = st = (p_1 \cdots p_i)(p_{i+1} \cdots p_n)$$

and

$$k = st = (q_1 \cdots q_j)(q_{j+1} \cdots q_m)$$

for indices i and j, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Since s and t are both less than k, by the induction hypothesis each has a unique decomposition. This implies that $i = j$, $n = m$, and for $1 \leq \ell \leq n$, $p_\ell = q_\ell$. In other words, k's decomposition is unique. This completes the induction case and we may conclude that all natural numbers are uniquely decomposable into primes.

∎

REMARK: In the induction step, the phrase "without loss of generality" acknowledges a specific assumption made in order to simplify the argument. In this case, it is assumed that the p's and the q's happen to be arranged in an order that exposes the decompositions of s and t. There is no loss of generality because multiplication is commutative—we are free to rearrange the p's and the q's as we want. However, to do so would make the argument much longer without adding any insight. END REMARK

Exercises 4.1

1. Prove that for all natural numbers, n,

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

2. Prove that for all natural numbers, n,

$$\sum_{i=0}^{n} i(i!) = (n+1)! - 1$$

3. Prove that for all natural numbers n, 6 evenly divides $n^3 - n$.

4. Prove that for all natural numbers $n > 4$, $2^n > n^2$.

5. Prove that for all natural numbers n, $\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2$.

6. Recall that the *choose* function, defined

$$\binom{s}{r} = \frac{s!}{r!\,(s-r)!}$$

is the number of different ways to choose r objects from a set of s objects. Prove that for all $n \in \mathbb{N}$,

$$\sum_{i=0}^{n-1} (2i)^2 = \binom{2n}{3}$$

7. Let H be a predicate on $\mathbb{N}$. Explain how the proposition *for all* $n \in \mathbb{N}$ *such that* $n > M$, $H(n)$ can be transformed to an equivalent proposition, *for all* $n \in \mathbb{N}$, $H'(n)$, for some predicate H' on $\mathbb{N}$. Perform this transformation on the proposition in Exercise 4

8. Let H be a predicate on $\mathbb{N}$. Explain how the proposition *for all* $n \in \mathbb{N}$ *such that* $n > M$, $H(n)$ can be transformed to an equivalent proposition, *for all* $n \in \mathbb{N}$, $H'(n)$, for some predicate H' on $\mathbb{N}$. Perform this transformation on the proposition in Exercise 4

9. Repeat the proof of Exercise 4.5 using the first Principle of Induction, with the induction hypothesis:

$H(k) \equiv$ For all $j \leq k$, j is uniquely decomposable into a product of primes

4.2 More Examples of Induction

Although inductive proofs can always be reduced to arguments about a natural number, induction is not used only to prove facts about arithmetic. In this section, we look at a few non-arithmetic results. The first has to do with geometry.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

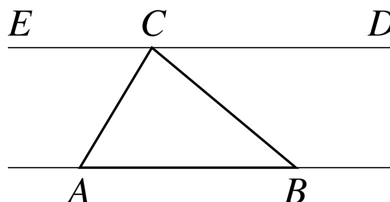
Example

Ex 4.6 *A polygon is* convex *if every line joining the points of the polygon lies within the polygon. Equivalently, all vertices of a convex polygon "point out." Prove that the sum of the interior angles of a n-sided convex polygon, $n \geq 3$, is equal to* $(n-2) \cdot 180°$.

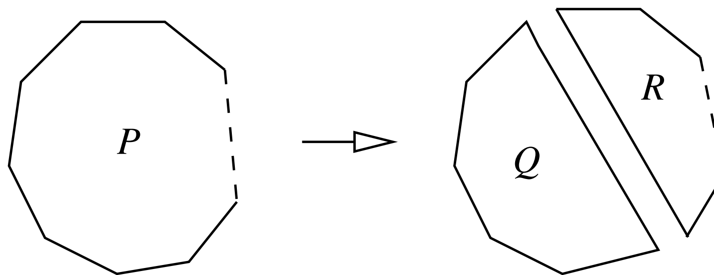
PROOF: The proof is by induction, with predicate

$H(k) \equiv$ The sum of the interior angles of a convex polygon with at most k sides is $(k-2) \cdot 180°$.

BASE CASE: In the base case, $H(3)$ asserts that the sum of the angles of a triangle is $(3-2) \cdot 180° = 180°$. We know from elementary geometry that this is the case. To prove it, place the triangle between two parallel lines as shown below. Since lines ED and AB are parallel, angles $ACE = CAB$ and $DCB = CBA$. Furthermore, since ECD is a straight line, $ECA + ACB + DCB = 180°$.



(Check that the argument doesn't depend on ABC being an acute triangle!)
INDUCTION STEP: Assume $H(k)$. Now consider any polygon P with $k+1$ sides. Since P has at least 4 sides, it is possible to divide it into two convex polygons by drawing an interior line between two of its vertices:



Call these two polygons Q and R. Now, both Q and R are convex and have fewer sides than P. In particular, each has *at most* k sides. Suppose Q has $l_Q \leq k$ sides and R has $l_R \leq k$ sides. Together, Q and R have two more sides than P; that is,

$$l_Q + l_R - 2 = k + 1 \tag{4.1}$$

In addition, since P is formed by putting Q and R side-by-side: The sum of P's interior angles is equal to the sum of the interior angles of Q and R. By the

Copyright © 2008 Steven D. Johnson

induction hypothesis, the sum of the interior angles of Q is $(l_Q - 2) \cdot 180°$. By the induction hypothesis the sum of the interior angles of R is $(l_R - 2) \cdot 180°$. Therefore, the sum of P's interior angles is

$$
\begin{aligned}
&((l_Q - 2) + (l_R - 2)) \cdot 180° \\
&= ((l_Q + l_R - 2) - 2) \cdot 180° && \text{(rearrange sums)} \\
&= ((k + 1) - 2) \cdot 180° && \text{(by Equation 4.1)}
\end{aligned}
$$

This completes the induction and proves that the sum of the interior angles of any convex polygon with n sides is $(n - 2) \cdot 180°$. ∎

Discussion: There are three points of interest about Example 4.6. First, the statement has the form: *For all* $n \geq 3$, $H(n)$, so the base case is $k = 3$. The hypothesis could be changed to make 0 the base case:

$H'(k) \equiv$ The sum of the interior angles of a convex polygon with at most $k + 3$ sides is $(k + 1) \cdot 180°$.

Second, had the proof been declared to be course-of-values induction, the hypothesis could have been written:

$H'(k) \equiv$ The sum of the interior angles of a convex polygon with k sides is $(k - 2) \cdot 180°$.

The missing phrase "at most" becomes part of the proof scheme. (In fact, this proposition can just as easily be proved using the first induction principle with the above hypothesis. See Exercise 3 at the end of this section.)

Third, notice that in this proof the induction hypothesis is used *twice* to reason about polygons Q and R.

In the next example, we prove a distributive law for the operations of intersection and union on sets. A number of similar laws are proven as exercises.

Example

Ex 4.7 *Let* $B, A_1, A_2, \ldots, A_n$, $n \geq 1$, *be any collection of sets. Then*

$$H(n) \;\equiv\; B \cap \bigcup_{i=1}^{n} A_i = \bigcup_{i=1}^{n} B \cap A_i$$

PROOF: The proof is by induction on k with hypothesis $H(k)$.
BASE CASE: If there is just one set, A_1, then the extended union degenerates:

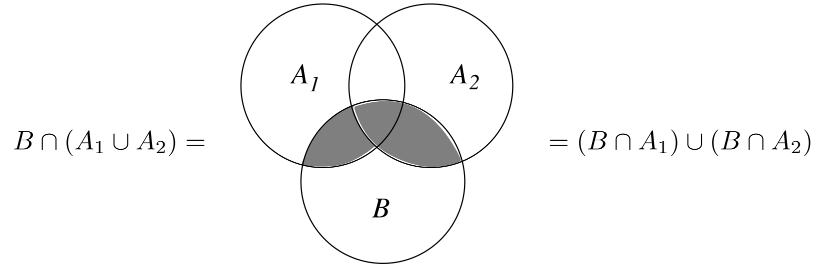
$$B \cap \bigcup_{i=1}^{1} A_i = B \cap A_1 = \bigcup_{i=1}^{1} (B \cap A_i)$$

Although this proves the base case, it is still it is worthwhile looking at the case for two sets. We can prove

LEMMA

$$B \cap (A_1 \cup A_2) = (B \cap A_1) \cup (B \cap A_2)$$

PROOF: This Venn diagram:



∎

The same argument as in the base case is used for the induction step:
INDUCTION STEP: Assume $H(k)$ holds. For a collection of sets $A_1, A_2, \ldots, A_{k+1}$, consider

$$B \cap \bigcup_{i=1}^{k+1} A_i$$

The union of the A_i can be expressed as the union of two sets,

$$= B \cap \left(\bigcup_{i=1}^{k} A_i \cup A_{k+1} \right)$$

By the argument just given intersection can be written,

$$= \left(B \cap \bigcup_{i=1}^{k} A_i \right) \cup (B \cap A_{k+1})$$

According to the induction hypothesis, intersection with B distributes over the extended union:

$$= \bigcup_{i=1}^{k} (B \cap A_i) \cup (B \cap A_{k+1})$$

Extending the limits of the extended union, we get

$$= \bigcup_{i=1}^{k+1} (B \cap A_i)$$

this completes the induction step and the proof of the claim. ∎

With the next two examples, we embark on a study of rigorous reasoning about programs. Our first step in the next example is a small one, but we immediately generalize to an important fundamental principle.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Consider the following program, $\mathcal{P}$, in which program variables w, x, y and z range over integers.

```
P: begin
   {A > 0}
   x := A;
   y := B;
   z := 0;
ℓ: while x ≠ 0 do
     begin
     x := x − 1;
     z := z + y
     end;
   w := z;
   end {w = AB}
```

The comments indicate that if A is nonnegative, this program computes the product AB, leaving the result in w. An intuitive explanation is that the `while` statement ℓ "loops" just x times, adding y to z each time through. Of course, this only works if x is a nonnegative value. An induction is needed to prove this claim:

Proposition 4.1 *If program $\mathcal{P}$, shown above, ever reaches the loop ℓ with $x \in \mathbb{N}$, then $\mathcal{P}$ halts with $w = z + xy$.*

Before doing the proof, observe that if this proposition holds, then we can conclude that $\mathcal{P}$ computes AB because after the initial assignments, program variables x, y, and z hold values A, B, and 0, respectively, so that

$$w = 0 + AB = AB$$

REMARK: Note also that in making a formal statement of this property, there is a confusion between the *name* of a program variable and the *value* it contains. For example, the equation "$w = z + xy$" refers to the value in w *after* the loop executes and the values of x, y, and z *before* the loop starts executing. It is this confusion that makes programs hard to talk about; it is caused by the lack—at this stage of the book—of a good mathematical model of how programs work.
END REMARK

PROOF: The proof is by induction on natural number k with hypothesis

$H(k) \equiv$ If $\mathcal{P}$ ever reaches the loop ℓ with $x = k$, then $\mathcal{P}$ halts with $w = z + xy$.

BASE CASE: Should $\mathcal{P}$ reach ℓ with $x = 0$ then the test "$x \neq 0$" fails; so the program leaves the loop and sets w to $z = z + 0y = z + xy$. This concludes the base case.

INDUCTION STEP: Assume $H(k)$ and now suppose that $\mathcal{P}$ reaches ℓ with $x = k+1$. Then the test "$x \neq 0$" succeeds; so $\mathcal{P}$ executes the body of the loop and returns to ℓ with new values in the three program variables, given by:

$$\begin{aligned} x' &= x - 1 = k \\ y' &= y \qquad (y \text{ is unchanged}) \\ z' &= z + y \end{aligned}$$

But since program variable x now contains k (again, note the name/value confusion) the induction hypothesis applies. Hence, $\mathcal{P}$ halts with $w = z' + x'y'$. That is

$$\begin{aligned} w &= z' + x'y' \\ &= z + y + (x-1)y \\ &= z + y + xy - y \\ &= x + xy \end{aligned}$$

This concludes the induction step and we may now conclude:

> If $\mathcal{P}$ ever reaches the loop ℓ with $x \in \mathbb{N}$, then $\mathcal{P}$ halts with $w = z + xy$.

as needed. ∎

The proof of the previous example hinged on the discovery of a property that held *each time the program returned to the beginning of its loop.* Such a property is called a loop *invariant*; it captures the "essence" of what the loop does. We can generalize the preceding argument to talk about certain kinds of loops, and this gives us a new principle of induction to use in reasoning about programs.

Theorem 4.2 *[Theorem on Loop Invariants] Let B be any test, S any statement, and consider the program fragment*

```
      ⋮
ℓ:  while B do S;
ℓ′:
      ⋮
```

Suppose I is a property such that

> $G \equiv$ *whenever both I and B hold before S executes, I holds again after S executes.*

Then should $\mathcal{P}$ ever reach ℓ with I true, and if $\mathcal{P}$ then reaches the point ℓ', I will be true and B will be false.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

PROOF: The proof is by induction, of course. It uses the fact that if $\mathcal{P}$ reaches the point ℓ' it must do so by executing the body S some finite, integral number of times. Hence, the induction hypothesis is

$H(k) \equiv$ Should $\mathcal{P}$ ever reach ℓ with I true, and if $\mathcal{P}$ then reaches the point ℓ' after going through the loop exactly k times, I will be true and B will be false.

Completing the proof is a simple exercise. ∎

Example

Ex 4.8 The Theorem on Loop Invariants allows us to distill the proof of Proposition 4.1 to its essential details, once we have discovered the invariant property:

$$AB = z + xy$$

The theorem does the induction "once and for all."

(a) By looking at the initial assignments we know that program $\mathcal{P}$ reaches the loop with
$$z + xy = 0 + AB = AB$$

(b) In the proof of Proposition 4.1 we showed that whenever $\mathcal{P}$ reaches its loop with $AB = z + xy$, this equation will still hold after the loop's body executes.

Therefore, by the Theorem on Loop Invariants, when the program leaves its loop, we will have both $AB = z + xy$ and $x = 0$, hence $z = AB$.

Exercises 4.2

1. Prove that for all natural numbers $n \geq 1$, if $B, A_1, \ldots, A_n$ are sets, then

$$B \cup \bigcap_{i=1}^{n} A_i = \bigcap_{i=1}^{n} (B \cup A_i)$$

2. Prove that for all natural numbers $n \geq 1$, if $B, A_1, \ldots, A_n$ are sets, then

$$B \cup \bigcap_{i=1}^{n} A_i = \bigcap_{i=1}^{n} (B \cup A_i)$$

3. For $n \in \mathbb{N}$, let Q, P_1, $P_2, \ldots, P_n$ be propositions. Prove the following:

(a) $Q \wedge \bigvee_{i=1}^{n} P_i = \bigvee_{i=1}^{n} (Q \wedge P_i)$

(b) $\neg \bigwedge_{i=1}^{n} P_i = \bigvee_{i=1}^{n} (\neg P_i)$

(c) $Q \vee \bigwedge_{i=1}^{n} P_i = \bigwedge_{i=1}^{n} (Q \vee P_i)$

(d) $\neg \bigvee_{i=1}^{n} P_i = \bigwedge_{i=1}^{n} (\neg P_i)$

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

4. Prove Example 4.6 using the First Principle of Induction, with induction hypothesis $H(k) \equiv$ "The sum of the interior angles of any convex polygon with *exactly* k sides, $k > 3$, is $(k-2) \cdot 180°$."

5. Recall that a *tree* is a finite acyclic graph with a single root node of in-degree 0 and all of whose nodes except the root have in-degree 1. Prove that the number of nodes in a binary tree is exactly one greater than its number of edges.

6. Recall (Exercise 2) that a *binary tree* is a tree whose interior nodes all have out-degree 2. Prove that the number of leaves of a binary tree is exactly one greater than the number of interior nodes.

7. Complete the proof of Theorem 4.2.

8. Consider the program

```
P:  begin
    {A, B > 0}
    x := A;
    y := B;
ℓ:  while x ≠ 0 do
       begin
       x := x − 1;
       y := y + 1
       end;
    end {y = A + B}
```

Use Theorem 4.2 and invariant assertion

$$I \equiv x + y = A + B$$

to prove that this program computes $A + B$.

9. Consider the program

```
P:  begin
    {A, B > 0}
    q := 0;
    r := A;
ℓ:  while r ≥ B do
       begin
       q := q + 1;
       r := r − B
       end;
    end {A = qB + r ∧ r < B}
```

Use Theorem 4.2 and invariant assertion

$$I \equiv A = qB + r$$

to prove that this program computes the quotient and remainder of A and B.

Copyright © 2008 Steven D. Johnson

Chapter 5

Countability and Order

5.1 Cardinality and Countability

For infinite sets, it makes little sense to say $|S|$ is a *number* or that S has a "size". Nevertheless, we are interested in *comparing* infinite sets, so the concept of a set's size must be made more general.

Definition 5.1 *Two sets, S and T, said to be of the* same cardinality *if one can exhibit a one-to-one correspondence between them, that is, if a bijection $f\colon A \leftrightarrow B$ exists.*

In the case of two finite sets, we need only show that $|A| = |B|$. It should also make sense—although we do not yet have the technical means to prove it—that if $|A| \neq |B|$, there can be no bijection between them: no mapping from the larger to the smaller set can be injective. In other words, any function from the larger set to the smaller must map two or more elements to the same target.

Fact 5.1 (Pigeon-Hole Principle) *If $|A| > |B|$ and $g\colon A \to B$, then there exists at least one $b \in B$ for which $|g^{-1}\{b\}| > 1$.*

The Pigeon-Hole Principle often arises in proofs-by-counter-example, as for example, in showing that a path in $R \subseteq A \times A$ of length $n > |A|$ must contain a cycle.

In the case of infinite sets, we are particularly interested in those having the same cardinality as $\mathbb{W}$. In essence, these are sets that can be indexed by whole numbers, and so can be can be listed "in order,"

$$S = \{s_1, s_2, s_3, \ldots, s_i, \ldots\}$$

For most purposes, it is enough to know that S is "small enough" to be listed in a linear order, even if some of the indices are missing. So we don't need a bijection; and injection is good enough.

Definition 5.2 *A set S is* countable *if there exists an injection, $C\colon S \to \mathbb{W}$.*

Proposition 5.2 $\mathbb{N}$ *is countable.*

PROOF: The mapping $C: n \to n+1$ gives a bijection, hence also an injection from $\mathbb{N}$ to $\mathbb{W}$. ∎

Proposition 5.3 *The integers, $\mathbb{Z}$, are countable.*

PROOF: The proof is done by exhibiting a function that satisfies Definition 5.2. Define $C: \mathbb{Z} \to \mathbb{N}$ according to:

$$C(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ -(2x+1) & \text{if } x < 0 \end{cases}$$

C maps positive integers to even numbers and negative integers to odd numbers. Hence, it is injective; different numbers never get the same index. ∎

The next proposition allows us to use any countable set to index another countable set. In other words, we don't have to be overly specific about how the elements are ordered, because in most cases we are more interested in cardinality, not the exact correspondence.

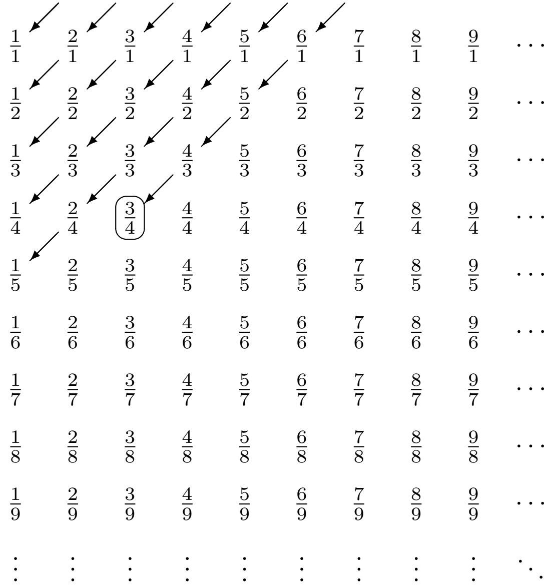
Proposition 5.4 *A set S is countable if there exists an injection from S to a countable set, A.*

PROOF: If $f: S \to A$ and $C: A \to \mathbb{W}$ are both injections, then so is $f \circ C: S \to \mathbb{W}$. ∎

Theorem 5.5 *The rational numbers, $\mathbb{Q}$, are countable.*

PROOF: Recall that $\mathbb{Q} = \{\frac{n}{m} \mid n, m \in \mathbb{Z} \text{ and } m \neq 0\}$. Looking at Propositions 5.2, 5.3 and 5.4 it suffices to restrict to positive rationals, $\mathbb{Q}^+ = \{\frac{n}{m} \mid n, m \in \mathbb{W}\}$. We are actually showing that $\mathbb{W} \times \mathbb{W}$ is a countable set. Consider this set layed out as an infinite two-dimensional array, below. We will define the required bijection C so that it orders this array along successive diagonals.

Copyright © 2008 Steven D. Johnson



To calculate $C(\frac{x}{y})$ consider:

(a) $x + y$ is two greater than the length of the diagonal just above it, and all the rationals on that diagonal and above are counted. The number of these points is the sum of the lengths of the preceding diagonals (Recall Example 4.1),

$$\sum_{i=1}^{x+y-2} i = \frac{(x+y-1)\cdot(x+y-2)}{2}$$

(b) The number of points along the diagonal leading to $\frac{x}{y}$ is the same as the number of points directly above $\frac{x}{y}$, which is $y - 1$. See Fig. 5.1

Adding these together, we get

$$\begin{aligned} C(\tfrac{x}{y}) &= \tfrac{((x+y)-1)((x+y)-2)}{2} + y - 1 \\ &= \tfrac{(x+y)^2-3(x+y)+2}{2} + \tfrac{2y-2}{2} \\ &= \tfrac{(x+y)^2-3x-3y+2+2y-2}{2} \\ &= \tfrac{(x+y)^2-3x-y}{2} \end{aligned}$$

REMARK: The display leaves out rationals of the form $\frac{0}{y}$, which could have been included by adding another row along the top and adjusting the formula for C. This is left as an Exercise.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

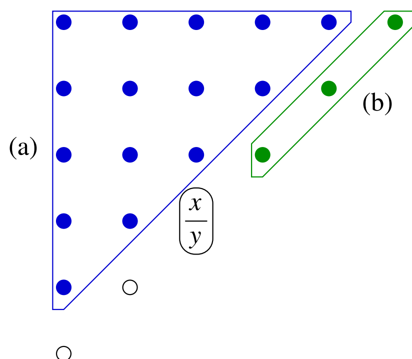


Figure 5.1: Calculation of $C(\frac{x}{y})$

You might be concerned about the fact that C maps "equivalent" rationals to different values. For instance $C(\frac{1}{1}) = 1$, $C(\frac{2}{2}) = 5$, $C(\frac{3}{3}) = 13$, $C(\frac{4}{4}) = 25$, and so forth, but all these fractions reduce to the number 1. However, in this theorem we are not counting equivalence classes, but numeric representations according to the definition of $\mathbb{Q}$. Even if we were to consider equivalent values to represent the same number, countability just says that there are *no more* elements of a set S than there are whole numbers, so it is all right to count something more than once, as long as everything is counted at least once.

When one thinks of *rational numbers*, one ordinarily thinks of the real numbers they represent and also includes negative values. These could be incorporated in C, for instance, by mapping positive rationals to even numbers and negative rationals to odd numbers, as in Proposition 5.3. END REMARK

The next theorem illustrates an important proof technique called *diagonalization*, used frequently in theoretical computer science.

Theorem 5.6 $\mathbb{R}$ *is not countable.*

PROOF: This is a *proof by contradiction*, in which we assume the result is true and then deduce that it cannot hold. It depends on the fact that any real number can be expressed as a possibly infinite decimal expansion, and conversely, that any decimal numeral represents some real number. For the sake of simplicity we will consider only the numbers in the interval $[0, 1)$. If we can't count these, we certainly can't count the entire set $\mathbb{R}$.

Suppose that we can count the numbers in this interval. Then there must be a way to list them in order, $\{r_1, r_2, r_3, \ldots\}$. Consider the decimal expansions

Copyright © 2008 Steven D. Johnson

of the r_i:

$$
\begin{array}{ccccccccccc}
r_1 & = & 0 & . & \boxed{d_{11}} & d_{12} & d_{13} & d_{14} & d_{15} & d_{16} & d_{17} \; d_{18}\ldots \\
r_2 & = & 0 & . & d_{21} & \boxed{d_{22}} & d_{23} & d_{24} & d_{25} & d_{26} & d_{27} \; d_{28}\ldots \\
r_3 & = & 0 & . & d_{31} & d_{32} & \boxed{d_{33}} & d_{34} & d_{35} & d_{36} & d_{37} \; d_{38}\ldots \\
r_4 & = & 0 & . & d_{41} & d_{42} & d_{43} & \boxed{d_{44}} & d_{45} & d_{46} & d_{47} \; d_{48}\ldots \\
r_5 & = & 0 & . & d_{51} & d_{52} & d_{53} & d_{54} & \boxed{d_{55}} & d_{56} & d_{57} \; d_{58}\ldots \\
r_6 & = & 0 & . & d_{61} & d_{62} & d_{63} & d_{64} & d_{65} & \boxed{d_{66}} & d_{67} \; d_{68}\ldots \\
 & & & \vdots & & & & & & & \ddots
\end{array}
$$

where each $d_{ij} \in \{0,1,2,3,4,5,6,7,8,9\}$ Now consider the numeral

$$0 \; . \; d'_{11} \; d'_{22} \; d'_{33} \; d'_{44} \; d'_{55} \; d'_{66} \; \ldots \; d'_{kk} \; \ldots$$

In which each digit d'_{kk} is chosen to be *different* than d_{kk} from the list of r_is. The number this numeral represents cannot be in the list because for all $n \in \mathbb{W}$ it differs at the nth decimal place from the nth numeral in the list (hence the term *diagonalization*). This contradicts the assumption that all numbers are listed, so that assumption must be false; and this completes the proof of the theorem. ∎

REMARK: Although it is essentially correct, this argument is subtly flawed, because the same number can have two different decimal representations. For instance, $0.0999\ldots = 0.1$ which follows from the fact that

$$(10x - x) \div 9 = \frac{9}{9}x = x$$

The proof can be repaired by excluding such redundant decimal expansions from the list, but the details are not illuminating.

END REMARK

Exercises 5.1

1. Modify the proof of Proposition 5.5 to include rationals of the form $\frac{0}{y}$ for $y \in \mathbb{W}$.

2. Prove: *If sets A and B are countable, then so are $A \cup B$, $A \times B$ and $A \setminus B$.*

3. Suppose sets A and B are countable. Explain why $\mathcal{P}(S)$ and $f : A \to B$ are not necessarily countable.

5.2 Order Notation and Order Arithmetic

Functions from $\mathbb{N}$ to $\mathbb{R}$ are used to estimate program running times, memory requirements, file sizes, and other resources that may vary depending on some

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

program parameter. The rate at which these resources grow as the problem size gets larger is one way to characterize an algorithm for solving the problem. Often, this rate cannot be precisely determined, of if it can, the formula is too complicated to work with.

Figure 5.2 shows the measured running times of a program and three functions approximating its behavior. The dotted line represents an estimated statistical average, obtained by finding a "smooth" curve of minimal distance from the measured times. Upper line is a curve that lies close to but above all measured times; it represents an *upper bound* estimate of performance. Similarly, the curve lying below the points is a *lower bound*. The area between the upper and lower bounds is called the performance *envelope* of the program.

These approximations are all parabolas, given by formulas of the form

$$f(n) = an^2 + bn + c$$

for various coefficients a, b and c. So the program's running time can be characterized as quadratic (varying with the square of input n. When comparing f with the envelope for some other program, the first concern is whether their growth rates are related; do they both grow quadratically, or does is one significantly different. In making this comparison, we are interested their behavior as n grows ever larger, and we are not interested in constant ratios. These ideas are captured in the next definition, in which $\mathbb{R}^+$ stands for the set of postitive real numbers, $\{x \in \mathbb{R} | x > 0\}$.

Definition 5.3 *Let $f, g\colon \mathbb{N} \to \mathbb{R}^+$. We say that f* is of order g*, written $f \in O(g)$, if there exist $N \in \mathbb{N}$ and $C \in \mathbb{R}$ such that for all $n \leq N$, $f(n) \leq C \cdot g(n)$.*

The N and C Definition 5.3 are sometimes called *witnesses*; N may be referred to as a *threshold*; and C as a *proportionality constant*.

Example

Ex 5.1 *Let $f(n) = 2n^2 + 5n + 3$ and $e(n) = n^2$. Show that $f \in O(e)$.*

SOLUTION: *We need to find a* constant factor*, C and a* threshold *value N at which*

$$2N^2 + 5N + 3 \leq cN^2$$

Consider each term in $f(n)$ to find a dominating value in terms of n^2:

(a) *If $5 \leq m$ then $5m \leq m^2$.*

(b) *If $2 \leq m$ then $3 \leq m^2$.*

(c) *To satisfy both (a) and (b) take N to be 5. Then for any $n \geq N$*

$$f(n) = 2n^2 + 5n + 3 \leq 2n^2 + n^2 + n^2 = 4n^2$$

Thus $f \in O(n^2)$ with witnesses $N = 5$ and $C = 4$.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

"Big Oh" relationships may be expressed without giving names to the functions, by just using their defining expressions. So Example 5.1 might have been written,

$$\text{Show that } 2n^2 + 5n + 3 \in O(n^2)$$

The propositions and the exercises that follow state some basic properties about O.

Proposition 5.7 *If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.*

PROOF: Exercise 3 ∎

Proposition 5.8 *If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1(n) + f_2(n) \in O\big(g_1(n) + g_2(n)\big)$.*

PROOF: Exercise 6 ∎

Proposition 5.9 *If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1(n) \times f_2(n) \in O\big(g_1(n) \times g_2(n)\big)$.*

An example of a property for which there is no simple relationship is function composition. If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ we would wish for simple order relationship between $f_1 \circ f_2$ and $O(g_1 \circ g_2$. Unfortunately, there is no simple formula for this relationship.

PROOF: Exercise 7 ∎

Exercises 5.2

1. For positive real coefficients $a_0, a_1, \ldots, a_n$, let $f \colon \mathbb{N} \to \mathbb{R}$ be a polynomial function,

$$f(x) = a_n x^n + \cdots + a_1 x^1 + a_0 x^0$$

For example, one such function is

$$F(x) = 3x^4 + 2.0x^3 + 3.3x^2 + 9x + 1$$

in which the coefficients are $a_0 = 1$, $a_1 = 9$, $a_2 = 3.3$, $a_3 = 2.0$, and $a_4 = 3$.

Show, in general, that $f \in O(x^n)$.

2. Define two functions, f and g, for which $f \notin O(g)$ and $g \notin O(f)$.

3. Prove Proposition 5.7: If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1(n) + f_2(n) \in O\big(g_1(n) + g_2(n)\big)$.

4. Prove that for any base b, $\log_b(x) \in O(x)$.

5. Prove that for any bases b and b', $\log_b(x) \in O(\log_{b'}(x))$.

Copyright © 2008 Steven D. Johnson

6. Prove Proposition 5.8

7. Prove Proposition 5.9: *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then*

$$f_1(n) \times f_2(n) \in O\big(g_1(n) \times g_2(n)\big)$$

.

8. It is a fact that $n^x \in O(2^n)$. Explain or prove this result. HINT: A complete proof requires some familiarity with ideas from The Calculus; specifically, that for strictly increasing functions f and g over the interval $[a, b]$, if $f(a) < g(a)$ and $f(b) < g(b)$ then $f(x) < g(x)$ for all $a \leq x \leq b$.

9. Is $2^n \in O(n!)$ or is $n! \in O(2^n)$?

5.3 Complexity

Note: This section is incomplete In this chapter, we have looked at the concept of "size," first extended to non-finite sets, and later to order relationships between functions. Two results are worth special attention:

(a) Theorem 5.6 shows that there are different orders of infinity (or infinite magnitude).

(b) Exercises 8 and 1, together, say that the *exponential* function, $e(n) = e^n$ dominates every *polynomial*, $p(n) = a_0 n^0 + a_1 n^1 + \cdots + a_m n^m$

These results hint at a stratification of order (or cardinality) that turns out to be relevant to computational models.

5.3.1 The Halting Problem

An important question to ask about computing is whether a program can, or cannot, be written to solve a kind of problem. If a program can be written to solve all instances of a problem class, that problem is said to be *decidable*. To say that a problem is *undecidable* means that any program written to solve the problem class will fail on some, but not necessarily all, problem instances. "Failure" does not say that the program will get the wrong answer, but that it will fail to find any answer at all; that is, that the program may fail to terminate.

Theorem 5.10 *(The Halting Problem) It is undecidable whether a program p terminates on input i.*

PROOF OUTLINE. Here, the problem class is to determine whether a given program terminates or not. So a program to solve this problem must take a program as one of its inputs. Suppose PL is a programming language. A more detailed statement of this theorem would be:

There is no program H satisfying the property that, For all programs $p \in PL$ and inputs a, $H(p, a)$ returns *true* iff p terminates on input i.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Proof Sketch: The proof is by a *diagonalization*. Suppose there is a program $\mathsf{HALT}\colon PL \times INPUT \to \{true, false\}$ that satisfies the property, "$\mathsf{HALT}(P, I)$ returns *true* if and only if program P terminates when given input I." Using HALT construct a new program,

```
HALT′ ≡ begin
          ┌──────────────────────────────────┐
          │ code of program HALT, leaving    │
          │ the result in variable halt;     │
          └──────────────────────────────────┘
          if halt = true
            then while true do nothing
            else return(true)
        end
```

Given any program P and input I, HALT′ returns *true* whenever the original HALT would return *false* and loops forever, not returning anything, whenever HALT would return *true*. Now, consider what would happen if HALT′ were to be given *itself*—this is the diagonalization part—and an arbitrary input, say 0 for instance.

- if $\mathsf{HALT}'(\mathsf{HALT}', 0)$ terminates, then $\mathsf{HALT}(\mathsf{HALT}', 0)$ returns *true*, in which case HALT′ loops forever.
- if $\mathsf{HALT}'(\mathsf{HALT}', 0)$ fails to terminate then $\mathsf{HALT}(\mathsf{HALT}', 0)$ returns *false*, in which case HALT′ terminates, returning *true*.

In other words,

$$\mathsf{HALT}'(\mathsf{HALT}', 0) \text{ terminates iff } \mathsf{HALT}'(\mathsf{HALT}', 0) \text{ doesn't terminate.}$$

This is a logical contradiction, so the premise—that HALT exists in the first place—must be invalid.

∎

Remark: The theorem does not say HALT can *never* tell whether P terminates. We can surely write programs the detect some looping patterns. However, such a program cannot *always* answer the question for all possible programs, $p \in STMT$.

The theorem does say not that HALT will sometimes get the wrong answer. With the rigorous definition of PL interpretation, we be would be able to verify that

(a) whenever $\mathsf{HALT}(p, i)$ returns *true* p terminates on i, and

(b) whenever $\mathsf{HALT}(p, i)$ returns *false* p diverges on i.

So the theorem says that for some programs and some inputs, HALT must fail to terminate.

Diagonalization is a common technique, often but not always used in for proof-by-contraction. One example is the

Theorem. In any programming language there is a program, P whose output is P.

It is an interesting exercise to write a program that prints its own text. The theorem says that you can do this with any general-purpose programming language. Note, however, that you may have to accept some variations in “white space” (blanks and new-line characters). END REMARK

5.3.2 Infeasible Problems.

We have just seen that there are problem classes for which no computer program can solve all problem instances. There are also problem classes that are decidabe but *infeasible*, solvable but inherently hard for a computer to solve. In the mathematical theory of computational complexity, there is a a rather clear boundary between feasible and infeasible problems. Exercise 8 states that 2^n dominates *all* geometric functions, n^x.

Furthermore, some problems are inherently exponential. In other words, it is not just the case that the best *known* decision algorithm is $O(2^n)$, but it can be proved that any decision algorithm must be at least $O(2^n)$.

Perhaps surprisingly, some key examples of inherently infeasible problems include basic problems we encountered in Chapter 2, including:

(a) Deciding whether a proposition is a tautology.

(b) Finding a case for which a proposition is true.

That these are decidable problems should be evident, since we have solved small instances systematically with truth tables. Analysis by truth table is exponential because the number of cases grows exponentially with the number of variables in the formula. What may be surprising is the fact that there is no way to improve on this exponential cost, in general.

Problem (b), above, is called the *satisfiability problemsatisfiability problem*. It has special status because it is “representative” of the class NP of infeasible problems for which solving is infeasible but verifying a solution is feasible.

5.3.3 Orders of Infinity.

Theorem 5.6 shows that there are different or *orders* of infinity. In particular, $|\mathbb{R}|$ is strictly larger than $|\mathbb{W}|$ in the sense that $\mathbb{W}$ is countable and $\mathbb{R}$ is not. A question that naturally arises is: how many infinities lie between $|\mathbb{W}|$ and $|\mathbb{R}|$? Provisionally, the answer is that there are discrete levels of infinity, that $|\mathbb{W}|$ is the lowest level and $|\mathbb{R}$ the next larger level, and there are none between. This is called the *Continuum Hypothesis*; which has the unusual status of being neither

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

provable nor disprovable from the standard assumptions of mathematics, which is not to say that it is true or false, but that its truth cannot be derived from the usual foundations.

Exercises 5.3

1. In a programming language of your choice, write a program that outputs its own text.

5.4 Additional Problems

Most of these exercises have to do with applications of induction to program performance analysis.

Exercises 5.4

1. (hard) Prove:For all $n \in \mathbb{N}$,

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Now take another look at Exercise 3.2.5. HINT: This problem's proof uses the result of another exercise in this book.

2. Let $A = \{\texttt{0}, \texttt{1}\}$ and consider the language of *binary strings*, A^*. A binary string $b_n\, b_{n-1} \cdots b_2\, b_1\, b_0$ Can be interpreted as a *base 2 numeral* representing

$$\sum_{i=0}^{n} \hat{b_i} \cdot 2^i$$

where $\hat{b}$ is the numeric value of "bit" b, namely

$$\hat{b} = \begin{cases} \text{the } \textit{number } 0 \text{ if } b \text{ is the letter } \texttt{0} \\ \text{the } \textit{number } 1 \text{ if } b \text{ is the letter } \texttt{1} \end{cases}$$

For alphabet $B = \{\texttt{0}, \texttt{1}, \exists\}$, define the language $L \subseteq B^+$—similar to similar the L in Section 7.7:

$$\begin{array}{rl} 1. & \exists \in L \\ 2a. & u \in L \Rightarrow \texttt{0}\hat{\ }u \in L \\ 2b. & u \in L \Rightarrow \texttt{1}\hat{\ }u \in L \\ 3. & \text{n. e.} \end{array}$$

(a) Define a recursive function $V : L \to \mathbb{N}$ that gives the binary value of a word in L.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

(b) Define a recursive function $I\colon L \to L$ that, given the binary word representing the number $n \in \mathbb{N}$, returns the word representing $n+1$. That is, prove that

$$\text{For all } w \in L,\ V(I(w)) = V(w) + 1$$

(c) Define an "addition" function, $A\colon (L \times L) \to L$, and prove

$$\text{For all } u, v \in L,\ V(A(u,v)) = V(u) + V(v)$$

3. Define $F\colon \mathbb{N} \to \mathbb{N}$ and $G\colon \mathbb{N}^2 \to \mathbb{N}$ as follows:

$$\begin{array}{rcl} F(0) & = & 1 \\ F(k+1) & = & (k+1) \cdot F(k) \end{array} \qquad \begin{array}{rcl} G(0,m) & = & m \\ G(k+1,m) & = & G(k,\ m \cdot (k+1)) \end{array}$$

(a) Prove by induction on $n \in \mathbb{N}$: For all $n, m \in \mathbb{N}$, $G(n,m) = m \cdot G(n,1)$.

(b) Prove: For all $n \in \mathbb{N}$, $F(n) = G(n,1)$.

4. Define $F\colon \mathbb{N} \to \mathbb{N}$ and $T\colon \mathbb{N}^3 \to \mathbb{N}$ as follows:

$$\begin{array}{rcl} F(0) & = & 1 \\ F(1) & = & 1 \\ F(k+2) & = & F(k) + F(k+1) \end{array} \qquad \begin{array}{rcl} FT(0,n,m) & = & n \\ FT(k+1,n,m) & = & FT(k,m,n+m)) \end{array}$$

Prove: For all $n \in \mathbb{N}$, $F(n) = FT(n,1,1)$

5. Let $a \in \mathbb{N}$. The function $T\colon \mathbb{N} \to \mathbb{N}$ is defined recursively by

$$\begin{array}{rcl} T(0) & = & 0 \\ T(1) & = & a + \frac{1}{2} \\ T(k+2) & = & 2T(k+1) - T(k) + 1 \end{array}$$

Prove that $(n^2/2) + an$ satisfies the recurrence, that is, *for all natural numbers n*

$$T(n) = \frac{n^2}{2} + an$$

6. Let $a \in \mathbb{N}$. The function $T\colon \mathbb{N} \to \mathbb{N}$ is defined recursively by

$$\begin{array}{rcl} T(0) & = & a \\ T(k+1) & = & T(k) + k + 1 \end{array}$$

Prove that for all $n \in \mathbb{N}$,

$$T(n) = a + \frac{n^2 + n}{2}$$

7. The function $q\colon \mathbb{N}^2 \to \mathbb{N}$ is defined:

$$\begin{array}{rcl} q(0,m) & = & m \\ q(k+1,m) & = & 1 + q(k,m) \end{array}$$

Prove that for all $n \in \mathbb{N}$, $q(n,0) = q(0,n)$.

Copyright © 2008 Steven D. Johnson

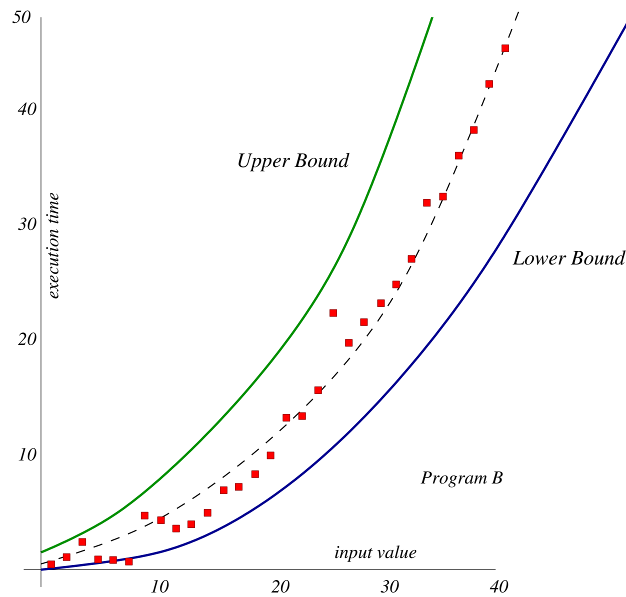


Figure 5.2: Bounding functions for a program.

Chapter 6

Relations

After sets, the most fundamental concept we will use is that of a *relation*.

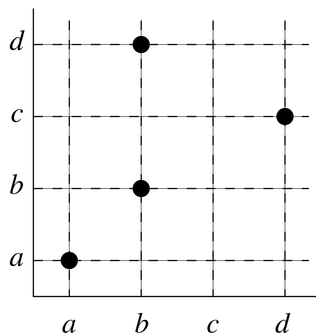
Definition 6.1 *If A and B are sets, then any subset of $A \times B$ may be called a relation from A to B, or equivalently, a relation with domain A and range B.*

We often write simply $R \subseteq A \times B$ to say, "R is a relation from A to B." This chapter is devoted to introducing the rather extensive vocabulary used in classifying and describing relations. We shall begin with the very important class of *functions*, and then discuss a number of other classes.

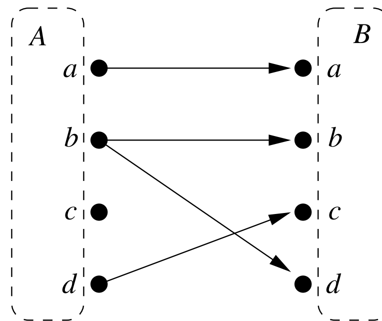
It is often helpful to draw pictures of relations, or *graphs* . In analytic geometry, a common form of graph is the *Cartesian product*. In this form, the domain and range are laid out on horizontal and vertical axes. Elements of the relation are shown according to their coordinates on the resulting plane. For example, let the domain A be the set $\{a, b, c, d\}$; let the range B be the set $\{a, b, c, d\}$; and consider the relation from A to B,

$$R = \{(a, a), (b, b), (b, d), (d, c)\}$$

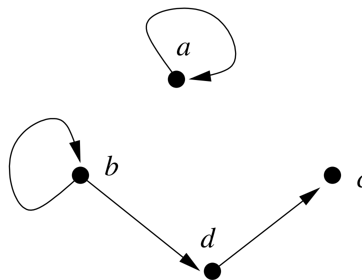
A Cartesian graph of R, looks like this:



Another way to represent R, called a *bipartite graph*,[1] is drawn as follows. First, write down all the elements of A in a column (or row). Next, write down all the elements of B in another column. Finally, whenever there is an ordered pair, $(x, y) \in R$, draw an arrow from x to y. A bipartite graph of R is shown below:



When the domain and range of a relation are the same set, one can draw a *directed graph* representing the relation. In a directed graph, the elements are written down just once. Here is a directed graph of our example R:



In computer science we often deal with finite, discrete sets. The bipartite and directed graphs of relations on such sets sometimes convey more information than the corresponding Cartesian graphs, and we shall see them often in this book.

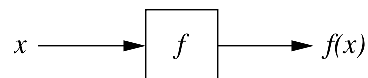
6.1 Functions

A common way to think about a function is as a rule. Sometimes, the rule is specified by a formula; for instance, we might write

$$f(x) = x^2 + 5x + 6$$

[1]Technically, a <u>relation</u> $R \subset A \times A$ is said to be *bipartite* if A can be divided into disjoint subsets in such a way that for all $(x, y) \in R$, x and y come from different subsets. The graph shown here diagrams a particular instance of a bipartite relation in which the domain and range are disjoint.

to specify a parabolic function. There is the idea of a function as a "box" that computes a result based on its inputs:



This notion of a function is close to our notion of a computer program. However, the concepts of "function" and "program" differ in some fundamental ways and it is important not to identify them too closely.

We can also think of a function as a particular kind of relation. The property that distinguishes functions from other kinds of relations is that functions associate just one value for a given input.

Definition 6.2 *A function from X to Y is a relation $f \subseteq X \times Y$ such that for every $x \in X$ there is exactly one $y \in Y$ such that $(x, y) \in f$.*

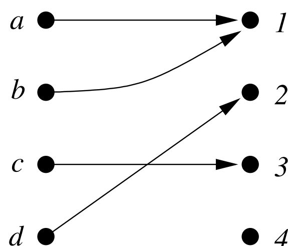
We write $f \colon X \to Y$ to indicate that f is a function from X to Y. If $(x, y) \in f$, we say that y *is the value of f at x,* and write $f(x) = y$.

Example

Ex 6.1 Let $X = \{a, b, c, d\}$; and let $Y = \{1, 2, 3, 4\}$. and let

$$= \{(a, 1), (b, 1), (c, 3), (d, 2)\}$$

The relation $f \subseteq X \times Y$ is a function because each of the possible inputs from domain X is associated with exactly one output. In the bipartite graph of f, we can see that there is just one arrow from each element of the domain.

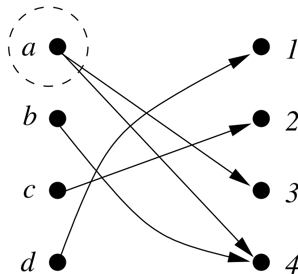


Example

Ex 6.2 Let sets X and Y be as given in the previous example, and let

$$g = \{(a, 3), (c, 2), (b, 4), (d, 1), (a, 4)\}$$

This relation is not a function because more than one value is associated with a:



It does not make sense to use the notation $g(x) = y$ when g is not a function, for then we would have

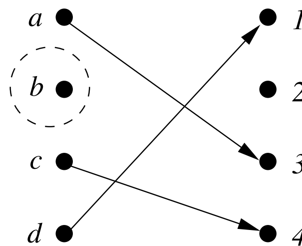
$$3 \stackrel{?}{=} g(a) \stackrel{?}{=} 4$$

Example

Ex 6.3 Let sets X and Y be as given in Example 6.1, and let

$$h = \{(a,3),(c,4),(d,1)\}$$

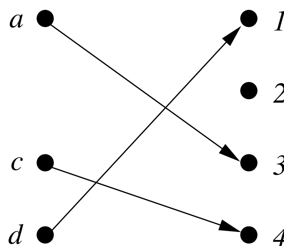
This relation is not a function because h has *no* value for b:



This too is a violation of the conditions of Definition 6.2.

Example

Ex 6.4 Considered as a relation from $\{a,c,d\}$ to $\{1,2,3,4\}$ the relation $h = \{(a,3),(c,2)(d,1)\}$ is a function:



Thus, the phrase "f is a function" is not meaningful unless we also specify the domain and range.

Copyright © 2008 Steven D. Johnson

The relation h in Example 6.3 is *almost* a function, except that it is undefined for some elements in its domain. In this book, the term "function" always refers to a completely defined relation, that is, a relation with a value for every possible input. We use the term *partial function* to describe relations such as h.

Definition 6.3 *A relation $f \subseteq X \times Y$ is called a* partial function from X to Y *if for every $x \in X$ there is at most one $y \in Y$ such that $(x, y) \in f$.*

Thinking again in terms of programs, a function is like a well behaved computation that produces an output for any chosen input. A partial function is just like a function except that, possibly, for some of the inputs the program gets "stuck in a loop" and produces no output. Notice though, that a relation which satisfies Definition 6.2 also satisfies the wording of Definition 6.3. In other words, we may say that a relation g is a partial function when it is unknown whether g is defined for all possible inputs.

It is sometimes useful to restrict our attention to a subset of a function's domain and that portion of its range to which that subset maps. The latter set set is called an *image* set. A converse notion is that of a *preimage* set. Given a function f with domain X and range Y, the image of f could be expressed in set-builder notation as

$$\{y \mid y \in Y \text{ and } y = f(x) \text{ for some } x \in X\}$$

and the preimage of f is

$$\{x \mid x \in X \text{ and } f(x) = y \text{ for some } y \in Y\}$$

These sets are specified in a more abbreviated form in the following definition.

Definition 6.4 *Let $f \subseteq X \times Y$ be a partial function; and let $A \subseteq X$ and $B \subseteq Y$. The* image of A under f *is the set*

$$\{f(a) \mid a \in A\}$$

and the preimage of B under f is the set

$$\{a \mid f(a) = b \text{ for some } b \in B\}$$

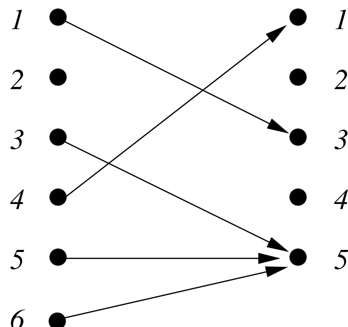
For notation, we use fA to denote the image of A under f and $f^{-1}B$ to denote the preimage of B under f.

Example

Ex 6.5 Let $X = \{1, 2, 3, 4, 5, 6\}$ and $Y = \{1, 2, 3, 4, 5\}$, and consider the partial

Copyright © 2008 Steven D. Johnson

function g described by the following bipartite graph:



The following are examples of image and preimage sets:

$$\begin{aligned} gX &= \{1,3,5\} \\ g^{-1}Y &= \{1,3,4,5,6\} \\ g\{4,5,6\} &= \{1,5\} \\ g^{-1}\{2,4\} &= \emptyset \\ g^{-1}\{4,5\} &= \{3,5,6\} \end{aligned}$$

The proof of the following proposition reminds us of just what qualities a function must have. Notice that the argument that $f(x)$ is unique breaks down into two simple arguments; one showing that f has *at least one* value at x; and the other showing that f has *at most one* value at x.

Proposition 6.1 *If $f \subseteq X \times Y$ is a partial function, then when considered as a relation from $f^{-1}Y$ to Y, f is a function.*

PROOF: Since $f^{-1}Y$ is precisely the subset of X for which f is defined, it follows that $f \subseteq f^{-1}Y \times Y$. For the same reason, we know that f has *at least* one value for every $x \in f^{-1}Y$. Since we have assumed it is a partial function, f also has *at most* one value for every $x \in f^{-1}Y$ and is therefore a function.

∎

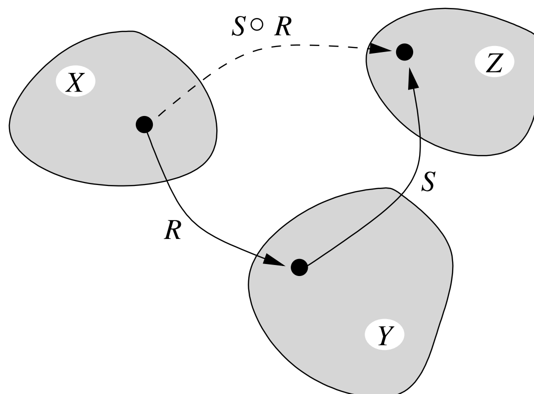
Considered as relations, functions have some very pleasant properties which we often take for granted. The following definition helps illustrate this point, and also provides a useful way to build new relations.

Definition 6.5 *Given two relations, $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, the* composition of R and S, *written $S \circ R$, is a relation from X to Z defined by:*

$$S \circ R = \{(x,z) \mid \textit{for some } y \in Y,\ (x,y) \in R \textit{ and } (y,z) \in S\}$$

The picture below illustrates the idea of composition. It defines an edge between elements of X and Z whenever they are both related to a common element in

Y:



The next proposition establishes that composing two functions yields a function.

Proposition 6.2 *Let A, B, and C be sets and suppose there are two functions, $f\colon A \to B$ and $g\colon B \to C$. Then $g \circ f$ is a also a function.*

Proof: Let $u \in A$. Since f is a function, there is a unique $f(u) \in B$, such that $(u, f(u)) \in f$. Since g is a function, there is a unique $g(f(u)) \in C$, such that $(f(u), g(f(u))) \in g$. By definition, we have $(u, g(f(u))) \in g \circ f$, and we have shown that this ordered pair is unique with respect to u. Since u was arbitrary, $g \circ f$ is a function.

∎

Remark: Function composition is usually defined without reference to the more general notion of composition of relations. The typical wording of the definition is

> *Let $f\colon A \to B$ and $g\colon B \to C$. The* composition of g and f *is defined by $(g \circ f)(x) = g(f(x))$.*

Our proof of Proposition 6.2 establishes that $g \circ f$ is "well defined"; that is, $g \circ f$ is actually a function. Verifying well-defined-ness it is an important part of the defining process. but it is often left to the reader. End Remark

The following definition describes three properties that a function might have.

Definition 6.6

(a) *If $f\colon X \to Y$ has the property that every $y \in Y$ is a possible output of f, we say f is a* surjection *or* onto*. More formally, we say $f\colon X \to Y$ is surjective iff for each $y \in Y$ there is an $x \in X$ such that $f(x) = y$.*

(b) *If* $f\colon X \to Y$ *always sends distinct elements of* X *to distinct elements of* Y*, we say* f *is an* injection *or* one-to-one. *More formally, we say* $f\colon X \to Y$ *is injective iff for every* x *and* $x' \in X$, $x \neq x'$ *implies that* $f(x) \neq f(x')$. *An equivalent statement of this property is: If* $f(x) = f(x')$ *then* $x = x'$.

(c) *If* $f\colon X \to Y$ *is called a* bijection *when* f *is both one-to-one and onto.*

In Chapter 1, the "property" formulas used in set-builder notation are true-or-false questions to be asked of any candidate for membership in the set. In fact, they are functions over the appropriate domain whose range consists of truth-values.

Definition 6.7 *Given any set* A*, a function* $f\colon A \to \{\mathit{true}, \mathit{false}\}$ *is called a* predicate on A. *In the context of a set description,*

$$S = \{x \in U \mid P[x]\}$$

the predicate P *on* U *is called the* characteristic function of S.

6.1.1 Infix Notation

If $f\colon X \times Y \to Z$ we say f is a *two-place* function which takes its first argument from X and its second argument from Y. By Definition 6.2, f must be a set of ordered pairs, not ordered triples nor anything else. Such an f is a relation from $X \times Y$ to Z, and so it is a subset of $(X \times Y) \times Z$. A typical element of f is $((x, y), z)$. By the notation introduced after Definition 6.2, we should write

$$f((a, b)) = c$$

but it is conventional in mathematics to write

$$f(a, b) = c$$

instead. Similarly, we write $g(a, b, c)$ for the result of a *three-place function*, $g\colon A \times B \times C \to D$, and so on.

When f is an n-place function, we sometimes say that f *is of rank* n.

Often, the expressions for common two-place functions are even more concise. For example, addition is a two-place function, $+\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. So, using to our "general purpose" notation for functions, we would write sums as, for example,

$$+(3, 5) = 8$$

Of course we don't write sums that way; instead we write

$$3 + 5 = 8$$

"$+(3, 5)$" is called *prefix* notation and "$3 + 5$" is called *infix* notation. Infix is the usual notation for two-place arithmetic operators, and for many other

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

two-place operators as well (such as concatenation and the logical connectives discussed in Chapter 1). We use infix in the next definition, which gives some special properties of two-place functions.

Special infix notation for 3-place functions is not as common, but is still sometimes used. One example from mathematics is modulus arithmetic: the formula

$$a + b \bmod n$$

stands for the remainder of $a+b$ on division by n. Another example—this time from computer programming—is the conditional expression:

$$\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2$$

Definition 6.8 *Let '$\odot$' be a two-place function, $\odot\colon A \times A \to A$. We say $\odot$ is*

(a) commutative *iff for all $x, y \in A$, $x \odot y = y \odot x$;*

(b) associative *iff for all $x, y, z \in A$, $x \odot (y \odot z) = (x \odot y) \odot z$.*

(c) *Finally, $e \in A$ is an* identity *for $\odot$ iff for all $x \in A$, $x \odot e = e \odot x = x$.*

Example

Ex 6.6 Integer addition is associative and commutative with zero as an identity. Integer multiplication is associative and commutative with 1 as an identity. Integer subtraction is *not* commutative or associative, and also has no identity.

Example

Ex 6.7 Concatenation of words is associative and has an identity, ε, but is not commutative.

Example

Ex 6.8 Logical conjunction is associative and commutative with identity T. Logical disjunction is associative and commutative with identity F. Logical implication is neither associative nor commutative and has no identity;

Example

Ex 6.9 A boolean algebra (Definition 2.5) is both commutative and associative; and both of its binary operations has an identity.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Exercises 6.1

1. Let $A = \{1, 2\}$ and $B = \{2, 3, 4\}$. Which of the following relations from A to B are functions?

(a) $\{(1, 3),\ (2, 4)\}$
(b) $\{(1, 3),\ (1, 4)\}$
(c) $\{(1, 3),\ (1, 3)\}$
(d) $\{(1, 3),\ (2, 5)\}$
(e) $\{(2, 2),\ (1, 4)\}$

2. Is $\{(1, 2),\ (2, 3)\}$ a function

(a) from $\{(1, 2)\}$ to $\{(2, 3)\}$?
(b) from $\mathbb{N}$ to $\mathbb{N}$?
(c) from $\{1, 2\}$ to $\mathbb{N}$?
(d) from $\{1, 2, 3\}$ to $\{2, 3\}$?
(e) from $\{1, 2, 3\}$ to $\{1, 2, 3\}$?

3. Let $A = \{1, 2\}$ and $B = \{2, 3, 4\}$.

(a) List a relation that is an injective function from A to B.
(b) List a relation that is a surjective function from B to A.
(c) List two bijections from B to B.

4. Let $f : A^2 \to A$ be given by the following table:

x	y	$f(x, y)$
1	1	1
1	2	2
2	1	2
2	2	2

Show that f is commutative and associative. What is the identity of f?

5. Let $f : A \to B$ and suppose S and T are both subsets of A.

(a) Prove that $f(S \cap T) \subseteq fS \cap fT$.
(b) Take both A and B to be $\{a, b, c\}$. Define a function f and two subsets S and T for which $f(S \cap T) \neq fS \cap fT$
(c) State a condition under which, in general, $f(S \cap T) = fS \cap fT$

6. Let $f : A \to B$ and $S \subseteq A$ and $T \subseteq B$ Prove or disprove the following:

(a) $f^{-1}(fS) = S$.
(b) $f(f^{-1}T) = T$.

7. *Prove*: If $f : X \times X \to X$, and e and e' are both identities of f, then $e = e'$.

8. *Prove*: If $f : A \to B$ and $g : B \to C$ are onto functions, then $g \circ f$ is onto.

9. *Prove*: If $f : A \to B$ and $g : B \to C$ are one-to-one functions, then $g \circ f$ is one-to-one.

10. *Prove*: if $f : A \to B$ is a bijection, then there exists a bijection from B to A.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

6.2 Relations on a Single Set

Let us now consider the class of relations $R \subseteq A \times A$, that is, relations whose domains and ranges are the same. Recall that such relations (if they are small enough) can be represented as directed graphs. The next definition establishes basic terminology for describing relations of this kind.

Definition 6.9 *A relation $R \subseteq A \times A$ is*

(a) *reflexive iff for every $a \in A$, the pair $(a, a) \in R$.*

(b) *symmetric iff $(x, y) \in R$ and $x \neq y$ implies $(y, x) \in R$*

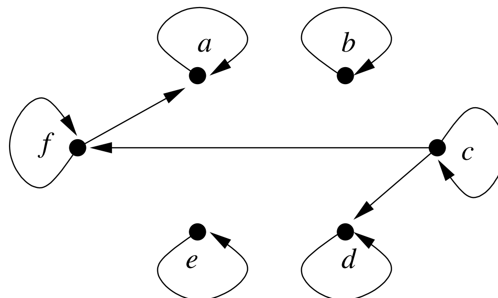
(c) *antisymmetric iff $(x, y) \in R$ and $x \neq y$ implies $(y, x) \notin R$.*

(d) *transitive iff for every $(x, y) \in R$ and $(y, z) \in R$, $(x, z) \in R$.*

The term *asymmetric* is sometimes used for a relation that is not symmetric. This is not the same as being antisymmetric (See Exercise 4).

Example

Ex 6.10 Let $A = \{a, b, c, d, e, f\}$. The relation R, whose graph is shown below, is reflexive.



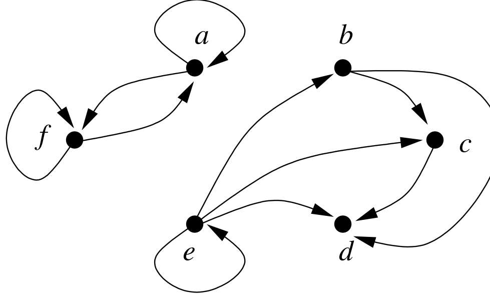
It is not symmetric because, for example, $(c, f) \in R$ but $(f, c) \notin R$. The predicate "$(x, y) \in R \Rightarrow (y, x) \in R$" is false in at least one case.

Example

Ex 6.11 Let $A = \{a, b, c, d, e, f\}$, as in the previous example. The relation T whose graph is shown below is transitive, but is neither reflexive, nor symmetric,

Copyright © 2008 Steven D. Johnson

nor irreflexive, nor antisymmetric.



Saying that a relation is "not symmetric" (or non-symmetric) is not the same as saying it is antisymmetric. Here for example, we have (a, f) and (f, a) in T (so T isn't antisymmetric), but we also have $(e, c) \in T$ while $(c, e) \notin T$ (so T isn't symmetric, either).

The notion of transitivity suggests that whenever there are two arrows "in sequence," then there is a *single* arrow from the tail of the first to the head of the second. In the relation T of Example 6.11, for example, we have:

$(e, b) \in T$ and $(b, c) \in T$ but also $(e, c) \in T$;
$(b, c) \in T$ and $(c, d) \in T$ but also $(b, d) \in T$;
$(e, c) \in T$ and $(c, d) \in T$ but also $(e, d) \in T$;
and so on.

In general, let R be transitive, and (a_1, a_2), (a_2, a_3), and (a_3, a_4) all be arrows in R. By transitivity, $(a_1, a_3) \in R$, so, applying transitivity to (a_1, a_3) and (a_3, a_4), the arrow $(a_1, a_4) \in R$. Clearly, this argument can be extended to any chain of arrows. We begin by formalizing the notion of a "chain of arrows."

Definition 6.10 *Let $R \subseteq A \times A$ be a relation. A path from a to b in R is a sequence*

$$\langle x_1, x_2, \ldots, x_n \rangle$$

such that

(a) $n \geq 1$
(b) *for each i, $x_i \in A$*
(c) *$x_1 = a$ and $x_n = b$*
(d) *for each i such that $1 \leq i < n$, $(x_i, x_{i+1}) \in R$*

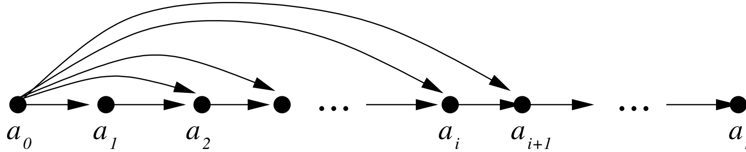
If $a_0 = a_n$, we call the path a cycle. We say n is the length A graph which has no cycles is said to be acyclic.

Proposition 6.3 *R is transitive iff whenever there is a path from a to b in R, then there is an edge $(a, b) \in R$.*

Copyright © 2008 Steven D. Johnson

PROOF: ($\Leftarrow$) Assume that if there is a path from a to b in R, then there is an arrow from a to b in R. We would like to show that R is transitive, that is, if $(a_1, a_2) \in R$ and $(a_2, a_3) \in R$, then $(a_1, a_3) \in R$. If $(a_1, a_2) \in R$ and $(a_2, a_3) \in R$, then $\langle a_1, a_2, a_3 \rangle$ is a path from a_1 to a_3 in R. By the assumption, since there is a path from a_1 to a_3 in R, there is an arrow from a_1 to a_3 in R, which is just what we need to show that R is transitive.

($\Rightarrow$) Assume that $\langle a_1, \ldots, a_n \rangle$ is a path in R, and that R is transitive. We shall show that for each i such that $1 \leq i \leq n$, there is an arrow $(a_0, a_i) \in R$. We shall do this by giving an *algorithm* that, starting with the arrow (a_0, a_1), builds up an arrow (a_0, a_n) by successive applications of transitivity: Imagine we have already built the arrow $(a_0, a_i) \in R$.



Since $\langle a_0, \ldots, a_i, a_{i+1}, \ldots, a_n \rangle$ is a path in R, we know that there is an arrow $(a_i, a_{i+1}) \in R$. Now transitivity requires that there be an arrow $(a_0, a_{i+1}) \in R$. We repeat this "extension" until we reach a_n.

∎

REMARK: Knowing that all paths are of finite length (Definition 6.10 says this by specifying a length, n) the algorithm certainly demonstrates that the desired edge, (a_1, a_n), exists and it even shows how to determine it. *Constructive* arguments of this form are often used to prove something exists. When a proof is based on an algorithm, one should first ask two questions:

(a) Is it "definite procedure?" At every step, it must be clear and unambiguous what to do next.

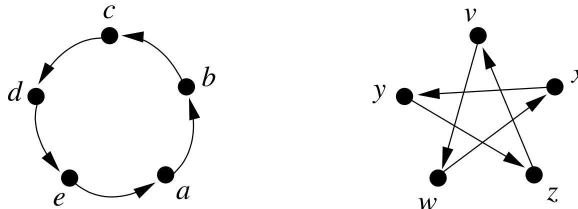
(b) Does it terminate? It must be evident that the procedure makes progress toward completion and does not go on forever.

Of course, the algorithm must also achieve its intended purpose—in this case, finding a edge from a_0 to a_n. Proof narratives often focuses on the "correctness," aspect, leaving it to the Reader to check definiteness and termination. END REMARK

In computing, we are often more interested in the *structure* of a graph than in details such as what names are given to its nodes. For example, the two graphs shown below have the same shape, even though their nodes are labeled

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

differently and they are laid out differently:



To “have the same shape,” an exact correspondence must exist between nodes, and in addition, this correspondence must extend to the edges in a particular way. These qualities are formalized in the next definition.

Definition 6.11 *Two directed graphs $R \subseteq A \times A$ and $S \subseteq B \times B$ are said to be isomorphic iff there exists a bijection $f \colon A \to B$ such that $(a, a') \in R$ iff $(f(a), f(a')) \in S$.*

Example

Ex 6.12 The bijection represents the correspondence between nodes. In the diagram above, we have $A = \{a, b, c, d, e\}$ and $B = \{v, w, x, y, z\}$. One possible bijection is

$$f = \{(a, x), (b, z), (c, w), (d, y), (e, v)\}$$

Check that there is a resulting correspondence between the edges of the new relation.

6.2.1 Attaching Information to Graphs

We have already seen graph diagrams that contain information other than the just the graph structure. For instance, the element a node represents is shown next to the node. These instances are annotations that make it easier to interpret the drawing. In many applications, it is desirable to affix information as part of the mathematical representation—as opposed to just depicting it in a drawing. Defining this association is called *labeling*.

Definition 6.12 *Given a relation $R \subset A \times A$, a* labeling *of R is a either a:*

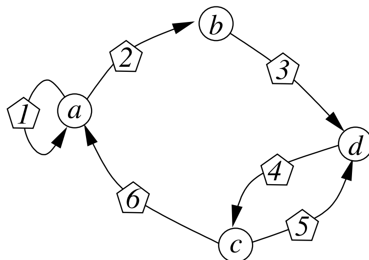
(a) edge labeling, *$\ell \colon R \to L$, mapping the edges to some set L, or*

(b) node labeling, *$\ell' \colon A \to L'$, mapping nodes to some set L'.*

Labelings may be depicted in any way that makes clear what the labeling is. The graph below has both node and edge labels, depicted as circles and

Copyright © 2008 Steven D. Johnson

pentangles, respectively.



$A = \{w, x, y, z\}$, $L = \{a, b, c, d\}$, $L' = \{1, 2, 3, 4, 5, 6\}$ and

$$\begin{array}{rcl} \ell\colon w & \mapsto & a \\ x & \mapsto & b \\ y & \mapsto & c \\ z & \mapsto & d \end{array} \qquad \begin{array}{rcl} \ell'\colon (w,w) & \mapsto & 1 \\ (w,x) & \mapsto & 2 \\ (x,y) & \mapsto & 3 \\ (y,z) & \mapsto & 4 \\ (z,y) & \mapsto & 5 \\ (z,w) & \mapsto & 6 \end{array}$$

In this picture the node's names, w, x, y and z, do not appear, but their labels do. Nevertheless, we will often refer to w as "node a," instead of the more technically correct "the node labeled by a."

Exercises 6.2

1. Let

$$\begin{array}{rcl} A & = & \{a, b, c, d, e\} \\ R & = & \{(a,b), (a,c), (b,a), (c,a), (c,d), (c,e), (d,c), (e,c)\} \end{array}$$

(a) Draw the bipartite graph representation of R.

(b) Draw the directed graph representation of R.

(c) Is R symmetric? reflexive? transitive?

2. Let $R = \{(a,a)\}$, $A = \{a\}$, and $B = \{a, b\}$. Is $R \subseteq A \times A$ reflexive? Is $R \subseteq B \times B$ reflexive? Draw both relations as bipartite graphs and directed graphs.

3. Let $A = \{a\}$, $B = \{a, b\}$.

(a) List all the relations $R \subseteq A \times A$.

(b) List all the relations $R \subseteq B \times B$.

(c) Of the relations in (a) and (b), which are reflexive? Symmetric? Tranansitive?

4. Draw a directed graph that is asymmetric but not antisymmetric.

5. Draw a directed graph that is symmetric and transitive, but not reflexive.

6. Draw all the directed graphs on a set with two elements. Indicate which of these graphs are isomorphic to one another.

Copyright © 2008 Steven D. Johnson

6.3 Trees

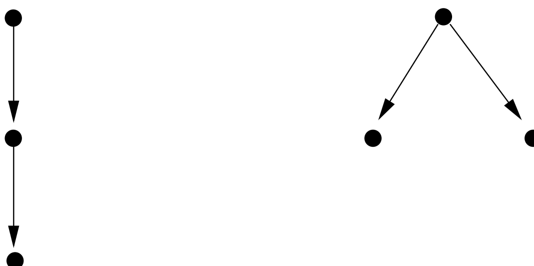
The graphs modeling data structures are usually far from transitive. Often there is at most a single way to get from one node to another. An important class of graphs used in data structures is the class of *trees*:

Definition 6.13 *A tree is a finite acyclic directed graph $R \subseteq A \times A$ in which there is one node (called the root) with in-degree 0, and every other node has in-degree 1. A node in a tree with out-degree 0 is called a leaf (See Figure 1.4.1).*

As always, it is necessary to include A in the declaration "$R \subseteq A \times A$ is a tree" (See Exercise 3).

Example

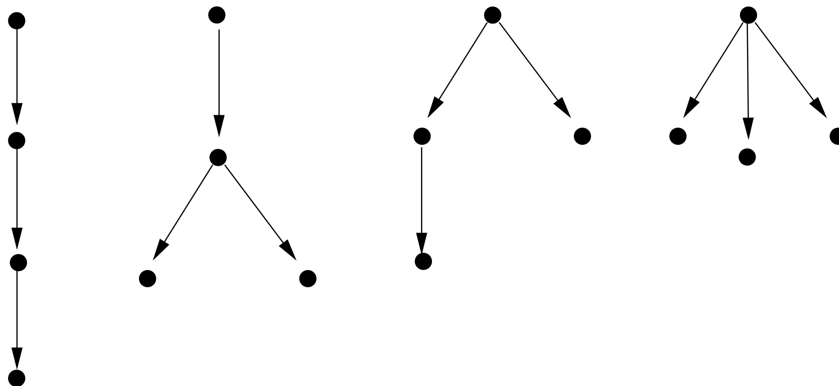
Ex 6.13 There are two distinct (that is, non-isomorphic, see Definition 6.11) trees for a set of three nodes. Here they are:



If we draw trees as above, with the root at the top and all arrows pointing downward, then we can omit the arrowheads.

Example

Ex 6.14 There are four distinct trees for a set of four nodes. Here they are:



Copyright © 2008 Steven D. Johnson

Theorem 6.4 *If $R \subseteq A \times A$ is a tree and $x \in A$ is not the root of R, then there is exactly one path from the root to x in R.*

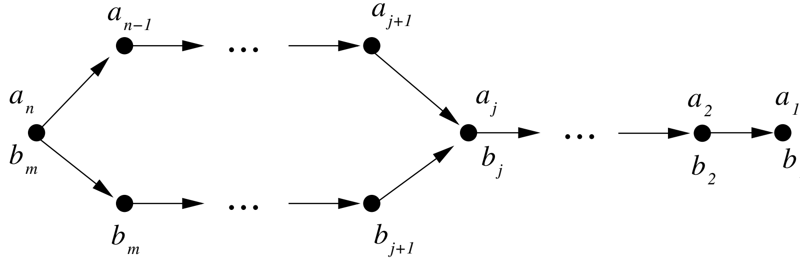
PROOF: We shall first construct one path from the root to x, and then show that it is unique. For the first part, see Figure 6.3. Since x is not the root, x has in-degree 1, so there must be a path $\langle a_1, x \rangle$.

Assume we have constructed a path $\langle a_n, a_{n-1}, \ldots, x \rangle$. If a_n is the root, we are done. If a_n is not the root, then the in-degree of a_n is 1, so there must be some node a_n and arrow $(a_{n+1}, a_n) \in R$. Hence, $\langle a_{n+1}, a_n, \ldots, x \rangle$ is a path in R. Now, the length of this path can be no greater than the number of elements in A. For if some node were repeated, then R would have a cycle, contradicting the assumption that R is a tree. Therefore, our path-building procedure must eventually halt; but it can only halt by finding an a_{n+1} which is the root. In other words, the procedure must halt with a path from the root to x.

Now assume that there are two paths

$$\langle a_n, a_{n-1}, \ldots, x \rangle \quad \text{and} \quad \langle b_m, b_{m-1}, \ldots, x \rangle$$

with $a_n = b_m =$ the root. Then at some point the paths must converge: There is an integer j such that $a_{j+1} \neq b_{j+1}$, but $a_j = b_j$, $a_{j-1} = b_{j-1}, \ldots, a_1 = b_1$:



So $(a_{j+1}, a_j) \in R$ and $(b_{j+1}, a_j) \in R$. Therefore, a_j must have in-degree of at least 2, contradicting the assumption that R is a tree. Thus, there cannot be two paths and, by the previous argument, there must be at least one. This concludes the proof.

∎

REMARK: Like the proof of Theorem 6.3, this proof is constructive, describing two algorithms that iterate an unknown number of times before the argument is complete. Recall the discussion after Theorem 6.3 and decide whether whether the procedures described are definite and terminating. In both cases, what to do next is well determined and iteration is limited either by the finite size of the node set or the finite length of the path.

The proof narrative can be criticized as being "redundant" in the sense that the two parts are very similar and can rather easily be condensed into a single argument combining both existence and

Copyright © 2008 Steven D. Johnson

uniqueness. (See Exercise 6. However, the argument follows a form commonly used for uniqueness proofs:

existence: Show that there is *at least one* object with the desired property.

uniqueness: Show that there is *no more than one* object with the desired property. This is often done by assuming more than one exist and showing that this assumption leads to a contradiction.

This proof strategy is reflected in the phrase, "There is *one* and *only one* x with property $P(x)$," suggesting that there are two things to prove. END REMARK

Because of this unique-path property, it is easy to write programs that visit every node of a tree exactly once. One could use the same algorithms on a general directed graph if one could identify a tree "hidden" in the directed graph Such a hidden tree is called a *spanning tree* of the graph.

Definition 6.14 *If $G \subseteq A \times A$ is a directed graph and if $R \subseteq A \times A$ is a tree and $R \subseteq G$, then we say R is a spanning tree of G.*

As Figure 6.3 illustrates, a single graph may have many different spanning trees. If r is the root of a spanning tree R of G, there must be a path in R from r to x for every node in the tree. Since $R \subseteq G$, there must be a path in R from r to x for every node in G. Theorem 6.5 states that this property is all that is needed for G to have a spanning tree.

Definition 6.15 *If $G \subseteq A \times A$ is a rooted graph iff there is a node $r \in A$ (the root) such that for every $x \in A$ there is a path from r to x in G.*

Theorem 6.5 *Let $G \subseteq A \times A$ by a finite rooted graph with root r. Then G has a spanning tree with root r.*

PROOF: We shall construct a sequence of trees

$$\begin{array}{ccc} R_1 & \subseteq & A_1 \times A_1 \\ R_2 & \subseteq & A_2 \times A_2 \\ & \vdots & \\ R_k & \subseteq & A_k \times A_k \\ & \vdots & \end{array}$$

each with root r and with $R_k \subseteq G$. Each A_k will contain k nodes, so if G has N nodes, $A_N = A$ and R_N will be a spanning tree for G.

First, let $A_1 = \{r\}$ and $R_1 = \emptyset$. (You can check that $R_1 \subseteq A_1 \times A_1$ is a tree.) Now imagine we have built $R_k \subseteq A_k \times A_k$, with $k < N$, and let us construct

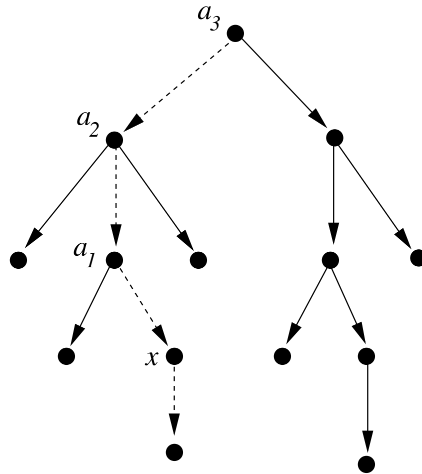


Figure 6.1: Construction of a path from the root of a tree

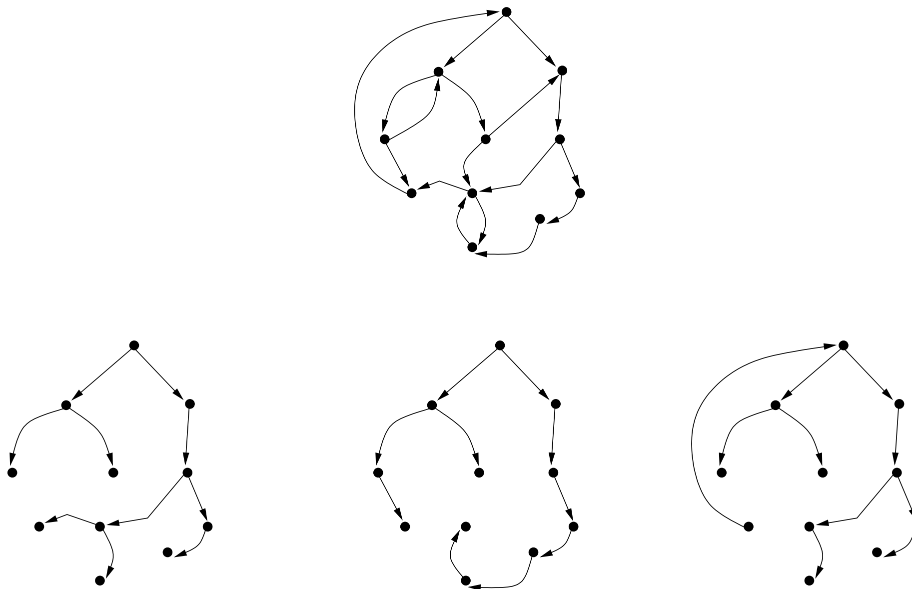


Figure 6.2: A graph and some of its spanning trees.

$R_{k+1} \subseteq A_{k+1} \times A_{k+1}$. Since A_k has k elements, and $k < N$, there must be some $z \in A$ such that $z \notin A_k$. Since G is rooted, let $\langle a_0, a_1, \ldots, a_p \rangle$ be a path from the root $r = a_0$ to $z = a_p$. Since $R_k \subseteq A_k \times A_k$, $r \in A_k$ and $z \notin A_k$, there must be some j such that $a_0,\ a_1, \ldots,\ a_j$ all belong to A_k, but $a_{j+1} \notin A_k$.

Set $A_{k+1} = A \cup \{a_{j+1}\}$ and $R_{k+1} = R_k \cup \{(a_j, a_{j+1})\}$.

Now $a_{j+1} \notin A_k$, so (a_j, a_{j+1}) is the only arrow to to a_{j+1}. So a_{j+1} has in-degree 1, and we could not have created a cycle by adding (a_j, a_{j+1}). Furthermore, $(a_j, a_{j+1}) \in G$, so $R_{k+1} \subseteq G$. Last, there is still no arrow ending at r, so r is the root of R_{k+1}. Hence, R_{k+1} has the required properties. Perform the construction N times, and R_N will be the desired spanning tree.

∎

Figure 6.3 shows the construction of a spanning tree as described in the theorem. Try the construction yourself, using different z's and different paths, to construct a different spanning tree.

Exercises 6.3

1. Draw all the nonisomorphic trees with five vertices.

2. A *binary tree* is a tree in which every nonleaf has an out-degree of two.

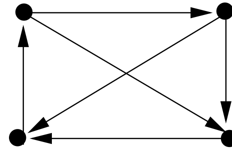
(a) Draw all the distinct (nonisomorphic) binary trees with five nodes.

(b) Draw all the distinct (nonisomorphic) binary trees with six nodes.

(c) Based on you answers to (a) and (b), state a property about binary trees.

3. Let $R = \{(a, b)\}$, $A = \{a, b\}$, $B = \{a, b, c\}$. Is $R \subseteq A \times A$ a tree? Is $R \subseteq B \times B$ a tree?

4. Draw all the spanning trees of the following directed graph:



5. *Prove:* If $A = \{a\}$, there exists exactly one $R \subseteq A \times A$ that is a tree.

6. Rewrite the proof of Theorem 6.4, combining the two parts of the proof into a single argument. This argument might begin, "Since x is not the root, it has in-degree 1, so (a_1, x) is the only edge leading to x in R; and $\langle a_1, x \rangle$ is the only path of length one ending at x. ...."

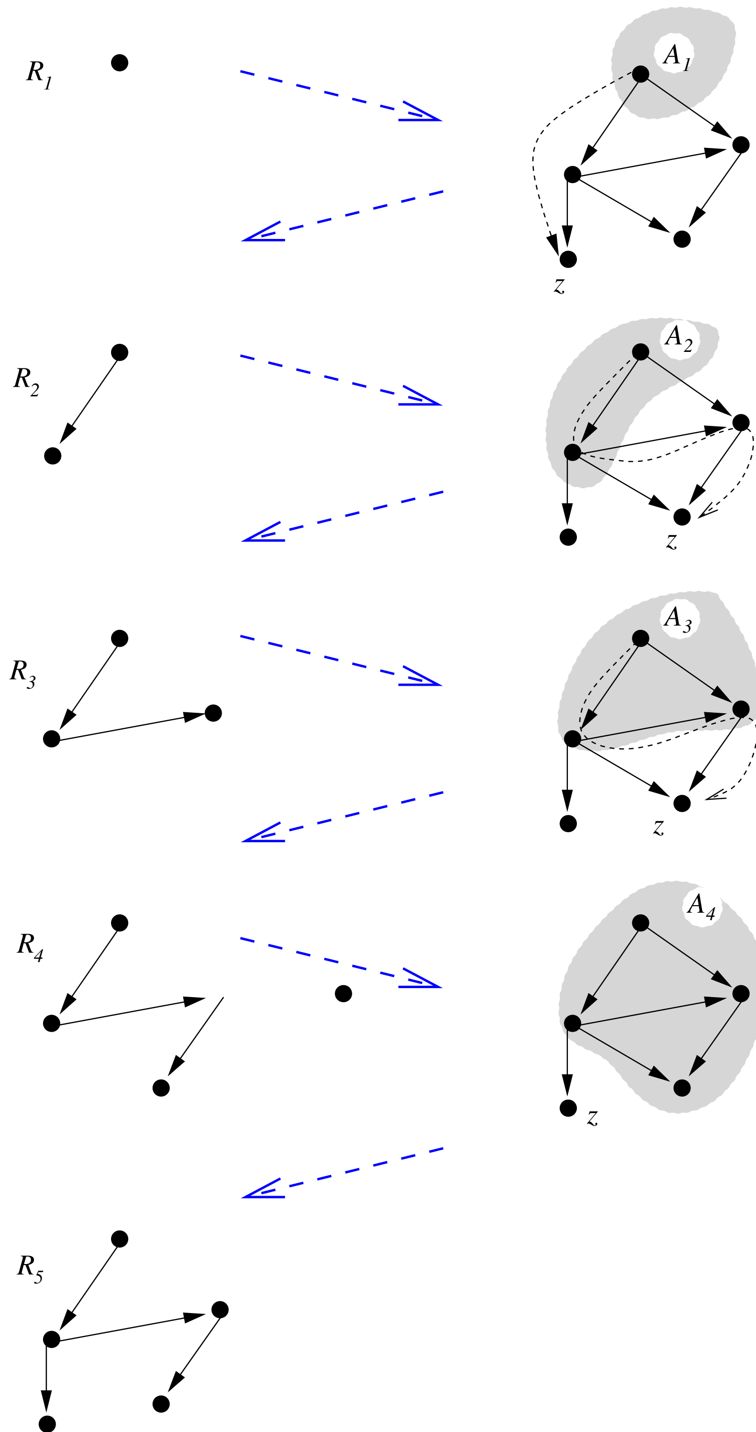


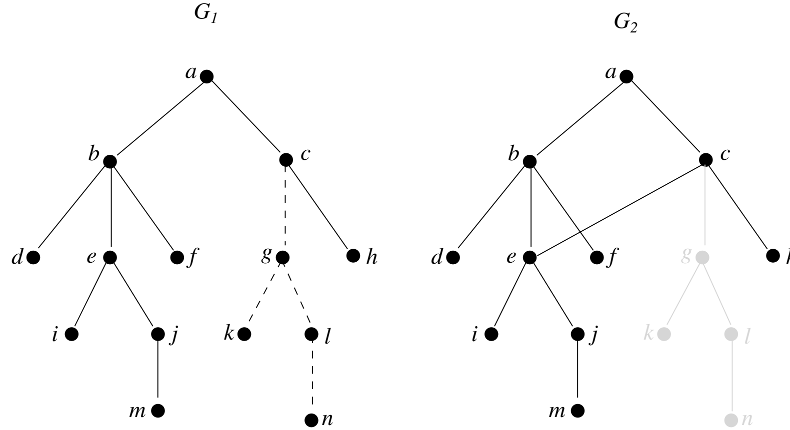
Figure 6.3: Construction of a spanning tree

6.4 DAGs

Computer data structures often take advantage of the fact that every datum has an address in the computer's memory. If two data structures contain exactly the same information, that information coalesced into a single object whose address may be shared by different access points to the same address. This kind of sharing suggests a kind of graph structure similar to that of a tree but more compact.

Definition 6.16 *A* directed acyclic graph*, or* DAG*, is a rooted graph containing no cyclic paths*

Consider the tree G_1 on the left below. It has isomorphic *sub*trees rooted at e and g. DAG G_2on the right is obtained by adding the edge (c, e) and removing notes g, k, l and n as well as the edges among them.



The two graphs can be thought of a being "structurally" similar in the sense that whenever there is a path from a to x in G_1, there is a corresponding path in G_2 from a to a node corresponding to x in G_2. For instance, the path

$$\langle a, c, g, l, n \rangle \text{ in } G_1$$

corresponds to

$$\langle a, c, e, j, m \rangle \text{ in } G_1$$

The two paths contain different nodes, so this notion of "similarity" must take into account a correspondence between nodes, as was the case with graph isomorphism defined earlier (Defn. 6.11). This is a weaker correspondence capturing the idea that one graph structure can be embedded in another.

Definition 6.17 *Two directed graphs $R \subseteq A \times A$ and $S \subseteq B \times B$ are said to be* homomorphic *iff there exists a function $h\colon A \to B$ such that if $(a, a') \in R$ then $(h(a), h(a')) \in S$. Such a function h is called a* homomorphism *from A to B.*

Copyright © 2008 Steven D. Johnson

Example

Ex 6.15 *Graphs G_1 and G_2, shown earlier are homomorphic under the mapping h given by*

$h\colon a \mapsto a$	$h\colon e \mapsto e$	$h\colon i \mapsto i$	$h\colon m \mapsto m$
$h\colon b \mapsto b$	$h\colon f \mapsto f$	$h\colon j \mapsto j$	$h\colon n \mapsto m$
$h\colon c \mapsto c$	$h\colon g \mapsto e$	$h\colon k \mapsto i$	
$h\colon d \mapsto d$	$h\colon h \mapsto h$	$h\colon l \mapsto j$	

Exercises 6.4

1. Let $R \subseteq A^2$ $S \subseteq B^2$ $T \subseteq C^2$ and suppose that $f\colon A \to B$ and $g\colon B \to C$ are homomorphisms. Prove that the composition $g \circ f$ is a homomorphism.

2. Some textbooks define a graph homomorphism to be a surjective function:

> *Two directed graphs $R \subseteq A \times A$ and $S \subseteq B \times B$ are said to be homomorphic iff there exists a surjection $h\colon A \to B$ such that if $(a, a') \in R$ then $(h(a), h(a')) \in S$.*

Let us call this kind of homomorphism a *strong homomorphism*. Prove that the composition of two strong homomorphisms is a strong homomorphism.

6.5 Equivalence Relations*

Notions of equivalence are used throughout mathematics A fundamental kind of equivalence is *equality*, between numbers or sets for example. But there are also many ways that we might regard two distinct objects to be equivalent. For example, suppose we have a sack of marbles. We might regard two marbles as equivalent if they are the same size and color. Similarly, we might say two programs are equivalent if they produce the same output for a given input, ignoring other features such as speed, clarity, and so forth.

What qualities *must* the notion of equivalence have? In the first place, equivalence is a relation on some set of objects. The following three properties capture the sense of what equivalence means:

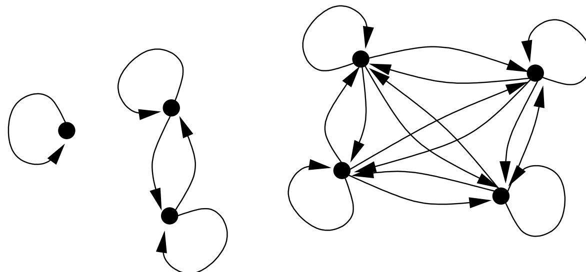
- Every object is equivalent to itself.
- Whenever x is equivalent to y, it is also the case that y is equivalent to x.
- Whenever x is equivalent to y and y is equivalent to z, it is also the case that x is equivalent to z.

In other words,

Copyright © 2008 Steven D. Johnson

Definition 6.18 *A relation that is reflexive, symmetric, and transitive is called an equivalence relation.*

Here is an example of an equivalence relation, depicted as a directed graph:



The picture shows that the nodes are divided into "clusters." Within each cluster, there is an arrow from any node to any node in the same cluster, but between different clusters there are no arrows. It is easy to see why there are no arrows between clusters: If we added an arrow from some node in one cluster to some node in another cluster, then in order to make the relation transitive, we would have to add arrows between all the nodes in the two clusters. The "clusters" are called *equivalence classes.*

Definition 6.19 *Let $R \subseteq A \times A$ be an equivalence relation, and let $a \in A$. The equivalence class of a under R, written $[a]_R$, is defined as*

$$\{a \in A \mid (a, a') \in R\}$$

When we can determine R from the context, we omit the "under R" and write just $[a]$. The "clusters" property may be expressed as follows.

Theorem 6.6 *If R is an equivalence relation on A and $a, b \in A$, then*

(a) *if $(a, b) \in R$, then $[a] = [b]$.*

(b) *if $(a, b) \notin R$, then $[a] \cap [b] = \emptyset$.*

Proof: (a) Assume $(a, b) \in R$. We shall show $[b] \subseteq [a]$ and $[a] \subseteq [b]$. To show $[b] \subseteq [a]$, let $x \in [b]$. Then $(b, x) \in R$. Since $(a, b) \in R$ and $(b, x) \in R$, by transitivity, $(a, x) \in R$. Hence, $x \in [a]$. Since we have shown any member of $[b]$ is also a member of $[a]$, $[b] \subseteq [a]$.

To show $[a] \subseteq [b]$, let $y \in [a]$. So $(a, y) \in R$ and since R is a symmetric relation, $(y, a) \in R$. We have assumed that $(a, b) \in R$ and since R is transitive, $(y, b) \in R$. By symmetry again, $(b, y) \in R$; and so $y \in [b]$. Hence $[a] \subseteq [b]$.

(b) We shall show $[a] \cap [b] \neq \emptyset$ implies $(a, b) \in R$. If $[a] \cap [b] \neq \emptyset$. Then there is some z such that $z \in [a]$ and $z \in [b]$. Since $z \in [a]$, we have $(a, c) \in R$, and since $z \in [b]$, we have $(b, c) \in R$ and by symmetry, $(c, b) \in R$. R is a transitive relation, so it follows that $(a, b) \in R$.

∎

Copyright © 2008 Steven D. Johnson

Corollary 6.7 *If $a, b \in A$, then either $[a] = [b]$ or $[a] \cap [b] = \emptyset$.*

PROOF: By Theorem 6.6, if $(a, b) \in R$, then $[a] = [b]$; and if $(a, b) \notin R$, then $[a] \cap [b] = \emptyset$.

∎

Sometimes, it is preferable to think of a set in terms of its equivalence classes, rather than its individual elements. In programming, for example, this is the difference between a *specification* and an *implementation*. Think of a library of mathematical routines. The user of the library may want a procedure to compute the square root of a number—any number of functionally equivalent routines could be written to do that. The implementer of the library needs to provide one representative of this equivalence class.

Definition 6.20 *If R is an equivalence relation on A, the quotient set A/R is $\{[a]_R \mid a \in A\}$.*

Example

Ex 6.16 Let $A = \{1, 2, 3\}$ and $R = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)\}$. Then R is an equivalence class on A and

$$\begin{aligned} [1]_R &= \{1, 2\} \\ [2]_R &= \{1, 2\} \\ [3]_R &= \{3\} \end{aligned}$$

So $A/R = \{\{1, 2\}, \{3\}\}$.

Now, A/R is a subset of the power set, $\mathcal{P}(A)$, and so we might ask: which subsets of $\mathcal{P}(A)$ are also quotient sets? That is, which subsets of $\mathcal{P}(A)$ are equal to A/R for some equivalence relation R? The following definition and theorem supply the answers and give us another way to characterize equivalence relations.

Definition 6.21 *Let A be a set. A subset Δ of $\mathcal{P}(A)$ is a partition of A iff*

(a) *each $S \in \Delta$ is nonempty, and*

(b) *for each $a \in A$ there is exactly one $S \in \Delta$ such that $a \in S$.*

Theorem 6.8 *A subset Δ of $\mathcal{P}(A)$ is a partition iff $\Delta = A/R$ for some equivalence relation R on A.*

PROOF: ($\Leftarrow$) If R is an equivalence relation, we claim A/R is a partition. If $a \in A$, then $a \in [a]$, so there is some $S \in A/R$ such that $a \in S$. By Corollary 6.7, $[a]$ is the *only* equivalence class containing a. Furthermore, each equivalence class is nonempty. So a/R is a partition.

Copyright © 2008 Steven D. Johnson

($\Rightarrow$) The definition of a partition implies that we can define a function, $f\colon A \to \Delta$, mapping each $a \in A$ to the set S of which it is an element. That is,

$$f(a) = S \text{ iff } a \in S$$

Define a relation $R \subseteq A \times A$ as follows

$$R = \{(a,b) \mid f(a) = f(b)\}$$

R is easily seen to be an equivalence relation. We claim $A/R = \Delta$. Let $a \in A$ and $f(a) = S$. Then

$$\begin{aligned} S &= \{x \mid f(x) = S\} \quad (\text{since } x \in S \text{ iff } f(x) = S) \\ &= \{x \mid f(x) = f(a)\} \\ &= [a]_R \end{aligned}$$

So for each $a \in A$, $[a] = f(a) \in \Delta$; hence, $A/R \subseteq \Delta$. Conversely, let $S \in \Delta$. By the definition of partition, S is nonempty, so choose $a \in S$. Hence, $f(a) = S$, and by the previous argument, $S = [a]_R$. So $\Delta \subseteq A/R$.

∎

Exercises 6.5

1. List A/R for each of the following equivalence relations:

(a) $A = \{x \in \mathbb{N} \mid 1 \leq x \leq 6\}$
$R = \{(x,y) \mid (x - y) \text{ is evenly divisible by } 3\}$

(b) $A = \{x \in \mathbb{N} \mid 1 \leq x \leq 8\}$
$R = \{(x,y) \mid x = 2^i k \text{ and } y = 2^j k \text{ for odd } k\}$

(c) $A = \{x \in \mathbb{N} \mid 1 \leq x \leq 24\}$
$R = \{(x,y) \mid x = 2^i 3^j k \text{ and } y = 2^{i'} 3^{j'} k' \text{ and } i + j = i' + j'$ and k, k' are not evenly divisible by 2 or 3$\}$

2. If A and B are sets and $f\colon A \to B$, define the *kernal* of f [denoted $\mathrm{Ker}(f)$] to be $\{(x,y) \mid x, y \in A \text{ and } f(x) = f(y)\}$. Prove that for any f, $\mathrm{Ker}(f)$ is an equivalence relation.

3. *Prove:* Suppose R is an equivalence relation on A. Define a set B and a function $f\colon A \to B$ such than $R = \mathrm{Ker}(f)$.

6.6 Partial Orders*

Set containment and numeric inequality, are examples of a class of relations called *partial orders*. These relations are of great importance in the theory of computer science.

Definition 6.22 *A relation that is reflexive, antisymmetric,*[2] *and transitive is called a partial order.*

[2] If the condition of antisymmetry is removed, the relation is called a *preorder*.

Example

Ex 6.17 *Set containment is a partial order. The proofs are left as exercises:*

- Reflexivity. *If S is any set, then $S \subseteq S$.*
- Antisymmetry. *If $S \subseteq T$ and $S \neq T$, then it is not the case that $T \subseteq S$.*
- Transitivity. *If $S \subseteq T$ and $T \subseteq U$ then $S \subseteq U$.*

A special case of partial order is one in which all the elements are related to all others.

Definition 6.23 *A total order is a partial order $R \subseteq A \times A$ with the additional property that for all $x, y \in A$, either $(x, y) \in R$ or $(y, x) \in R$.*

Example

Ex 6.18 $\mathbb{N}$*,* $\mathbb{Q}$ *and* $\mathbb{R}$ *are assumed to be totally ordered by the relation '$\leq$'.*

Example

Ex 6.19 *A tree is not a partial order because trees are neither reflective nor transitive. There are many times when one wants to "extend" the graph of a tree to make it into a partial order, incorporating concepts like "node m is a descendent of node n." Extending a tree T (or any graph for that matter) to a partial order is conceptually straightforward:*

(a) *To make $T \subseteq A \times A$ reflexive, add self-edges to every $a \in A$. The graph $T^r = T \cup \{(a, a) \mid a \in A\}$ is called the* reflexive closure of T.

(b) *To make T transitive, add an edge from (a, b) to T whenever there is a path $\langle a, n_1, \ldots, n_{k-1}, b\rangle$ in T. The resulting graph, call it T^*, is called the* transitive closure of T^r.

REMARK: Although it is intuitively clear what it is, writing down a definition of T^* is thought provoking. T^* could be defined as

$$T^* = T^r \cup \{(a, b) \mid \text{there is a path from } a \text{ to } b \text{ in } T^*\}$$

In this definition T^* is being defined in terms of *itself*, and we need to consider whether this equation is meaningful. A self referencing definitions are not always valid or even meaningful. This point is discussed further in Chapter 7. END REMARK

Example

Ex 6.20 *Define the relation $R \subseteq \mathbb{N}^2 \times \mathbb{N}^2$ on ordered pairs as follows:*

$((n,m),(k,l)) \in R$ *iff*

(a) $n \leq k$ *or*

(b) $n = k$ *and* $m \leq l$.

That that R is a partial order follows from the fact that '$\leq$' is a partial order. It is reflexive because, for all $n, m \in \mathbb{N}$, $((n,m),(n,m)) \in R$; and if $((n,m),(k,l)) \in R$ and $((k,l),(u,v)) \in R$, so is $((n,m),(u,v)) \in R$. To prove transitivity, there are several cases to consider, specifically,

- $n = k = u$
- $n = k \leq u$
- $n \leq k = u$
- $n \leq k \leq u$

Check that in each case, transitivity holds. Because this ordering is like an "alphabetical" listing, it is called the lexigraphic ordering *and is often denoted by the symbol $\leq^L$. Notice that even though '$\leq$' is a total order, '$\leq_L$' is not, and that even when the underlying ordering is not total, the lexigraphic ordering is still well defined.*

Example

Ex 6.21 *Let A be an alphabet and consider the language A^* of all words built from letters in A, including ε. If $u, v \in A^*$ the u is called a* prefix *of v if there exists a word w such that $u\hat{\ }w = v$. Check that the prefix relation is a partial order.*

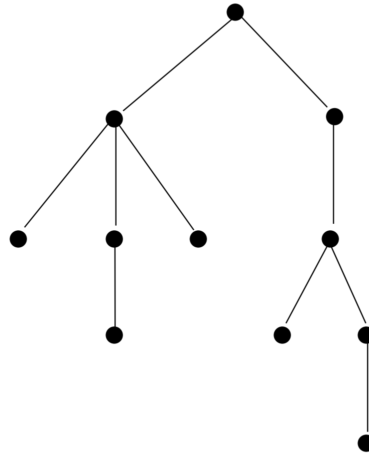
Exercises 6.6

1. In the lexigraphic ordering of $\mathbb{N}^2$, give an example of two elements that are not related.

2. In the prefix ordering of A^*, give an example of two elements that are not related.

3. Defintion 6.22 states that a *partial order* is a reflexive, anti-symmetric, transitive relation. Add the edges needed to extend this tree to a partial

order.



Exercises 6.6

1. Let A be a set. Prove that $\mathcal{P}(A)$ is a lattice with respect to the partial order '$\subseteq$'.

6.7 Decision Diagrams*

The boolean expression $E \equiv \overline{p}\,\overline{r} + \overline{q} + r$ has the truth table

p	q	r	E
0	0	0	1
0	0	0	0
0	1	0	1
0	1	0	0
1	0	0	1
1	0	0	0
1	1	0	1
1	1	0	1

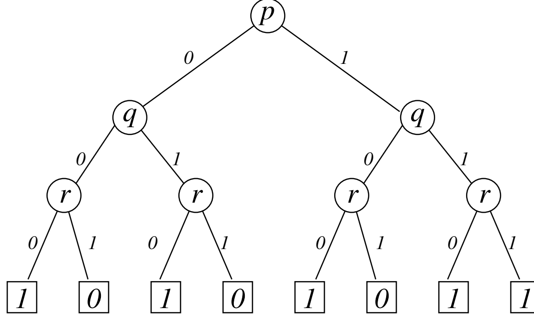
Section 2.4 introduced a way to represent this truth table in disjunctive normal form as

$$\overline{p}\,\overline{q}\,\overline{r} + \overline{p}\,q\,\overline{r} + p\,\overline{q}\,\overline{r} + p\,q\,\overline{r} + p\,q\,r$$

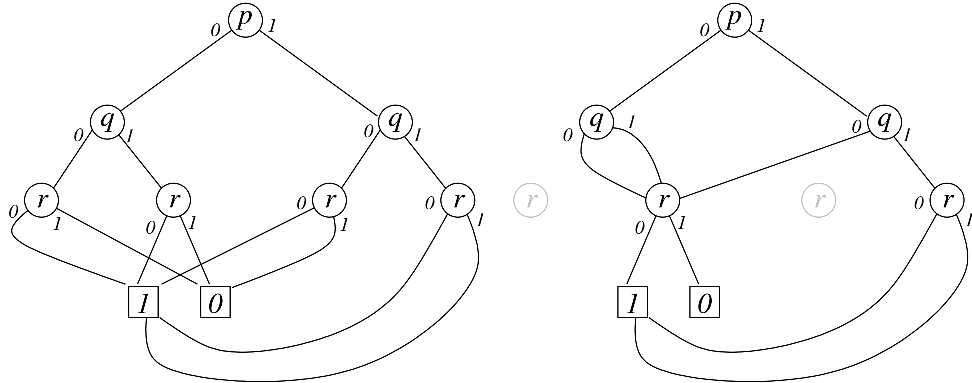
written here as a boolean term rather than a propositional formula. DNF was described as a "standard" representation that could be used to compare and analyze formulas, and some computer representations were mentioned.

Another way to represent terms is to use a tree. In the tree below, the nodes are labeled with the names of variables occurring in E, and the edges

with boolean values indicating whether the variable is true or false along that path.



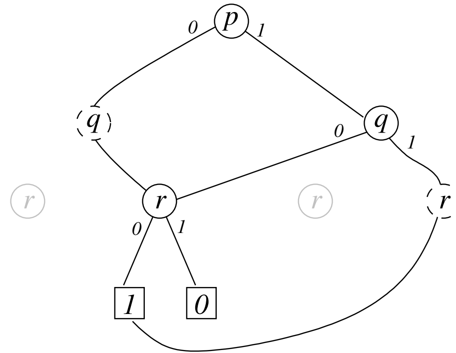
This is a particular kind of *decision tree*, called a *binary decision tree* because each non-leaf has out-degree 2. We will see more general classes of decision trees in the next chapter. A more compact representation of term E can be obtained by building a DAG to which the tree above is homomorphic. We can do this step-wise, from the bottom up, by locating isomorphic subtrees and eliminating all but one of them. This is done in two steps below, first eliminating redundant leaves, and then isomorphic subtrees rooted at r



So now we have a binary decision DAG in which every path from root to leaf corresponds to a path in the original tree. These paths, taken together, correspond to the clauses in a DNF.

In computer representations, additional transformations are used to make the representation still more compact. If the two branches from the node go to

the same target, that node can be eliminated.



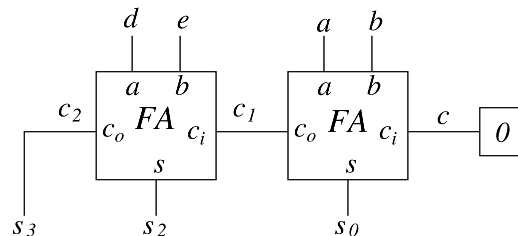
The resulting DAG is no longer homomorphic to the original tree (Can you see why?) unless the missing variable is somehow "remembered" when traversing the graph. The DAG above is called a *reduced ordered binary decision diagram*, or *ROBDD* for short[3]

Example

Ex 6.22 *Recall from 2.5 that a* full adder *is specified by boolean equations*

$$
\begin{aligned}
s &= \overline{a}\,\overline{b}\,c + \overline{a}\,b\,\overline{c} + a\,\overline{b}\,\overline{c} + a\,b\,c \\
c_o &= \overline{a}\,b\,c + a\,\overline{b}\,c + a\,b\,\overline{c} + a\,b\,c
\end{aligned}
$$

Suppose we wish to implement a 2-bit adder consisting of two full adders,



Writing this out using the specification equations above, we get the system of equations

$$
\begin{aligned}
s_0 &= \overline{a}\,\overline{b}\,c + \overline{a}\,b\,\overline{c} + a\,\overline{b}\,\overline{c} + a\,b\,c \\
c_1 &= \overline{a}\,b\,c + a\,\overline{b}\,c + a\,b\,\overline{c} + a\,b\,c \\
s_1 &= \overline{d}\,\overline{e}\,c + \overline{d}\,e\,\overline{c_1} + d\,\overline{e}\,\overline{c_1} + d\,e\,c_1 \\
c_2 &= \overline{d}\,e\,c_1 + d\,\overline{e}\,c_1 + d\,e\,\overline{c_1} + d\,e\,c_1 \\
s_2 &= c_2
\end{aligned}
$$

[3] Authors commonly abbreviate the acronym "*ROBDD*" to just "*BDD*."

Let us focus on the output s_2. Substituting for the variables defined in this system, we get

$$
\begin{aligned}
s_2 &= c_2 \\
&= \bar{d}\, e\, c_1 + d\, \bar{e}\, c_1 + d\, e\, \overline{c_1} + d\, e\, c_1 \\
&= \quad \bar{d}\, e\, (\bar{a}\, b\, c + a\, \bar{b}\, c + a\, b\, \bar{c} + a\, b\, c) \\
&\quad + \; d\, \bar{e}\, (\bar{a}\, b\, c + a\, \bar{b}\, c + a\, b\, \bar{c} + a\, b\, c) \\
&\quad + \; d\, e\, \overline{(\bar{a}\, b\, c + a\, \bar{b}\, c + a\, b\, \bar{c} + a\, b\, c)} \\
&\quad + \; d\, e\, (\bar{a}\, b\, c + a\, \bar{b}\, c + a\, b\, \bar{c} + a\, b\, c)
\end{aligned}
$$

The two ROBDDs for s_2 in Figure 6.4 illustrate that the size and shape of the DAG depends on the order in which the variables occur along a path. In the DAG on the left, the order is $\langle d, e, a, b, c\rangle$; and the the right, the order is $\langle e, b, d, a, c\rangle$.

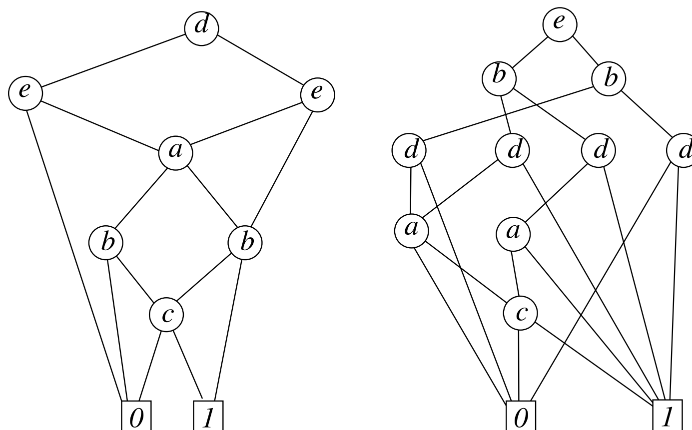


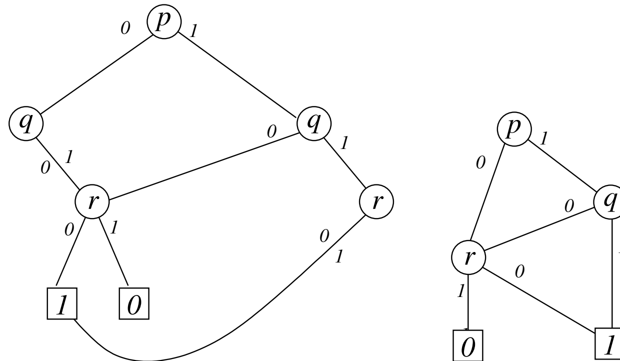
Figure 6.4: Two ROBDDs for the third sum bit of an adder

Exercises 6.7

1. In Section 6.7 it is claimed that the two graphs shown below are *not*

Copyright © 2008 Steven D. Johnson

homomorphic. Explain why this is true.



2. Draw two distinct ROBDDs for the output s_1 of the 2-bit adder.

3. Draw the ROBDD for output s_2 of the 2-bit adder under the variable order $\langle c, b, a, e, d \rangle$. [Hint. *Work top-down, simplifying as you go along. Suppose that a node labelled v represents a term Φ. $\Phi = \overline{v} \cdot \Phi_0 + v \cdot \Phi_1$, in which Φ_0 and Φ_1 are obtained by evaluating Φ with $v = 0$ and $v = 1$, repectively.*]

Chapter 7

Induction II

7.1 Introduction

The sets of interest in computer science tend to be complicated. For instance, consider the problem of defining the set of all legal C programs. Of course, the best way to think about such complicated objects is usually to to break them down into simpler pieces. For example, we build a C program out of constituent phrases, such as declarations, expressions, assignment statements, and so forth.

The decomposition of a complex set must be systematic. The pieces we break it into must be meaningful, so that we can reason about more complicated objects from facts about their pieces. We want definition methods and reasoning techniques that apply not just to a particular set of objects, but more broadly. We need a general scheme for the mathematical treatment of languages and what they mean. We begin with two examples to motivate these ideas. These examples will be used throughout this section and the next to illustrate the topics.

Example

Ex 7.1 The Language L

Suppose we wish to define a language, call it L, of simple multiplication expressions (You may want to review Section 1.2). Let V be a set of legal constant symbols and program variables and assume that the symbol '`*`' is not a member of V. L will consist of words over the alphabet $W = V \cup \{*\}$, such as `120`, `5*x`, and `width*height*5`.

Constants and variables themselves are simple expressions; and given two expressions u and v, one can build a new expression by concatenating, $u\hat{}*\hat{}v$. Now, if we wish to prove that some word $w \in W^+$ is in L, we should be able to provide a derivation of w according to these two rules:

A. Either $w \in V$, or

B. $w = u * v$ and both u and v have a derivation.

For example, we know that `width*height*5` is a word in L because

1.	`width` $\in L$	rule A
2.	`height` $\in L$	rule A
3.	`width*height` $\in L$	rule B with 1, 2
4.	`5` $\in L$	rule A
3.	`width*height*5` $\in L$	rule B with 3, 4

Notice that this is not the only possible derivation. Conversely, if we wish to prove that $w \notin L$ then we should show that no such derivation can exist.

Of course, L does not capture *all* the arithmetic expressions of a programming language. In a later section we will build a more realistic language. Let us now examine a claim about elements of L:

> **Claim** *The number of constants and variables contained in any word $w \in L$ is exactly one greater than the number of '`*`' symbols.*
>
> INFORMAL PROOF In the first place, each element of V has just one constant or variable and no '`*`'s. In the second place, if any two words u and v have this property, then so does the word u`*`v.

This argument is inductive! The hypothesis is:

$$H(w) \equiv \textit{The number of constants and variables in } w \textit{ is exactly one greater than the number of } \texttt{*}\textit{'s}$$

In a "base case" we proved $H(v)$ for all $v \in V$. In an "induction step" we prove $H(u)$ and $H(v)$ imply $H(u\texttt{*}v)$. However, the argument is not *technically* a mathematical induction because H is a predicate on L rather than $\mathbb{N}$.

Example

Ex 7.2 The Relation R

We are going to build a relation $R \subseteq \mathbb{N} \times \mathbb{N}$, starting with the ordered pair $(0, 0)$. The next pair in the relation is $(1, 2)$. In general, whenever we have a pair $(n, m) \in R$ we may add the pair $(n + 1, m + 2)$ to R, so we can think of R as the limit of a sequence of sets:

$$\begin{array}{rcl}
R_0 & = & \{(0,0)\} \\
R_1 & = & \{(0,0),(1,2)\} \\
R_2 & = & \{(0,0),(1,2),(2,4)\} \\
 & \vdots & \\
R_k & = & \{(0,0),(1,2),(2,4),\ \ldots,(k,2k)\} \\
 & \vdots & \\
R & = & \bigcup_{i\in\mathbb{N}} R_i = \{(0,0),(1,2),(2,4)\ \ldots,(k,2k),\ \ldots\}
\end{array}$$

We can see in each R_k the derivation of the pair $(k, 2k)$. We can also write down an explicit definition of R_k:

$$R_k = \{(x, 2x) \mid x \in \mathbb{N} \text{ and } x \leq k\}$$

The limit of this construction is

$$R = \{(x, 2x) \mid x \in \mathbb{N}\}$$

In fact, R happens to be a function, so not only can we build languages in this step-by-step way, but we can also define relations and functions. We will see much more of this technique later.

7.1.1 The Problem of Self Reference

Consider the riddle

> *In a certain restaurant there works a Waiter who serves every person that does not serve their self.*
>
> QUESTION: *Who serves the Waiter?*
>
> ANSWER: *There is no such restaurant.*

Let W stand for the supposed waiter. The logical problem in this riddle is that, were such a restaurant exist, it would be have to be the case that W *serves* W *iff* W *does not serve* W. Thus, the problem statement itself leads immediately to a contradiction.

The source of the problem may be that the assertion "W *serves* W" does not make sense. It seems to, but not all such self-referential statements do. Consider, for example, this proposition, known as the *Liar's Paradox*:

> *The sentence in this box is false.*

In order to talk about this sentence, it helps to give it a name, say S. Then we can write

$$S \equiv \text{"}S \text{ is false."}$$

S is a simple declaration of fact; it should be either *true* or *false*, but it cannot be either. The same is true of the "system" of two sentences,

$$\begin{array}{lcl} A & \equiv & \text{"Sentence } B \text{ is true."} \\ B & \equiv & \text{"Sentence } A \text{ is false."} \end{array}$$

There is no consistent truth assignment for A and B. Again, self-reference (direct or indirect) lies at the heart of the riddle.

Should we just outlaw self reference? This would be paying too high a price. Consider the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Taking $\mathtt{a}^n$ to mean a word consisting of n as, this language consists of all words containing

an equal number of as and bs, with all the as occuring before any of the bs. In a programmer's manual, languages are often described using a a "grammar," called Backus-Naur notation that looks like:

$$\langle L\rangle ::= \varepsilon \;[]\; \mathtt{a}\hat{\ }\langle L\rangle\hat{\ }\mathtt{b}$$

Translated to set notation, the notation says that the language Ł $\subseteq \{\mathtt{a},\mathtt{b}\}^*$ is

$$L = \{\varepsilon\} \cup \{\mathtt{a}\hat{\ }w\hat{\ }\mathtt{b} \mid w \in L\}$$

The language description is self-referencing, but certainly makes sense. In fact, describing languages this way is very useful.

When does self-reference make sense and when does it not? As we shall see in this chapter, some kinds of self-referencing inductive and recursive definitions are "sensible" and very useful.

Exercises 7.1

1. Which, if any, of the following statements are true?

$$\begin{array}{rcl} P & \equiv & \text{“Sentence } P \text{ is true or Sentence Q is true.”} \\ Q & \equiv & \text{“Sentences } P \text{ and } Q \text{ are not both true.”} \end{array}$$

7.2 Inductively Defined Sets

Let us look at what the motivating examples in the Section 7.1 have in common.

- Both examples involved a set U of values: words in $(V \cup \{*\})^+$ and pairs in $\mathbb{N} \times \mathbb{N}$, respectively.
- Both constructions started from a "seed" or *base set* from which all elements ultimately are built. For L the base set was V; and for R the base set was $\{(0,0)\}$.
- Finally, both examples employed a *constructor* function, to make more complicated elements from simpler ones. For L, there was a two-place constructor,

  $$f(u,v) = u\hat{\ }*\hat{\ }v$$

  For R, the constructor function was $g\colon \mathbb{N}^2 \to \mathbb{N}^2$,

  $$g(n,m) = (n+1, m+2)$$

  In general, such constructions may involve several different constructor functions of various ranks.

We would like to find properties that tell us exactly what these kinds of sets are, and how to reason about them. Part of the answer lies in the fact that sets like L and R are *closed* with respect to their constructors. This means that if you apply a constructor to elements of the set, the result is also in the set:

Copyright © 2008 Steven D. Johnson

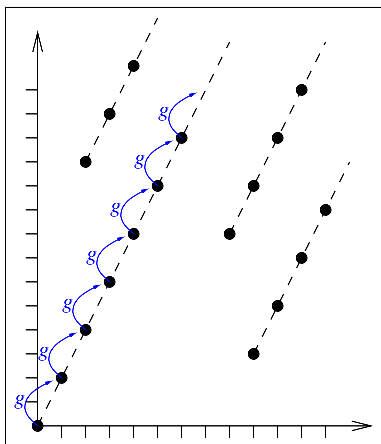
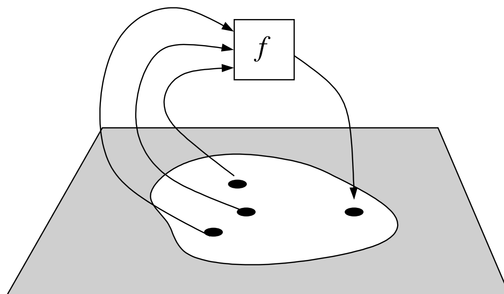


Figure 7.1: A typical subset of $\mathbb{N} \times \mathbb{N}$ containing base set $\{(0,0)\}$ and closed under the constructor function $g(n,m) = (n+1, m+2)$.

Definition 7.1 *If $f : U^r \to U$, and $S \subseteq U$, we say S is closed with respect to f iff $s_1, \ldots, s_r \in S$ implies that $f(s_1, s_2, \ldots, s_r) \in S$.*

Each constructor takes inputs from S to outputs in S.



The example sets L and R that we have been discussing are closed with respect to their constructor functions. If u and v are elements of L then so is the word $u*v$. And whenever $(n,m) \in R$, so is $(n+1, m+2)$. So closure with respect to the constructor functions is evidently a property of the sets we are building.

However, there may be many subsets of U closed under f. The empty set is closed with respect to every function, vaccuously. The entire set of values, U, is closed with respect to every constructor function, trivially. Hence, the next question to answer is whether we can uniquely determine which closed set a construction gives us.

Another property of the sets of interest is that they are generated by, and therefore contain, the base set. But again, there may be many subsets $S \subseteq U$ which are closed with respect to a constructor f. For example, Figure 7.2

shows the cartesian graph of a typical closed set for the function $g(n, m) = (n + 1, m + 2)$, the constructor function for the relation R. Any collection of half-lines with slope 2 (of course, we are referring only to the discrete points in $\mathbb{N} \times \mathbb{N}$ which lie on these lines) will be closed under g.

The figure also shows that points "generated" from the base point $(0, 0)$ lie on a single line. As the figure suggests, any closed subset that contains $\{(0, 0)\}$, must also contain the whole of R. In other words, R is the *smallest* closed subset which contains the base set. The property of being smallest determines R uniquely. Let us combine these observations into a definition.

Definition 7.2 *Let U be a set; let $B \subseteq U$. and let $f_1, \ldots, f_m$ be a collection of functions on U. A set A is said to be* inductively defined *from* base set B *and* constructors $f_1, \ldots, f_m$, *if*

(a) *A contains B*

(b) *A is closed with respect to each $f_1, \ldots, f_m$.*

(c) *if S is any subset of U that contains B and is closed with respect to every constructor, then $A \subseteq S$.*

It follows immediately from the definition that inductively defined sets are uniquely defined:

Proposition 7.1 *If A and A' are inductively defined from base set B and functions $f_1, \ldots, f_m$, then $A = A'$.*

Proof: By Definition 7.2(a, b), both sets contain B and are closed under the constructors. If we assume that both A and A' are inductively defined, then by 7.2(c), $A \subseteq A'$ and $A' \subseteq A$. Therefore $A = A'$.

∎

The next definition gives us a concise notation for specifying inductively defined sets.

Definition 7.3 (Inductive Set Definition Scheme) *The following language is used to specify an inductively defined set, A:*

1.	$B \subseteq A$	*[A contains the base set]*
2a.	$x_1, \ldots, x_r \in A$ $\Rightarrow f(x_1, \ldots, x_r) \in A$	*[A is closed under constructor f]*
	⋮	
3.	*nothing else is in A*	*[A is the smallest such set]*

Example

Ex 7.3 Let us use the definition scheme to specify the language of simple multiplication expressions from the beginning of this section. Let V be a set of constants and program variables. The language $L \subseteq (V \cup \{*\})^+$ is inductively defined according to:

1. $V \subseteq L$
2. $u, v \in L \Rightarrow u{*}v \in L$
3. nothing else

Example

Ex 7.4 The relation $R \subseteq \mathbb{N} \times \mathbb{N}$ from the beginning of this section is inductively defined according to.

1. $(0, 0) \in R$
2. $(n, m) \in R \Rightarrow (n + 1, m + 2) \in R$
3. n. e.

Example

Ex 7.5 We can define the *natural numbers* inductively, according to

1. $0 \in \mathbb{N}$
2. $k \in \mathbb{N} \Rightarrow k + 1 \in \mathbb{N}$
3. n. e.

As the exercises at the end of Section 7.4 suggest, our definition of inductive sets could be more general than it is. Of particular importance is the idea of simultaneously defining several sets inductively (see Exercise 5). We will see this kind of construction many times in later chapters. It is hard to come up with a *most* general set definition scheme.

7.3 The Principle of Structural Induction.

Let us attempt to use Definition 7.2 to show that the relation R of Example 7.4 is exactly the relation $\{(x, 2x) \mid x \in \mathbb{N}\}$, which we claimed but did not prove at the begining of the last section.

Proposition 7.2 *Let the relation R be defined as in Example 7.4. Define $E = \{(x, 2x) \mid x \in \mathbb{N}\}$. Then $R = E$*

Proof: We will prove that $E \subseteq R$ and $R \subseteq E$. Let g stand for R's constructor function:

$$g(n, m) = (n + 1, m + 2)$$

Copyright © 2008 Steven D. Johnson

Since

$$(0,0) = (0, 2 \cdot 0) \in E$$

and since $(x, 2x) \in E$ implies

$$g(x, 2x) = (x+1, 2x+2) = (x+1, 2(x+1)) \in E$$

Theorem 7.2 says that $E \subseteq R$ because R is the smallest set that contains $(0,0)$ and is closed under g.

But how do we know that $R \subseteq E$? The intuition is pretty clear:

1. R's base element, $(0,0)$, is in E as argued above.
2. a constructed element $(x+1, y+2)$ will be in E only if (x, y) is.
3. The only way to get an element of R is to build it from $(0,0)$

∎

This intuition, generalized, gives us a new Principle of Induction for inductively defined sets:

Theorem 7.3 (Principle of Structural Induction) *Let $A \subseteq U$ be inductively defined from base set B and functions $f_1, \ldots, f_m$. Suppose P is a predicate on U with the following properties:*

> (BASE CASE) *For all $x \in B$, $P(x)$ holds.*
>
> (INDUCTION STEP) *For each r-place constructor f,*
>
> $$P(x_1) \wedge \cdots \wedge P(x_r) \quad \Rightarrow \quad P(f(x_1, \ldots, x_r))$$
>
> *Then $P(x)$ holds <u>for all</u> $x \in A$,*

PROOF: The proof of the principle is given in the next section.

∎

Recall that we can think of $\mathbb{N}$ as a set defined inductively according to

1. $0 \in \mathbb{N}$
2. $k \in \mathbb{N} \Rightarrow (k+1) \in \mathbb{N}$
3. n. e.

The Principle of Structural Induction, applied to $\mathbb{N}$, says that if you can prove (1) $H(0)$ and (2) $H(k) \Rightarrow H(k+1)$, then you may conclude, *for all* $n \in \mathbb{N}, H(n)$. In other words, the Principle of Structural Induction reduces to ordinary mathematical induction in the case of $\mathbb{N}$

Copyright © 2008 Steven D. Johnson

Example

Ex 7.6 By using structural induction, we can distill the second half of Proposition 7.2 to its essence. Compare the argument below with the form of a mathematical induction.

> **Claim**. For E and R as defined in Proposition 7.2, $R \subseteq E$.
>
> PROOF: The proof is by structural induction on $x \in R$ with hypothesis $H(x) \equiv x \in E$.
>
> BASE CASE: $(0,0) = (0, 2 \cdot 0) \in E$
>
> INDUCTION STEP: Suppose $(n,m) \in E$. Then $(n,m) = (x, 2x)$ for some number x. Then
>
> $$g(n,m) = (n+1, m+2) = (x+1, 2x+2) = (x+1, 2(x+1)) \in E$$
>
> By Theorem 7.3 we may conclude, *for all* $x \in R,\ x \in E$. In other words, $R \subseteq E$.

∎

Example

Ex 7.7 At the beginning of the previous section a language of simple multiplication expressions was defined as follows. Let V be a set of program constants and program variables. Let $W = V \cup \{*\}$. The language $L \subseteq W^+$ is defined inductively according to:

1. $V \subseteq L$
2. $u, v \in L \Rightarrow u{*}v \in L$
3. n. e.

The following claim was made about elements of L:

> **Claim**: *The number of constants and variables contained in any word $w \in A$ is exactly one greater than the number of '$*$'s.*

It was observed that the argument proving this claim was "inductive." We can now recognize it as a structural induction:

> PROOF: The proof is by induction on $u \in L$. For the base case, an element of V has just one constant or variable and no '$*$'s. For the induction case, assume $u, v \in L$ each have this property. That is, u has n '$*$'s and $n+1$ symbols from V; and v has m '$*$'s and $m+1$ symbols from V. It follows, then, that the word $u{*}v$ has $n+m+1$ '$*$'s and $(n+1)+(m+1) = (n+m+1)+1$ symbols from V.

∎

Exercises 7.3

1. Let the set $A \subseteq \mathbb{N}$ be inductively defined according to

1. $1 \in A$
2. $k \in A \Rightarrow k + k \in A$
3. n. e.

Define the set $E = \{2^n \mid n \in \mathbb{N}\}$. *Prove*: $A = E$.

2. Let $V = \{$ `(` , `)` $\}$ and consider the set L of words in V^+ that is defined inductively according to.

1. `()` $\in L$

2a. $u \in L \Rightarrow$ `(` u `)` $\in L$

2b. $u, v \in L \Rightarrow$ `(` $u\,v$ `)` $\in L$

3. *nothing else*

Which of the following words are in L?

(a)	`((()))`	(d)	`(()())`
(b)	`()()`	(e)	`(()()())`
(c)	`((()())())`	(f)	`((()()))`

3. Let $V = \{$ `(` , `)` $\}$ and consider the set L of words in V^+ that is defined inductively according to.

1. `()` $\in L$

2a. $u \in L \Rightarrow$ `(` u `)` $\in L$

2b. $u, v \in L \Rightarrow$ `(` $u\,v$ `)` $\in L$

3. *nothing else*

Prove: *Every element in L has the same number of* `(` *'s and* `)` *'s.*

4. Think of another property that you can prove about words in the language L of Exercise 2.

7.4 Validity of the Induction Principle*

The goal in this section is to establish the truth of Theorem 7.3; that is, to establish rigorously and in general that inductive arguments like those of the previous section are valid. We shall do this by showing how structural induction is simply an encoded form of mathematical induction. Once we have proved the basic principle, we can safely go on to reason at a higher level.[1]

[1]If this were a college Calculus textbook, we would be at the beginning stages of using "delta-epsilon" arguments to prove the validity of certain laws of derivatives. Once the laws are proved valid, one safely can go about the real business of integrating functions. Similarly, we are now looking at *why* the methods used later on work.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

We characterized the elements of the language L and the relation R as having "derivations" from base elements. The next definition formalizes the notion of derivation as a *construction sequence*. The definition accounts for the general case in which there are several constructor functions.

Definition 7.4 *Let U be a set, B a subset of U, and $f_1, f_2, \ldots, f_m$ a collection of functions on U of various ranks. An element $a \in U$ is said to have a* construction sequence in U from B under $f_1, f_2, \ldots, f_m$ *if there exists a sequence of elements in U,*

$$\langle u_1, u_2, \ldots, u_n \rangle, \ n \geq 1$$

with the property that each u_i is either:

(a) *an element of B, or*

(b) *$u_i = f_j(a_1, \ldots, a_r)$, where f_j is an $r-$place function and each argument a_k occurs prior to u_i in the sequence.*

Example

Ex 7.8 In the language L defined in Example 7.1, we had the following construction sequence for the word `width*height*5`:

⟨`width`, `height`, `width*height`, `5`, `width*height*5`⟩

There are many other possible constructions sequences for the same word, for instance,

⟨`5`, `height`, `height*5`, `5`, `width`, `5*width`, `5`, `width*height*5`⟩

Check that both of these sequences are built according to rules (a) and (b) of Definition 7.4; so they are constructions sequences for `width*height*5`.

Example

Ex 7.9 In the relation R of Example 7.2, the pair $(5, 10)$ has a construction sequence

$$\langle (0,0),\ (1,2),\ (2,4),\ (3,6),\ (4,8),\ (5,10) \rangle$$

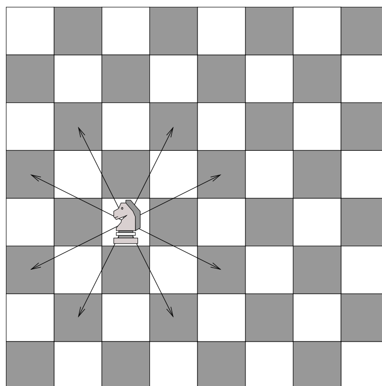
There are many possible construction sequences for $(5, 10)$ but this is the shortest one.

Example

Ex 7.10 Let $B = \{1, 2, \ldots, 8\}$ and let $U = B^2$. Define the following eight functions on U:

$$
\begin{aligned}
f_1(r,c) &= (r+2, c+1) \text{ if } r \leq 6 \text{ and } c \leq 7;\ (r,c) \text{ otherwise.}\\
f_2(r,c) &= (r+1, c+2) \text{ if } r \leq 7 \text{ and } c \leq 6;\ (r,c) \text{ otherwise.}\\
f_3(r,c) &= (r-1, c+2) \text{ if } r \geq 1 \text{ and } c \leq 6;\ (r,c) \text{ otherwise.}\\
f_4(r,c) &= (r-2, c+1) \text{ if } r \geq 2 \text{ and } c \leq 7;\ (r,c) \text{ otherwise.}\\
f_5(r,c) &= (r-2, c-1) \text{ if } r \geq 2 \text{ and } c \geq 1;\ (r,c) \text{ otherwise.}\\
f_6(r,c) &= (r-1, c-2) \text{ if } r \geq 1 \text{ and } c \geq 2;\ (r,c) \text{ otherwise.}\\
f_7(r,c) &= (r+1, c-2) \text{ if } r \leq 7 \text{ and } c \geq 2;\ (r,c) \text{ otherwise.}\\
f_8(r,c) &= (r+2, c-1) \text{ if } r \leq 6 \text{ and } c \geq 1;\ (r,c) \text{ otherwise.}
\end{aligned}
$$

Think of U as a chess board. These functions represent all the possible moves of a knight:



Now pick a square on the board, say $(3, 4)$. The construction sequences in U from $\{(3, 4)\}$ under $f_1, \ldots, f_8$ generate all the squares that a knight can reach starting from there.

Theorem 7.4 (Fundamental Theorem on Induction) *The set A of all elements of U that have a construction sequence from B under constructors f_1, $\ldots$, f_m is inductively defined from B and f_1, $\ldots$, f_m. That is, A is the smallest set containing B and closed under each of the constructors.*

PROOF: The proof has three parts. We must show that A contains B, that A is closed, and finally, that A is the smallest such set.
CLAIM I $B \subseteq A$
If $b \in B$ then by Definition 7.4, $\langle b \rangle$ is a construction sequence. Hence, $b \in A$ and so we have shown that $B \subseteq A$. This proves Claim I.
CLAIM II A is closed with respect to each constructor.
Suppose that $a_1, \ldots, a_r \in A$. Then each a_i has a construction sequence, $\langle u_{i1}, \ldots, a_i \rangle$. But then we can get a construction sequence for $f(a_1, \ldots, a_r)$ by putting all these sequences together end-to-end in any order:

$$\langle u_{11}, \ldots, a_1, u_{21}, \ldots, a_2, \ldots, u_{r1}, \ldots, a_r, f(a_1, \ldots, a_r) \rangle$$

Copyright © 2008 Steven D. Johnson

Therefore, $f(a_1, \ldots, a_r) \in A$ and we have shown that A is closed with respect to f. Since f was arbitrary, A is closed with respect to all the constructors. This proves Claim II.

CLAIM III If $S \subseteq U$ contains B and is closed with respect to each $f_1, \ldots, f_n$, then $A \subseteq S$.

To prove the claim, let S be as assumed. We want to show that $A \subseteq S$. By Definition 7.2, it suffices to show that any every element with a construction sequence must be in S. This is proven by induction on $k \in \mathbb{N}$ with hypothesis

$H(k) \equiv$ *Every element with a construction sequence of length k or less is in S*

BASE CASE: The base case holds vaccuously because there are no construction sequences of length 0.

INDUCTION STEP: Assume $H(k)$ and suppose that u has a construction sequence

$$d = \langle v_1, v_2, \ldots, v_k, u \rangle$$

By Definition 7.4, u is either an element of B or $u = f(a_1, \ldots, a_r)$ with each a_i occurring in d. In the first case, since $B \subseteq S$, we know that $u \in S$. In the second case we can, without loss of generality, assume that

$$d = \langle \ldots, a_1, \ldots, a_2, \ldots, a_r, \ldots, u \rangle$$

Then each a_i has a construction sequence,

$$d_i = \langle \ldots, a_1, \ldots, a_2, \ldots, a_i \rangle$$

By the induction hypothesis, this implies that each $a_i \in S$, and since S is closed with respect to f, $f(a_1, \ldots, a_r) = u \in S$. This concludes the induction step, the proof of Claim III, and the proof of the theorem.

∎

Theorem 7.4 shows us one way to construct inductively defined sets—and hence that such sets exist. Proposition 7.1 confirms that they are uniquely defined. Because of these results, we know that we can specify a set exactly, just by describing its base set and constructor function(s). We do not have to mention construction sequences, although it is sometimes useful to remember that they exist (for example, see Exercise 5 in Section 7.6).

As the exercises at the end of this section suggest, our definition of inductive sets could be more general than it is. Of particular importance is the idea of simultaneously defining several sets inductively (see Exercise 5). We will see this kind of construction many times in later chapters. The methods for determining the uniqueness of these sets is the same: they are determined by looking at their constructions sequences, or equivalently, by specifying their closure properties. It is hard to come up with a *most* general set definition scheme. On the other

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

hand, if a novel scheme is needed it is straightforward to prove a version of the Fundamental Theorem for it.

Construction sequences are one mathematical mechanism for building inductively defined sets, but there are others. Exercise 3 illustrates another way that is commonly seen, and asks you to prove that the two constructions are equivalent.

Our goal is to find a way to characterize inductively defined sets that is independent of the "mechanics" of how they are built. The notation scheme of Definition 7.3 gives a mechanim-independent way to define sets. The Principle of Structural Induction gives a mechanism-independent way to ask questions about them. We now prove the validity of the Principle.

Theorem 7.3 (restated) Let $A \subseteq U$ be inductively defined from base set B and functions $f_1, \ldots, f_m$. If $P \colon U \to \{T, F\}$ has the following properties:

(BASE CASE) For all $x \in B$, $P(x)$ holds.

(INDUCTION STEP) For each r-place constructor f,

$$P(x_1) \wedge \cdots \wedge P(x_r) \quad \Rightarrow \quad P(f(x_1, \ldots, x_r))$$

Then $P(x)$ holds for all $x \in A$,

PROOF: A is just the set of elements with construction sequences, so the proof is by induction on $k \in \mathbb{N}$ with hypothesis

$H(k) \equiv P$ *holds for every* $x \in U$ *with a construction sequence of length* k *or less.*

BASE CASE: $H(0)$ is true vaccuously.
INDUCTION STEP: Assume $H(k)$ and consider an element x with construction sequence $\langle a_1, a_2, \ldots, a_k, x\rangle$. According to Definition 7.4, either $x \in B$, in which case $P(x) = T$ by assumption (a), or $x = f_i(x_1, \ldots, x_r)$ and each x_i appears earlier in this construction sequence. But if x_i appears in the construction sequence, then x_i has a construction sequence of length k or less, so by the induction hypothesis, $P(a_i) = T$, for $1 \leq i \leq r$. Hence, by assumption (b), $P(x) = T$. This concludes the induction case and the proof of the theorem.

∎

Just as Theorem 7.4 allows us to define inductive sets without specifically saying how they might be built, Theorem 7.3 allows us to deduce properties about inductive sets without mentioning construction sequences. The theorem makes the underlying induction argument, once and for all.

Copyright © 2008 Steven D. Johnson

Example

Ex 7.11 Recall that in Proposition 7.2 we were to show the relation R, defined inductively from base set $\{(,0,0)\}$ and function

$$g(n,m) = (n+1, m+2),$$

is a subset of $E = \{(x, 2x) \mid x \in \mathbb{N}\}$.

Here is the proof, expressed as a structural induction.

Claim. For E and R as defined in Proposition 7.2, $R \subseteq E$.

PROOF: The proof is by structural induction on $x \in R$ with hypothesis

$$H(x) \equiv x \in E$$

BASE CASE: $(0,0) = (0, 2 \cdot 0) \in E$

INDUCTION STEP: Suppose $(n,m) \in E$. Then $(n,m) = (x, 2x)$ for some number x. Then

$$g(n,m) = (n+1, m+2) = (x+1, 2x+2) = (x+1, 2(x+1)) \in E$$

∎

Example

Ex 7.12 *To see what work is saved by the Principle of Structural Induction, let us "translate" the preceding argument into its underlying mathematical induction.*

PROOF: We prove $R \subseteq E$ by induction on $k \in \mathbb{N}$ with hypothesis,

$$H(k) \equiv \text{if } (n,m) \text{ has a construction of length } k \text{ or less, } (n,m) \in E.$$

BASE CASE: The only construction sequence of length one is $\langle(0,0)\rangle$ and $(0,0) \in E$ as has already been explained.

INDUCTION STEP: Assume $H(k)$, and suppose that (n,m) has a construction sequence of length $k+1$,

$$\langle(r_1, s_1), (r_2, s_2), \ldots, (r_k, s_k), (n,m)\rangle$$

According to Definition 7.4, either $(n,m) = (0,0)$, in which case we already know that $(n,m) \in E$, or $(n,m) = g(r_i, s_i) = (r_i + 1, s_i + 2)$ for some $i \leq k$. By the induction hypothesis, $(r_i, s_i) \in E$, so $s_i = 2r_i$. But then

$$(n,m) = (r_i + 1, 2r_i + 2) = (r_i + 1, 2(r_i + 1)) \in E$$

This concludes the induction.

Since R is just the set of all pairs with construction sequences, it follows that $R \subseteq E$.

∎

Copyright © 2008 Steven D. Johnson

Exercises 7.4

1. If the word "function" is replaced by the phrase "partial function" in Definition 7.4, does it change the validity of Theorem 7.4?

2. If the word "function" is replaced by the word "relation" in Definition 7.4, does it change the validity of Theorem 7.4?

3. Let $f: U \to V$ and $A \subseteq U$. Recall (Definition 6.4 that the *image* of A under f is defined to be

$$fA = \{f(x) \mid x \in A\}$$

The following is an alternative definition for *inductively defined set*, for a single constructor function (it can be generalized to many constructors).

> **Definition**. *Let U be a set, $B \subseteq U$, and $f: U \to U$. The set A is said to be* inductively defined by stages from B and f *if $A = \bigcup_{k\in\mathbb{N}} A_k$, where*
>
> $$\begin{aligned} A_0 &= B \\ A_1 &= A_0 \cup fA_0 \\ &\vdots \\ A_{k+1} &= A_k \cup fA_k \\ &\vdots \end{aligned}$$

Prove: A is inductively defined from B and f if and only if A is inductively defined by stages from B and f.

4. Given a relation $R \subseteq A \times A$, define the set $R^\star = \bigcup_{k\in\mathbb{N}} R_k$, where

$$\begin{aligned} R_0 &= R \\ &\vdots \\ R_{k+1} &= R_k \cup \{(x,z) \mid \exists y \in A\colon (x,y) \in R_k \text{ and } (y,z) \in R_k\} \\ &\vdots \end{aligned}$$

Prove that $R^\star$ is the smallest transitive relation that contains R. $R^\star$ is called the *transitive closure* of R.

5. Let U_1 and U_2 be sets, let $B_1 \subseteq U_1$ and $B_2 \subseteq U_2$; and let $f_1: U_1 \times U - 2 \to U_1$ and $f_2: U_1 \times U_2 \to U_2$. Let

> A_1 be the set of all elements of U_1 which have a construction sequence from $B_1 \cup B_1$ under f_1 and f_2

A_2 be the set of all elements of U_2 which have a construction sequence from $B_1 \cup B_1$ under f_1 and f_2

A_1 and A_2 are said to be *simultaneously inductively defined* Prove that A_1 and A_2 are the smallest of all sets S and T such that:

(a) $B_1 \subseteq S$ and $B_2 \subseteq T$

(b) For all $x \in S$ and $y \in T$, $f_1(x, y) \in S$ and $f_2(x, y) \in T$.

6. (tedious) Generalize the previous exercise to an arbitrary number of simultaneously defined sets under an arbitrary number of constructor functions (or relations).

7.5 Defining Functions with Recursion

Let $U = \mathbb{N} \times \mathbb{N}$ and define the relation $F \subseteq U^2$ inductively as follows.

1. $(0, 1) \in F$
2. $(n, m) \in F$ implies $(n + 1,\ m(n + 1)) \in F$
3. nothing else is in F

Let us build some construction sequences for F.

$$\begin{aligned}
d_0 &= \langle (0,1) \rangle \\
d_1 &= \langle (0,1),\ (1,1) \rangle \\
d_2 &= \langle (0,1),\ (1,1),\ (2,2) \rangle \\
d_3 &= \langle (0,1),\ (1,1),\ (2,2),\ (3,6) \rangle \\
d_4 &= \langle (0,1),\ (1,1),\ (2,2),\ (3,6),\ (4,24) \rangle \\
&\vdots \\
d_k &= \langle (0,1),\ (1,1),\ \ldots,\ (k,?) \rangle
\end{aligned}$$

Apparently, F is the factorial function. Let $G = \{(n, n!) \mid n \in \mathbb{N}\}$. Since G contains F's base element, $(0, 1) = (0, 0!)$; and since G is closed under F's constructor because

$$(n, n!) \quad \mapsto \quad (n + 1, n!(n + 1)) = (n + 1, (n + 1)!)$$

we know by Theorem 7.4 that $F \subseteq G$. To prove that $G \subseteq F$, use induction on $k \in \mathbb{N}$ with hypothesis, $H(k) \equiv (k, k!) \in F$. The details are left as an exercise.

Thus, we know that F is a function, and it is customary to write "$F(x) = y$" instead of "$(x, y) \in F$." If we express F's inductive definition in this way, we get:

1. $F(0) = 1$
2. $F(n) = m$ implies $F(n + 1) = m(n + 1)$
3. nothing else

Copyright © 2008 Steven D. Johnson

In part 2 the variable 'm' has become just a name for $F(n)$. By using $F(n)$ in place of m, we can further abbreviate the definition to

1. $F(0) = 1$
2. $F(n+1) = F(n) \cdot (n+1)$
3. nothing else

This is a *recursive* definition—F is defined "in terms of itself." It is not a circular definition because F's value for any number execpt 0 is in terms of values for a previous number.

Let us do another example, first in *relational form*, and then again in *functional form*.

Example

Ex 7.13 Consider the relation $G \subseteq \mathbb{N} \times \mathbb{N}$ defined inductively as follows:

1. $(0, 0) \in G$
2. $(x, y) \in G$ implies $(x+1,\ y+2x+1) \in G$
3. nothing else

Claim: $(x, y) \in G$ implies $y = x^2$.

PROOF: The proof is by induction on $k \in \mathbb{N}$ with hypothesis, $(k, k^2) \in G$.
BASE CASE: $(0, 0^2) = (0, 0) \in G$ by rule 1.
INDUCTION STEP: Assume that $(k, k^2) \in G$. Then by rule 2,

$$(k+1, x^2 + 2x + 1) = (x+1, (x+1)^2) \in G$$

∎

Example

Ex 7.14 The relation G in the previous exercise is a function, so let us repeat the previous argument, replacing "$(x, y) \in G$" by "$y = G(x)$." It is conventional in recursive definitions to leave out the nothing-else clause.

Consider the function $G\colon \mathbb{N} \to \mathbb{N}$ defined recusively as follows:

1. $G(0) = 0$
2. $G(x+1) = G(x) + 2x + 1$

CLAIM: For all $n \in \mathbb{N}$, $G(n) = n^2$.

PROOF: The proof is by induction on $k \in \mathbb{N}$ with hypothesis, $G(k) = k^2$.
BASE CASE: By the definition of G, part 1,

$$G(0) = 0 = 0^2$$

Copyright © 2008 Steven D. Johnson

INDUCTION STEP:

$$\begin{aligned} G(k+1) &= G(k) + 2k + 1 && \text{(G.2)} \\ &= k^2 + 2k + 1 && \text{(Induction Hypothesis)} \\ &= (k+1)^2 && \text{(factoring)} \end{aligned}$$

∎

7.6 Evaluation of Recursive Functions

Consider the relation $F \subseteq \mathbb{N}^2 \times \mathbb{N}$, defined inductively according to:

1. $((0,0),0) \in F$
2a. $((n,m),k) \in F$ implies $((n+1,m),k+1) \in F$
2b. $((n,m),k) \in F$ implies $((n,m+1),k+2) \in F$
3. n. e.

Now suppose we want to find a value ℓ for which $((2,3),\ell) \in F$. In relational form, we can build a "bottom-up" derivation: starting from the base rule:

	(*rule*)
$\langle\ ((0,0),0)$	(1)
$((0,1),2)$	(2b)
$((1,1),3)$	(2a)
$((2,1),4)$	(2a)
$((2,2),6)$	(2b)
$((2,3),8)\rangle$	(2b)

This is not the only way to build the result, and you may wish to convince yourself that $((2,3),8)$ is the only possible value.

F is a function and the functional form of its definition is:

1. $F(0,0) = 0$
2a. $F(n+1,m) = F(n,m) + 1$
2b. $F(n,m+1) = F(n,m) + 2$

In this form, a derivation of F's value at $(2,3)$ proceeds "top down," for instance,

$$\begin{aligned} F(2,3) &= \underline{F(2,2)} + 2 && \text{(F.2b)} \\ &= (\underline{F(2,1)} + 2) + 2 && \text{(F.2b)} \\ &= ((\underline{F(1,1)} + 1) + 2) + 2 && \text{(F.2a)} \\ &= (((\underline{F(0,1)} + 1) + 1) + 2) + 2 && \text{(F.2a)} \\ &= ((((\underline{F(0,0)} + 2) + 1) + 1) + 2) + 2 && \text{(F.2b)} \\ &= ((((0 + 2) + 1) + 1) + 2) + 2 && \text{(F.1)} \\ &= 8 && \text{(arithmetic)} \end{aligned}$$

In the relational form, the computation strictly follows the inductive definition of F. The functional form, on the other hand, appears as a set of algebraic identities that describe F; and to evaluate F on an argument, we apply whatever identities we can until we reach an answer. This form may seem more natural—it is closer to our idea of a *computation*—but this intuition it can sometimes lead us into difficulty, as we shall see in the next chapter.

There is a third alternative available to us for this example. It is to observe, perhaps by experimenting with examples, that $F(n, m) = n+2m$. This is easily proved by structural induction on F. In the base case, $F(0, 0) = 0 = 0 + 2 \cdot 0$. There are two induction cases,

$$F(n+1, m) \overset{2a}{=} F(n, m) + 1 \overset{IH}{=} (n + 2m) + 1 = (n + 1) + 2m$$

and

$$F(n, m+1) \overset{2b}{=} F(n, m) + 2 \overset{IH}{=} (n + 2m) + 2 = n + 2(m + 1)$$

Exercises 7.6

1. Covert the following definiton of relation $F \subseteq \mathbb{N}^2 \times \mathbb{N}$ into functional form:

1.	for all y, $((0, y), 0) \in F$
2.	$((x, y), k) \in F \Rightarrow ((x + 1, y), k + y) \in F$
3.	nothing else

2. Covert the following definiton of relation $F \subseteq \mathbb{N}^2 \times \mathbb{N}$ into functional form:

1.	for all x, $((x, 0), x) \in G$
2a.	for all y, $((0, y), 0) \in G$
2b.	$((x, y), k) \in F \Rightarrow ((x + 1, y + 1, k) \in G$
3.	nothing else

3. For F and G defined above, compute in relational form

(a) $F(2, 3)$

(b) $G(5, 2)$

(c) $G(2, 4)$

4. Repeat Exercise 3 using the functional forms from Exercises 1 and 2.

5. Consider the following functional definition for $F : \mathbb{N} \to \mathbb{N}$:

1.	if $x > 100$, then $F(x) = x - 10$
2.	if $x \leq 100$, then $F(x) = F(F(x + 11))$

Compute $F(97)$ using functional form.

Copyright © 2008 Steven D. Johnson

6. Consider the set $Num \subseteq \{\mathtt{0},\mathtt{1}\}^+$ defined as follows:

1. $\mathtt{1} \in Num$
2a. if $x \in Num$, then $x\mathtt{0} \in Num$
2b. if $x \in Num$, then $x\mathtt{1} \in Num$
3. nothing else

Num is clearly the set of binary numerals with a leading `1`. In either functional or relational form, define a function $B\colon Num \to \mathbb{N}$ that sends each numeral to the integer it represents.

7. n your favorite programming language, write a program that reads a sequence of `0`s and `1`s and translates the sequence into an integer.

8. Let $F \subseteq \mathbb{N} \times \mathbb{N}$ be defined as follows:

1. $(0, 2) \in F$
2. $(x, y) \in F \Rightarrow (x + 1, y + 3) \in F$
3. nothing else

Prove that $f = \{(x, 3x + 2) \mid x \in \mathbb{N}\}$.

9. (Hard) Consider the function F defined in Exercise 5. Prove that if $x \leq 100$, then $F(x) = 91$. (*Hint:* Translate into relational form and use induction on construction sequences with the indcution hypothesis: "If (x, y) has a construction sequence with length $j \leq k$, then ....")

10. Define the *lexicographic ordering* of $\mathbb{N} \times \mathbb{N}$, denoted by '$\preceq$', as follows: $(n, m) \preceq (k, \ell)$ iff $(n < k)$ or both $(n \leq k)$ and $(m \leq \ell)$). A *Principle of Lexicographic Induction* for predicate H on $\mathbb{N}^2$ is:

> *If you can prove that*
>
> BASE CASE *$P(0, 0)$ holds.*
>
> INDUCTION CASE *If $H(i, j)$ holds for all $(j, j) \preceq (k, \ell)$ then $H(k, \ell)$ also holds.*
>
> *Then you can conclude that H holds for all for all $(n, m) \in \mathbb{N}^2$*

Prove that this induction principle is valid.

11. At the end of this section the function $F\colon \mathbb{N}^2 \to \mathbb{N}$ was recursively defined according to:

1. $F(0, 0) = 0$
2a. $F(n + 1, m) = F(n, m) + 1$
2b. $F(n, m + 1) = F(n, m) + 2$

It was proved by structural induction on F (considered as a set) that for all pairs (n, m) *in the domain of* F, $F(n, m) = n + 2m$. Use lexicographic induction (see Exercise 10 to prove that for all $(n, m) \in \mathbb{N}^2$, $F(n, m) = n + 2m$.

7.7 Reasoning about Recursive Functions

The primary method for reasoning about recursive functions is structural induction. We look at a number of examples dealing with the words of the following simple language.

For alphabet $V = \{\mathtt{a}, \mathtt{b}, \bullet\}$, define the language $L \subseteq V^+$ inductively, according to

1.	$\bullet \in L$
2a.	$u \in L \Rightarrow \mathtt{a}u \in L$
2b.	$u \in L \Rightarrow \mathtt{b}u \in L$
3.	nothing else

L consists of words over $\{\mathtt{a}, \mathtt{b}\}$ to which the symbol '$\bullet$' has been appended on the right:

$$L = \{\bullet, \mathtt{a}\bullet, \mathtt{b}\bullet, \mathtt{aa}\bullet, \mathtt{ab}\bullet, \mathtt{ba}\bullet, \mathtt{bb}\bullet, \mathtt{aaa}\bullet, \ldots\}$$

Let P be any predicate on L. Since it has a single-element base set and two constructor functions, a proof by structural induction on L has the following form.

Theorem. For all $w \in L$, $H(w)$.

PROOF: The proof is by induction on $u \in L$ with hypothesis $H(u)$

BASE CASE: *A direct proof of* $P(\bullet)$.

INDUCTION STEP: *Proofs that*

$$P(u) \Rightarrow P(\mathtt{a}u)$$

and

$$P(u) \Rightarrow P(\mathtt{b}u)$$

Often, the two arguments are so similar that only one is shown.

∎

We shall define three functions on the language L.

Invert The function $I : L \to L$ changes all '$\mathtt{a}$'s in a word to '$\mathtt{b}$'s and all '$\mathtt{b}$'s to '$\mathtt{a}$'s. I is defined recursively according to

1.	$I(\bullet) = \bullet$
2a.	$I(\mathtt{a}u) = \mathtt{b}I(u)$
2b.	$I(\mathtt{b}u) = \mathtt{a}I(u)$

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

For example $I(\texttt{baab}\bullet) = \texttt{abba}\bullet$ because:

$$\begin{array}{ll} I(\texttt{baab}\bullet) & \\ \quad = \texttt{a}I(\texttt{aab}\bullet) & \text{(I.2b)} \\ \quad = \texttt{ab}I(\texttt{ab}\bullet) & \text{(I.2a)} \\ \quad = \texttt{abb}I(\texttt{b}\bullet) & \text{(I.2a)} \\ \quad = \texttt{abba}I(\bullet) & \text{(I.2b)} \\ \quad = \texttt{abba}\bullet & \text{(I.1)} \end{array}$$

Append The function $A\colon L \times L \to L$ joins two words together. A is defined recursively according to

$$\begin{array}{rl} 1. & A(\bullet, u) = u \\ 2a. & A(\texttt{a}u, v) = \texttt{a}A(u, v) \\ 2b. & A(\texttt{b}u, v) = \texttt{b}A(u, v) \end{array}$$

A is like a concatenation operation, but it throws away the '$\bullet$' symbol between the words. For example,

$$A(\texttt{aa}\bullet, \texttt{bba}\bullet) \stackrel{2a}{=} \texttt{a}A(\texttt{a}\bullet, \texttt{bba}\bullet) \stackrel{2a}{=} \texttt{aa}A(\bullet, \texttt{bba}\bullet) \stackrel{1}{=} \texttt{aabba}\bullet$$

Reverse The function $R\colon L \to L$ reverses the order of '$\texttt{a}$'s and '$\texttt{b}$'s in a word. Reverse uses Append. R is defined recursively according to

$$\begin{array}{rl} 1. & R(\bullet) = \bullet \\ 2a. & R(\texttt{a}u) = A(R(u), \texttt{a}\bullet) \\ 2b. & R(\texttt{b}u) = A(R(u), \texttt{b}\bullet) \end{array}$$

We shall now prove a series of facts about the three functions just defined.

Proposition 7.5 *For all $u \in L$, $I(I(u)) = u$.*

PROOF: The proof is by induction on $u \in L$ In the base case, by the defintion of I, rule 1,

$$I(I(\bullet)) = I(\bullet) = \bullet$$

For the induction step,

$$\begin{array}{rll} I(I(\texttt{a}u)) & = I(\texttt{b}I(u)) & \text{(I.2a)} \\ & = \texttt{a}I(I(u)) & \text{(I.2b)} \\ & = \texttt{a}u & \text{(I.H.)} \end{array}$$

Similarly,

$$I(I(\texttt{b}u)) \stackrel{2b}{=} I(\texttt{a}I(u)) \stackrel{2a}{=} \texttt{b}I(I(u)) \stackrel{IH}{=} \texttt{b}u$$

∎

From now on, the second proof of the induction case will not be shown unless it differs significantly from the first.

Proposition 7.6 *I distributes over A, that is, for all* $u, v \in L$, $I(A(u,v)) = A(I(u), I(v))$.

PROOF: The proof is by induction on $u \in L$. In the base case,

$$I(A(\bullet, v)) \overset{A.1}{=} I(v) \overset{A.1}{=} A(\bullet, I(v)) \overset{I.1}{=} A(I(\bullet), I(v))$$

For the induction step,

$$\begin{aligned} I(A(\mathtt{a}u, v)) &= I(\mathtt{a}A(u,v)) && \text{(A.2a)} \\ &= \mathtt{b}I(A(u,v)) && \text{(I.2a)} \\ &= \mathtt{b}A(I(u), I(v)) && \text{(I. H.)} \\ &= A(\mathtt{b}I(u), I(v)) && \text{(A.2b)} \\ &= A(I(\mathtt{b}u), I(v)) && \text{(I.2b)} \end{aligned}$$

The proof that $I(A(\mathtt{b}u, v)) = A(I(\mathtt{b}u), I(v))$ is similar.

∎

In this proposition there is a choice of two variables on which to perform the induction. The phrase "by induction on u," signals the choice. The induction hypothesis becomes,

$$H(u) \equiv \textit{for all } v \in L,\ I(A(u,v)) = A(I(u), I(v))$$

That is, the for-all quantifier on v is retrained. In this case, u is the appropriate choice because only the first argument varies in the definiiton of A. In the next proposition, we have three variables to choose from.

Proposition 7.7 *A is associative, that is, for all* $u, v,$ and w in L

$$A(u, A(v,w)) = A(A(u,v), w)$$

PROOF: The proof is by induction on $u \in L$.
BASE CASE:

$$A(\bullet, A(v,w)) \overset{A.1}{=} A(v,w) \overset{A.1}{=} A(A(\bullet, v), w)$$

INDUCTION STEP:

$$\begin{aligned} A(\mathtt{a}u, A(v,w)) &= \mathtt{a}A(u, A(v,w)) && \text{(A.2a)} \\ &= \mathtt{a}A(A(u,v), w) && \text{(I. H.)} \\ &= A(\mathtt{a}A(u,v), w) && \text{(A.2a)} \\ &= A(A(\mathtt{a}u, v), w) && \text{(A.2a)} \end{aligned}$$

The proof that $A(\mathtt{b}u, A(v,w)) = A(A(\mathtt{b}u, v), w)$ is similar.

∎

Proposition 7.8 *$\bullet$ is a right identity for A, that is, for all $u \in L$ $A(u, \bullet) = u$.*

PROOF: Exercise 3.

∎

Proposition 7.9 *R and A obey the following distributive law: for all $u, v \in L$,*

$$R(A(u, v)) = A(R(v), R(u))$$

[Note how the positions of u and v are switched.]

PROOF: Exercise 4.

∎

Proposition 7.10 *R is self cancelling, that is, for all $u \in L$ $R(R(u)) = u$.*

PROOF: The proof is by induction on u.
BASE CASE: by the definition of R,

$$R(R(\bullet)) = R(\bullet) = \bullet$$

INDUCTION STEP:

$$\begin{aligned}
R(R(\mathtt{a}u)) &= R(A(R(u), \mathtt{a}\bullet)) && \text{(A.2a)}\\
&= A(R(\mathtt{a}\bullet), R(R(u))) && \text{(Proposition 7.9)}\\
&= A(R(\mathtt{a}\bullet), u) && \text{(I. H.)}\\
&= A(A(R(\bullet), \mathtt{a}\bullet), u) && \text{(R(2a)}\\
&= A(A(\bullet, \mathtt{a}\bullet), u) && \text{(R.1)}\\
&= A(\mathtt{a}\bullet, u) && \text{(A.1)}\\
&= \mathtt{a}A(\bullet, u) && \text{(A.2a)}\\
&= \mathtt{a}u && \text{(A.1)}
\end{aligned}$$

The proof that $R(R(\mathtt{b}u)) = \mathtt{b}u$ is similar.

∎

Perhaps you have noticed that in each of the proofs of this section, the necessary facts had conveniently been established by previous propositions. Proposition 7.10 uses the result of Proposition 7.9, which in turn needs the results of Propositions 7.7 and 7.8. The process of *conceiving* a proof typically works in the other direction. It is a goal directed activity just like programming.

In attempting to prove Proposition 7.10, one gets stuck at the second line of the induction step, discovering there that some kind of distributive law is needed. Notice, however, that the distributive law proved in Proposition 7.9 is more general than what is needed for Proposition 7.10. It is usually a good idea to prove a more general fact if you can; most cases it is easier, and the result you have proven may also be applicable to other problems.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Let us define a new function, $T\colon L^2 \to L$, according to

$$\begin{array}{rl} 1. & T(\bullet, v) = v \\ 2a. & T(\mathtt{a}u, v) = T(u, \mathtt{a}v) \\ 2b. & T(\mathtt{b}u, v) = T(u, \mathtt{b}v) \end{array}$$

If you evaluate a few examples (Exercise 5) you may discover that T is related to both A and R. In particular, T is equivalent to R when its second argument is $\bullet$:

Proposition 7.11 *For all $u \in L$, $T(u, \bullet) = R(u)$.*

Proof: Exercise 7.

∎

The function R might be considered a more natural description of "reversing a word." The function T might be considered a better description of the reversing *computation*. It is easier to prove, for example, that R is self-cancelling and distributes over A than it is to prove these facts about T. On the other hand, there is a definite sense in which T seems more efficient—to see this, evaluate $R(\mathtt{aab}\bullet)$ and $T(\mathtt{aab}\bullet, \bullet)$, according to their definitions, each time counting the number of concatenations performed. T is a degenerate form of recursion called *iteration*. It translates to the looping construct of a programming language. Consider the program shown in Figure 7.7 Can you guess the invariant of this loop (recall Theorem 4.2)? In Chapter 10 we will take a closer and more formal look at this relationship between recursive functions and programs.

Exercises 7.7

1. Use the definitions of A and R to compute the values of

(a) $R(\mathtt{abb}\bullet)$

(b) $R(A(\mathtt{a}\bullet, \mathtt{b}\bullet))$

2. Use the definitions of A, R, and T to compute the values of

(a) $R(\mathtt{aab}\bullet)$

(b) $T(\mathtt{aab}\bullet, \bullet)$

(c) $A(R(\mathtt{a}\bullet), \mathtt{b}\bullet)$

(d) $T(\mathtt{a}\bullet, \mathtt{b}\bullet)$

3. Prove Proposition 7.8.

4. Prove Proposition 7.9.

Copyright © 2008 Steven D. Johnson

5. Evaluate the following, using the definitions of I, A, R, and T in this section.

(a)	$I(\mathtt{abbab}\bullet)$	(b)	$I(I(\mathtt{abb}\bullet))$
(c)	$A(\mathtt{aa}\bullet, \mathtt{ba}\bullet)$	(d)	$A(I(\mathtt{ab}\bullet), I(\bullet))$
(e)	$R(\mathtt{abb}\bullet)$	(f)	$R(I(\mathtt{ab}\bullet))$
(g)	$T(\mathtt{baa}\bullet, \mathtt{ab}\bullet)$	(h)	$T(\mathtt{babb}\bullet, \bullet)$

6. Without referring to the proof in text book, try to prove Proposition 7.9, developing auxiliary propositions as the need arises.

7. Prove Proposition 7.11. [*Hint: You will need to prove a more general fact.*]

8. For the following problems, consider the language $N \subseteq \{\mathtt{S}, \bullet\}$, defined inductively as follows:

1. $\bullet \in N$
2. $u \in N$ implies $\mathtt{S}u \in N$
3. nothing else

Define the function $P\colon N^2 \to N$ recursively, according to:

1. $P(\bullet, v) = v$
2. $P(\mathtt{S}u, v) = \mathtt{S}P(u, v)$

Define the function $M\colon N^2 \to N$ recursively, according to:

1. $M(\bullet, v) = \bullet$
2. $M(\mathtt{S}u, v) = P(v, M(u, v))$

[*Hint: It may be helpful to express these results using infix notation. It will be helpful to have an idea in mind of what these functions represent.*]

(a) Prove that M distributes over P; that is, for all $u, v, w \in N$, $M(u, P(v, w)) = P(M(u, v), M(u, w))$.

(b) Prove that P is *commutative*; that is, For all $u, v \in N$, $P(u, v) = P(v, u)$.

(c) Prove: For all $u, v \in N$, $M(u, \mathtt{S}v) = P(u, M(u, v))$.

(d) Prove: For all $u, v \in N$, $P(\mathtt{S}u, v) = P(u, \mathtt{S}v)$.

(e) Prove: For all $u \in N$, $P(u, \bullet) = u$.

Copyright © 2008 Steven D. Johnson

```
T: begin
   u := W;
   v := •;
   while u ≠ • do { what goes here? }
      begin
      if  FirstLetter(u) = a
         then begin
               v := av;
               u := AllButFirstLetter(u)
               end
         else begin
               v := bv;
               u := AllButFirstLetter(u)
               end
      end
   {v = R(W)}
```

Figure 7.2: A program to compute function R

Chapter 8

Languages and Meanings

In this chapter we look at how programming languages are defined. There are two aspects to consider. We must develop a method to specify exactly what words are valid as program expressions. And we must develop a method to say precisely what computation a valid program expresses. We now have the basic mathematical tools to make these specifications: programming languages are specified by inductive set definitions and their meanings by recursive function definitions. However, we still must develop techniques to use these tools effectively.

8.1 Language Definitions

We shall start with a seemingly simple language of expressions. The idea is to put the language together by showing how to build more complicated expressions from simpler ones. This is the way almost all computer languages are defined.

Let the set $A = \mathbb{N} \cup \{ \texttt{\$}, \texttt{\#} \}$ be our alphabet of symbols. Define the language $L \subseteq A^+$ inductively, according to

1.	$\mathbb{N} \subseteq L$
2a.	$u, v \in L \Rightarrow u \texttt{ \$ } v \in L$
2b.	$u, v \in L \Rightarrow u \texttt{ \# } v \in L$
3.	nothing else

The first kind of question that might be asked is whether a particular word in A^+ is in (i.e. an element of) the language L. To answer such a question, we must analyze the given word to see whether it could be built using the rules of L's definition.

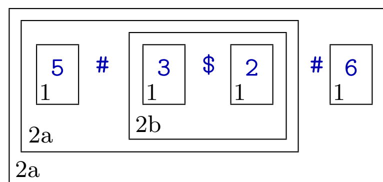
Example

Ex 8.1 *Is the word* $5 \texttt{ \# } 3 \texttt{ \$ } 2 \texttt{ \# } 6$ *in* L*?*

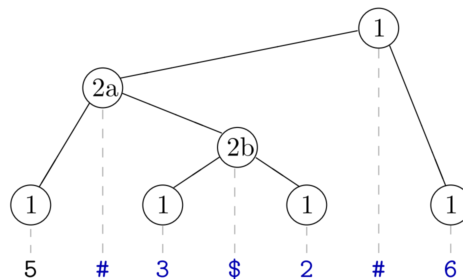
The answer is “yes,” because there is a construction sequence:

1. $3 \in L$ by rule 1
2. $2 \in L$ by rule 1
3. $3 \$ 2 \in L$ by rule 2b, 1 and 2
4. $5 \in L$ by rule 1
5. $5 \# 3 \$ 2 \in L$ by rule 2a, 4 and 3
6. $6 \in L$ by rule 1
7. $5 \# 3 \$ 2 \# 6 \in L$ by rule 2a, 5 and 6

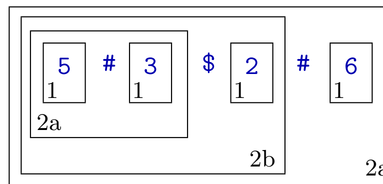
The analysis showing that a word is in a language is called *parsing*. There are two ways to diagram derivations like this. A *parsing diagram* uses nested boxes to show how the word decomposes:



The second way is to draw a A *parse tree*, whose interior nodes are labeled by rules and whose leaves are symbols of the alphabet:



Each of the diagrams above reflects the same parse. Here is a different parse showing that $5 \# 3 \$ 2 \# 6$ is in L:



That there are several ways to prove that a given word is in L is a problem, as we shall see next.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

8.2 Defining How Languages are Interpreted

The interpretation of a language associates a value with each word. The word is said to express the value. If the language is inductively defined, then the interpretation can be defined recursively.

Using the language L of the previous section, let us define a function $\mathcal{V}\colon L \to \mathbb{N}$, which gives a natural-number interpretation where symbols # and $ express addition and multiplication, respectively.

1.	for $k \in \mathbb{N}$,	$\mathcal{V}[k]$	$= k$
2a.	for words of the form $w = u \mathbin{\#} v$,	$\mathcal{V}[w]$	$= \mathcal{V}[u] + \mathcal{V}[v]$
2b.	for words of the form $w = u \mathbin{\$} v$,	$\mathcal{V}[w]$	$= \mathcal{V}[u] \times \mathcal{V}[v]$

Based on this definition and the first parsing analysis, we can determine an interpretation of $5 \mathbin{\#} 3 \mathbin{\$} 2 \mathbin{\#} 6$:

1.	$3 \in \mathbb{N}$	$\Rightarrow \mathcal{V}[3] = 3$
2.	$2 \in \mathbb{N}$	$\Rightarrow \mathcal{V}[2] = 2$
3.	$\left.\begin{array}{rcl} u &=& 3 \in L \\ v &=& 2 \in L \end{array}\right\}$	$\Rightarrow \mathcal{V}[u \mathbin{\$} v] = \mathcal{V}[u] \times \mathcal{V}[v] = 3 \times 2 = 6$
4.	$5 \in \mathbb{N}$	$\Rightarrow \mathcal{V}[5] = 5$
5.	$\left.\begin{array}{rcl} u &=& 5 \in L \\ v &=& 3 \mathbin{\$} 2 \in L \end{array}\right\}$	$\Rightarrow \mathcal{V}[u \mathbin{\#} v] = \mathcal{V}[u] + \mathcal{V}[v] = 5 + 6 = 11$
6.	$6 \in \mathbb{N}$	$\Rightarrow \mathcal{V}[6] = 6$
7.	$\left.\begin{array}{rcl} u &=& 5 \mathbin{\#} 3 \mathbin{\$} 2 \in L \\ v &=& 6 \in L \end{array}\right\}$	$\Rightarrow \mathcal{V}[u \mathbin{\#} v] = \mathcal{V}[u] + \mathcal{V}[v] = 11 + 6 = 17$

That is, $\mathcal{V}[5 \mathbin{\#} 3 \mathbin{\$} 2 \mathbin{\#} 6] = 17$. However, a different parse leads to a different interpretation. The derivation that follows is "top-down" in the sense that the interpreted expression is decomposed as the interpretation is applied. At each step, you should verify that the word is broken down in a manner that is consistent with L's definition.

$$
\begin{array}{lll}
\mathcal{V}[5 \mathbin{\#} 3 \mathbin{\$} 2 \mathbin{\#} 6] & & \\
\quad = & \mathcal{V}[5] + \mathcal{V}[3 \mathbin{\$} 2 \mathbin{\#} 6] & \text{defn. } \mathcal{V}\text{, case 2a} \\
\quad = & 5 + \mathcal{V}[3 \mathbin{\$} 2 \mathbin{\#} 6] & \mathcal{V}(1) \\
\quad = & 5 + \big(\mathcal{V}[3] \times \mathcal{V}[2 \mathbin{\#} 6]\big) & \mathcal{V}(2b) \\
\quad = & 5 + \big(3 \times \mathcal{V}[2 \mathbin{\#} 6]\big) & \mathcal{V}(1) \\
\quad = & 5 + \big(3 \times \big(\mathcal{V}[2] + \mathcal{V}[6]\big)\big) & \mathcal{V}(2a) \\
\quad = & 5 + \big(3 \times \big(2 + 6\big)\big) & \mathcal{V}(1)\text{, twice} \\
\quad = & 29 & \text{arithmetic}
\end{array}
$$

That is, $\mathcal{V}[5 \,\#\, 3 \,\$\, 2 \,\#\, 6] = 17 = 29$ (?!). The apparent contradiction is due to the assumption that $\mathcal{V}$ is a function; use of the '=' symbol, in defining $\mathcal{V}$ and in deriving a value, is simply wrong. Both interpretations are correct: $\mathcal{V}$ is a *relation* associating the values 17, 29, and several others, with the word $5 \,\#\, 3 \,\$\, 2 \,\#\, 6$.

When computer languages are defined, it is usually intended that each word have a unique interpretation—we *want* $\mathcal{V}$ to be a function. When multiple interpretations exist, it is said that the language is *ambiguous*. As we shall see later, ambiguity is not necessarily the fault of the language, but even so, languages can be designed to be unambiguous. In the following two examples, variations of L are defined, by which the problems just encountered are avoided.

Example

Ex 8.2 We can remove the ambiguity in L by introducing parenthesis symbols. Take $V = \mathbb{N} \cup \{\#, \$, (,)\}$. The language L_2 and its interpretation relation $\mathcal{V}_2$ are defined simultaneously below:

	$L_2 \subseteq V^+$	$\mathcal{V}_2 : L_2 \to \mathbb{N}$
1.	$\mathbb{N} \subseteq L_2$	$\mathcal{V}_2[k] = k,$ for $k \in \mathbb{N}$
2a.	$u, v \in L_2 \Rightarrow (u \,\#\, v) \in L_2$	$\mathcal{V}_2[(u \,\#\, v)] = \mathcal{V}_2[u] + \mathcal{V}_2[v]$
2b.	$u, v \in L_2 \Rightarrow (u \,\$\, v) \in L_2$	$\mathcal{V}_2[(u \,\$\, v)] = \mathcal{V}_2[u] \times \mathcal{V}_2[v]$
3.	nothing else	

In this language,

$$\mathcal{V}_2[((5 \,\#\, (3 \,\$\, 2)) \,\#\, 6)/] = 17$$

and

$$\mathcal{V}_2[(5 \,\#\, (3 \,\$\, (2 \,\#\, 6)))] = 29.$$

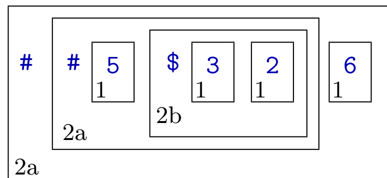
This version of L_2 forces every expression to be fully parenthesized, but the interpretation is unambiguous. $\mathcal{V}_2$ is a function because there is only one way to parse any word in L_2.

Example

Ex 8.3 In the version of L shown below, there are no parentheses but the operator symbols have been moved from an infix position to a *prefix* position:

	$L_3 \subseteq V^+$	$\mathcal{V}_3 : L_3 \to \mathbb{N}$
1.	$\mathbb{N} \subseteq L_3$	$\mathcal{V}_3[k] = k,$ for $k \in \mathbb{N}$
2a.	$u, v \in L_3 \Rightarrow \# \, u \, v \in L_3$	$\mathcal{V}_3[\# \, u \, v] = \mathcal{V}_3[u] + \mathcal{V}_3[v]$
2b.	$u, v \in L_3 \Rightarrow \$ \, u \, v \in L_3$	$\mathcal{V}_3[\$ \, u \, v] = \mathcal{V}_3[u] \times \mathcal{V}_3[v]$
3.	nothing else	

To express 17, one would write:



This is the only way to parse # # 5 $ 3 2 6 because its decomposition under the definition of L_3 is determined by the initial symbol of the word. Only one symbol can be the initial symbol, so there can only be one parse. Consequently, the only possible interpretation is:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{V}_3[\#\ \#\ 5\ \$\ 3\ 2\ 6]} \\
= & \mathcal{V}_3[\#\ 5\ \$\ 3\ 2] + \mathcal{V}_3[6] & \mathcal{V}_3(2a) \\
= & \big([\mathcal{V}_3[5] + \mathcal{V}_3[\$\ 3\ 2]\big) + \mathcal{V}_3[6] & \mathcal{V}_3(2a) \\
= & \big([\mathcal{V}_3[5] + \big([\mathcal{V}_3[3] \times \mathcal{V}_3[2]]\big)\big) + \mathcal{V}_3[6] & \mathcal{V}_3(2b) \\
= & (5 + (3 \times 2)) + 6 & \mathcal{V}_3[1]\text{, four times} \\
= & 17 & \text{arithmetic}
\end{array}
$$

Exercises 8.2

1. For the language L and interpretation relation $\mathcal{V}$ defined at the beginning of this section, list all the values that are associated with the expression $\mathcal{V}[5\ \#\ 3\ \$\ 2\ \#\ 6]$.

2. Draw the tree corresponding to the second parse of 5 # 3 $ 2 # 6. with respect to the language L.

3. Using the definition of $\mathcal{V}_3$, check that the expression shown in Example 3 evaluates to 17. Then evaluate the following words of L_3:

(a) # $ 3 # 8 9 $ 2 5

(b) # # # $ 3 4 5 6 7

(c) # 3 # 4 # 5 $ 6 7

4. Give an expression in the language L_3 of Example 8.3 which evaluates to 29.

5. Consider the following language, $L_4 \subseteq V^+$, for $V = \mathbb{N} \cup \{ \texttt{\#} , \texttt{\$} \}$.

	$L_4 \subseteq V^+$
1.	$\mathbb{N} \subseteq L_4$
2*a*.	$u, v \in L_4 \Rightarrow \texttt{\#}\ u\ v \in L_4$
2*b*.	$u, v \in L_4 \Rightarrow u\ v\ \texttt{\$} \in L_4$
3.	nothing else

Define an interpretation function for L_4 under which ‘ # ’ stands for addition and ‘ $ ’ for multiplication.

6. For each of the expressions (a)–(c) in Exercise 3, give the corresponding expression in L_4.

7. Determine whether the following words are in L_4, and if so evaluate them:

(a) # # 2 3 4 $ 4
(b) # 2 3 $ 4 5 $
(c) 2 # 3 # 5 $ 6
(d) 2 # 3 # 5 $ 6
(e) 2 3 # 4 # 5 6 $ 7 $ $
(f) 1 # # 2 3 4 5 6 $ 7 8 $ $ 9 $

8.3 Specifying Precedence

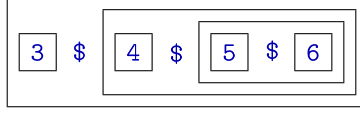
We have seen in Examples 8.2 and 8.3 that one can control the way operations are applied by using disambiguating syntax, like parentheses, or otherwise changing the grammar of the language. One can also deal with the problem mathematically by introducing more structure to the language definition. Let $V = \mathbb{N} \cup \{ \texttt{\$}, \texttt{\#} \}$, as before. First, define a language F and interpretation function $\mathcal{V}_F : F \to \mathbb{N}$, involving only the ‘$’ operation symbol:

	$F \subseteq V^+$	$\mathcal{V}_F : F \to \mathbb{N}$
1.	$\mathbb{N} \subseteq F$	$\mathcal{V}_F[k] = k$, for $k \in \mathbb{N}$
2.	$u \in \mathbb{N},\ v \in F \Rightarrow u\ \texttt{\$}\ v \in F$	$\mathcal{V}_F[u\ \texttt{\$}\ v] = u \times \mathcal{V}_F[v]$
3.	nothing else	

If you study these definitions carefully, you will see a subtle difference from the earlier definition of L (aside from the fact that part 2a is missing!). Part 2 of the definition, builds and interprets words in such a way that multiplications are carried out from right to left. It is said that ‘$’ *associates to the right*. To

Copyright © 2008 Steven D. Johnson

illustrate why this must be, let us consider the word $w = 3 \mathbin{\$} 4 \mathbin{\$} 5 \mathbin{\$} 6$. There is exactly one way to parse w:



Hence, there is exactly one interpretation:

$$
\begin{aligned}
&\mathcal{V}_F[3 \mathbin{\$} 4 \mathbin{\$} 5 \mathbin{\$} 6] \\
&\quad = \quad 3 \times \mathcal{V}_F[4 \mathbin{\$} 5 \mathbin{\$} 6] && \mathcal{V}_F(2) \\
&\quad = \quad 3 \times \big(4 \times \mathcal{V}_F[5 \mathbin{\$} 6]\big) && \mathcal{V}_F(2) \\
&\quad = \quad 3 \times \big(4 \times \big(5 \times \mathcal{V}_F[6]\big)\big) && \mathcal{V}_F(2) \\
&\quad = \quad 3 \times \big(4 \times \big(5 \times 6\big)\big) && \mathcal{V}_F(1) \\
&\quad = \quad 360 && \text{arithmetic}
\end{aligned}
$$

In this case, the order of evaluating multiplications doesn't matter; but it *would* matter if '$' were to denote, say, subtraction (or *real* computer multiplication with overflow).

Building from the sublanguage F, we can now create a language T by introducing the addition symbol.

	$T \subseteq \big(F \cup \{\$\}\big)^+$	$\mathcal{V}_F : F \to \mathbb{N}$
1.	$F \subseteq T$	$\mathcal{V}_T[u] = \mathcal{V}_F[u]$, for $u \in F$
2.	$u \in F, v \in T \Rightarrow u \mathbin{\#} v \in F$	$\mathcal{V}_T[u \mathbin{\#} v] = \mathcal{V}_F[u] + \mathcal{V}_T[v]$
3.	nothing else	

The language T is *exactly the same* as the language L that we originally defined. However, the interpretation, V_T, is constrained to perform both additions and multiplications from right to left. In addition, multiplications are performed before additions. It is said that '$' takes *precedence* over '#'. Let us check this with the original problem expression.

$$
\begin{aligned}
&\mathcal{V}_T[5 \mathbin{\#} 3 \mathbin{\$} 2 \mathbin{\#} 6] \\
&\quad = \quad \mathcal{V}_T[5 \mathbin{\#} 3 \mathbin{\$} 2] + \mathcal{V}_T[6] && {}_T(2) \\
&\quad = \quad \big(\mathcal{V}_T[5] + \mathcal{V}_T[3 \mathbin{\$} 2]\big) + \mathcal{V}_T[6] && {}_T(2) \\
&\quad = \quad \big(\mathcal{V}_T[5] + \mathcal{V}_T[3 \mathbin{\$} 2]\big) + \mathcal{V}_T[6] && {}_T(1) \\
&\quad = \quad \big(\mathcal{V}_T[5] + \big(3 \times \mathcal{V}_T[2]\big)\big) + \mathcal{V}_T[6] && {}_T(2) \\
&\quad = \quad \big(5 + \big(3 \times 2\big)\big) + 6 && {}_T(1)\text{, three times} \\
&\quad = \quad 17 && \text{arithmetic}
\end{aligned}
$$

Since this is the only way to evaluate the word, we are correct in regarding $\mathcal{V}_T$ and V_F as functions.

8.4 Environments

The languages we have seen so far have contained only constants and operations. Computer languages also contain program variables; this is what gives them much of their power. In most languages, a program variable designates some computer-memory location which contains the value. The language compiler translates the symbolic variable name into an address, which is used by the computer to retrieve the designated value.

For our purposes, it is all right to think of the computer's memory as a device that maps the symbolic variable directly to a value; we shall not concern ourselves with the hidden translation from identifier to address. Thus, a memory can be modeled as a function from the domain of program variables to the range of interpreted values. We call such a mapping an *environment*.

As a program executes, its environment changes. The value associated with a program variable is altered by assignment statements, the binding of procedure parameters, and so forth. Our language specifications must reflect this fact and this is done by including the environment in the definition of the interpretation.

Example

Ex 8.4 Let IDE be a set of program variables (such as `x` , `alpha` , `I5` , *etc.*). Let ENV be the set of all environments,

$$\text{ENV} = \{\sigma \mid \sigma\colon \text{IDE} \to \mathbb{N}\}$$

Finally, take V to be the alphabet

$$V = \mathbb{N} \cup \text{IDE} \cup \{\,\texttt{\#}\,\}$$

and define a prefix (hence unambiguous) language $L_6 \subseteq V^+$ and interpretation function $\mathcal{V}\colon L_6 \times \text{ENV} \to \mathbb{N}$ according to:

	$L_6 \subseteq V^+$	$\mathcal{V}_6\colon L_6 \times \text{ENV} \to \mathbb{N}$
1*a*.	$\mathbb{N} \subseteq L_6$	$\mathcal{V}_6[k](\sigma) = k,\ \text{for } k \in \mathbb{N}$
1*b*.	$\text{IDE} \subseteq L_6$	$\mathcal{V}_6[v](\sigma) = \sigma(v),\ \text{for } v \in \text{IDE}$
2.	$u, v \in L_6 \Rightarrow \texttt{\#}\ u\ v \in L_6$	$\mathcal{V}_6[\texttt{\#}\ u\ v](\sigma) = \mathcal{V}_6[u](\sigma) + \mathcal{V}_6[v](\sigma)$
3.	nothing else	

We write $\mathcal{V}_6[u](\sigma)$ rather than $\mathcal{V}_6[u,\sigma]$ or $\mathcal{V}(u,\sigma)$ because it is a little clearer. Since the environment variable is always a single letter, we will later drop the surrounding parentheses.

Copyright © 2008 Steven D. Johnson

Clause 1(b) is new; it specifies the interpretation of a program variable, $v \in \textsc{ide}$, whose value is provided by the environment σ.

Figure 8.4 defines a language of infix arithmetic expressions involving operations of addition, subtraction, multiplication, and negation. Among these operations, negation has the highest precedence, then multiplication, and addition and subtraction have lower, but equal, precedence. Precedence may be superseded by parentheses. All operations are performed from left to right. This example shows that by building the mathematical structures in the right way, we can be precise about the meaning of "ambiguous" languages.

Example

Ex 8.5 If environment $\sigma = \{(\texttt{row}, 5), (\texttt{col}, 8)\}$, then

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{V}_E[\texttt{5 * (row + 2) * - col)}]\sigma} \\
= \mathcal{V}_T[\texttt{5 * (row + 2) * - col)}]\sigma & \mathcal{V}_E(1) \\
= \mathcal{V}_F[\texttt{5}]\sigma \times \mathcal{V}_T[\texttt{(row + 2) * - col)}]\sigma & \mathcal{V}_T(2) \\
= 5 \times \mathcal{V}_T[\texttt{(row + 2) * - col)}]\sigma & \mathcal{V}_F(1) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times \mathcal{V}_T[\texttt{- col)}]\sigma\big) & \mathcal{V}_F(1) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times \mathcal{V}_F[\texttt{- col)}]\sigma\big) & \mathcal{V}_T(1) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times (-\mathcal{V}_E[\texttt{col)}]\sigma\big) & \mathcal{V}_F(2b) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times (-\mathcal{V}_T[\texttt{col)}]\sigma\big) & \mathcal{V}_E(1) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times (-\mathcal{V}_F[\texttt{col)}]\sigma\big) & \mathcal{V}_T(1) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times (-\sigma(\texttt{col}))\big) & \mathcal{V}_F(1b) \\
= 5 \times \big(\mathcal{V}_F[\texttt{(row + 2)}]\sigma \times (-8)\big) & (\texttt{col}, 8) \in \sigma \\
= 5 \times \big(\mathcal{V}_E[\texttt{row + 2}]\sigma \times (-8)\big) & \mathcal{V}_F(2a) \\
= 5 \times \big(\mathcal{V}_T[\texttt{row}]\sigma + \mathcal{V}_E[\texttt{2}]\sigma \times (-8)\big) & \mathcal{V}_E(2a) \\
= 5 \times \big(\mathcal{V}_F[\texttt{row}]\sigma + \mathcal{V}_T[\texttt{2}]\sigma \times (-8)\big) & \mathcal{V}_T(1),\ \mathcal{V}_E(1) \\
= 5 \times \big(\sigma(\texttt{row}) + \mathcal{V}_F[\texttt{2}]\sigma) \times (-8)\big) & \mathcal{V}_F(1b),\ \mathcal{V}_T(1) \\
= 5 \times ((5 + 2) \times (-8)) & (\texttt{row}, 5) \in \sigma,\ \mathcal{V}_F(1a) \\
= -280 & \text{arithmetic}
\end{array}
$$

Exercises 8.4

1. Add a division operation symbol '/ ' to the language E in Figure 8.4 in such a way that 'T/' and '* ' have equal precedence but there is no ambiguity.

Let $V = \mathbb{N} \cup \text{IDE} \cup \{$ `(`, `)`, `+`, `-`, `*` $\}$. Simultaneously define (see Exercise 5, Section 4) the languages F, T, E and interpretations $\mathcal{V}_F, \mathcal{V}_T, \mathcal{V}_E$ as follows:

	$F \subseteq V^+$	$\mathcal{V}_F : F \times \text{ENV} \to \mathbb{N}$
$1a.$	$\mathbb{N} \subseteq F$	$\mathcal{V}_F[k]\sigma = k,$ for $k \in \mathbb{N}$
$1b.$	$\text{IDE} \subseteq F$	$\mathcal{V}_F[v]\sigma = \sigma(v),$ for $v \in \text{IDE}$
$2a.$	$e \in E \Rightarrow$ `(` e `)` $\in F$	$\mathcal{V}_F[$ `(` e `)` $]\sigma = \mathcal{V}_E[e]\sigma$
$2b.$	$e \in E \Rightarrow$ `-` $e \in F$	$\mathcal{V}_F[$ `-` $e]\sigma = -\mathcal{V}_E[e]\sigma$
3.	nothing else	

	$T \subseteq V^+$	$\mathcal{V}_T : T \times \text{ENV} \to \mathbb{N}$
1.	$F \subseteq T$	$\mathcal{V}_T[f]\sigma = \mathcal{V}_F[f]\sigma$ for $f \in F$
2.	$f \in F, t \in T \Rightarrow f$ `*` $t \in T$	$\mathcal{V}_T[f$ `*` $t]\sigma = \mathcal{V}_F[f]\sigma \times \mathcal{V}_T[t]\sigma$
3.	nothing else	

	$E \subseteq V^+$	$\mathcal{V}_E : E \times \text{ENV} \to \mathbb{N}$
1.	$T \subseteq E$	$\mathcal{V}_E[t]\sigma = \mathcal{V}_T[t]\sigma$ for $t \in T$
$2a.$	$t \in T, e \in E \Rightarrow t$ `+` $e \in E$	$\mathcal{V}_E[t$ `+` $e]\sigma = \mathcal{V}_T[t]\sigma + \mathcal{V}_E[e]\sigma$
$2b.$	$t \in T, e \in E \Rightarrow t$ `-` $e \in E$	$\mathcal{V}_E[t$ `-` $e]\sigma = \mathcal{V}_T[t]\sigma - \mathcal{V}_E[e]\sigma$
3.	nothing else	

Figure 8.1: A language of infix arithmetic expressions and its interpretation

2. Add a division operation symbol '`/`' to the language E in Figure 8.4 in such a way that division operations are performed from right to left. Can right-to-left division and left-to-right multiplication have equal precedence?

3. Let environment $\sigma = \{(\texttt{a}, 3), (\texttt{b}, -2), (\texttt{c}, 5)\}$. Evaluate one of the following words from the language E of Figure ?.

(a) `a + 6 - 3 * 5` (c) `b * 4 + - - y`
(b) `((b * 2) * c) * 3` (d) `a * - (b + 5)`

8.5 Backus-Naur Form

Languages defined in the manner of the previous sections are called *context free languages*. There is a standard notation in computer science for context free grammars. It is called *Backus-Naur form* or *BNF*, after John Backus and Peter Naur, who used it to specify the syntax of the algol 60 programming language. A BNF description of the language E in Figure 8.1 looks like this:

$$
\begin{array}{lll}
\langle \mathrm{F} \rangle & ::= & \langle \textsc{natural number} \rangle \\
\langle \mathrm{F} \rangle & ::= & \langle \textsc{program variable} \rangle \\
\langle \mathrm{F} \rangle & ::= & \texttt{-}\ \langle \mathrm{E} \rangle \\
\langle \mathrm{F} \rangle & ::= & \texttt{(}\ \langle \mathrm{E} \rangle\ \texttt{)} \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle\ \texttt{*}\ \langle \mathrm{T} \rangle \\
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \\
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle\ \texttt{+}\ \langle \mathrm{E} \rangle \\
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle\ \texttt{-}\ \langle \mathrm{E} \rangle
\end{array}
$$

The names of inductively defined sets are surrounded by angle brackets $\langle \cdots \rangle$ and "$\langle L \rangle$ ::= *rule*" replaces "*rule* $\in L$. BNF allows us to describe languages concisely, without introducing variables for sub-phrases (e.g. f, t, and e in Figure 8.1).

Exercises 8.5

1. Give the BNF forms for the languages defined in Exercises 1 and 2.

2. Give the BNF form for the language of statements from Section 1.3. Assume that sets $\langle$*assignmentexpression*$\rangle$ and $\langle$*testexpression*$\rangle$ are already defined.

3. If you were going to define an interpretation function for the language of statements, what would its domain and range be?

8.6 Propositional Formulas

In this section we shall explore applications of the language definition style just introduced. For concreteness, the language of propositional logic is used. However, bear in mind that, except for the syntax, the results at the end of this section are valid for other languages, such as arithmetic expressions.

Figure 8.2 defines a language, PROP, of *propositional formulas*. It follows the style of Figure 8.1 in an abbreviated form:

(a) The language PROP is defined using BNF.

(b) Rather than building the language in stages, as was done with arithmetic terms, 8.2 simply specifies the operator precedence.

(c) The interpretation function $\mathcal{P}$ implicitly assumes that the language has been disambiguated.

It is supposed that the Reader can fill in the necessary details when needed.

In Figure 8.2, PROPs' meanings are given in terms of environments.

$$\text{ENV} = \{\sigma \mid \sigma\colon IVS \to \{T, F\}\}$$

A given environment $\sigma \in \text{ENV}$ corresponds to one row of a truth table. Accordingly, let us redefine *tautology*, *contradiction*, and *logical equivalence* in terms of environments.

Definition 8.1 *A* PROP Q *is a* tautology *iff for every* $\sigma \in \text{ENV}$, $\mathcal{P}\sigma\,[Q] = T$. *A* PROP Q *is a* contradition *iff for every* $\sigma \in \text{ENV}$, $\mathcal{P}\sigma\,[Q] = F$.

Definition 8.2 *Two* PROP*s* P *and* Q *are* logically equivalent*, written* $P\,\text{eq}\,Q$*, iff for every* $\sigma \in \text{ENV}$, $\mathcal{P}\sigma\,[P] = \mathcal{P}\sigma\,[Q]$.

As a first exercise, let us validate the intuitive correspondence between logical equivalence (eq) and bi-implication ($\Leftrightarrow$). Proposition 9.1 confirms a fact that we have already used. Conversely, its proof helps validate that our new definitions are sensible. For the purpose of this proposition, suppose an implication operator, `=>`, is included in PROP.

Proposition 8.1 *If* P *and* Q *are* PROP*s, then for all* $\sigma \in \text{ENV}$

$$P \text{ eq } Q \text{ iff } \texttt{(}P \texttt{ => } Q\texttt{) \& (}Q \texttt{ => } P\texttt{)} \textit{ is a tautology.}$$

PROOF: The proof is a straightforward application of the definition of $\mathcal{P}$; induction is not required, although we do need to know that $\mathcal{P}$ is a function (not a relation or partial function).

Let PVAR be a set of propositional variables, and the alphabet $A = \text{PVAR} \cup \{\texttt{(},\ \texttt{)},\ \texttt{0}\ \texttt{1},\ \texttt{-},\ \texttt{|},\ \texttt{\&}\ >\}$. The language PROP $\subseteq A^+$ of *propositional formulas* and its interpretation

$$\begin{aligned} \text{ENV}: &\quad \text{PVAR} \to \{\mathit{true}, \mathit{false}\} \\ \mathcal{P}: &\quad \text{ENV} \times \text{PROP} \to \{\mathit{true}, \mathit{false}\} \end{aligned}$$

are defined as follows:

⟨PROP⟩	::=	`0`	$\mathcal{P}_\sigma[\,\texttt{0}\,] = \mathit{false}$
	❙	`1`	$\mathcal{P}_\sigma[\,\texttt{1}\,] = \mathit{true}$
	❙	⟨PVAR⟩	$\mathcal{P}_\sigma[\,v\,] = \sigma(v)$, for $v \in$PVAR
	❙	`-` ⟨PROP⟩	$\mathcal{P}_\sigma[\,\texttt{-}\,f\,] = \neg\,\mathcal{P}_\sigma[\,f\,]$
	❙	`(` ⟨PROP⟩ `)`	$\mathcal{P}_\sigma[\,\texttt{(}\,f\,\texttt{)}\,] = \mathcal{P}_\sigma[\,f\,]$
	❙	⟨PROP⟩ `&` ⟨PROP⟩	$\mathcal{P}_\sigma[\,f_1\,\texttt{\&}\,f_2\,] = \mathcal{P}_\sigma[\,f_1\,] \wedge \mathcal{P}_\sigma[\,f_2\,]$
	❙	⟨PROP⟩ `|` ⟨PROP⟩	$\mathcal{P}_\sigma[\,f_1\,\texttt{|}\,f_2\,] = \mathcal{P}_\sigma[\,f_1\,] \vee \mathcal{P}_\sigma[\,f_2\,]$

with precedence $\boxed{-} \succ \boxed{\&} \succ \boxed{|}$.

Figure 8.2: A language of propositional formulas and its interpretation

IF PART: Let $\sigma \in$ ENV and assume that P eq Q, that is, by Definition 9.3, $\mathcal{P}\sigma[P] = \mathcal{P}\sigma[Q]$. Then

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{(}P\ T{=}{>}\ Q\texttt{) \& (}Q \texttt{ => } P\texttt{)}} \\
= & (\mathcal{P}\sigma[P] \Rightarrow \mathcal{P}\sigma[Q]) \wedge (\mathcal{P}\sigma[Q] \Rightarrow \mathcal{P}\sigma[P]) & \text{(Defn. of } \mathcal{P}\text{)} \\
= & (\mathcal{P}\sigma[P] \Rightarrow \mathcal{P}\sigma[P]) \wedge (\mathcal{P}\sigma[P] \Rightarrow \mathcal{P}\sigma[P]) & \text{(Assumption that } \mathcal{P}\sigma[P] = \mathcal{P}\sigma[Q]\text{)} \\
= & T & \text{(meanings of } \Rightarrow, \wedge\text{)}
\end{array}
$$

Thus, by Defintion 9.2, $\texttt{(}P \texttt{ => } Q\texttt{) \& (}Q \texttt{ => } P\texttt{)}$ is a tautology.

ONLY-IF PART: Assume that $\texttt{(}P \texttt{ => } Q\texttt{) \& (}Q \texttt{ => } P\texttt{)}$ is a tautology, and let $\sigma \in$ ENV be any environment.

$$
\begin{array}{rcll}
T & = & \mathcal{P}\sigma[\texttt{(}P \texttt{ => } Q\texttt{) \& (}Q \texttt{ => } P\texttt{)}] & \\
 & = & (\mathcal{P}\sigma[P] \Rightarrow \mathcal{P}\sigma[Q]) \wedge (\mathcal{P}\sigma[Q] \Rightarrow \mathcal{P}\sigma[P]) & \text{(defn. } \mathcal{P}\text{)}
\end{array}
$$

This equality can hold only if $\mathcal{P}\sigma[P] = \mathcal{P}\sigma[Q]$. Therefore, by Definition 9.3, P *eq* Q.

∎

Example

Ex 8.6 A function that translates sentences from one language to another (possibly the same) language called a *transliteration*. We are often interested in whether and how transliterations change the meaning of the original sentence. This example applies this idea to define a generalization of DeMorgan's identity for boolean algebras.

The *DeMorgan Dual* of $f \in$ PROP is obtained by switching all 0's and 1's, +'s and *'s, and inserting a `-` just before every variable symbol.

For instance, the DeMorgan dual of `( a &  b) | (( c | 0) & ( -b |  a))`
is `(-a | -b) & ((-c & 1) | (--b & -a))`

Inspecting the DNFs of these formulas: $\left\{ \begin{array}{l} \overline{a}\,\overline{b}\,c + a\,\overline{b}\,c + a\,b\,\overline{c} + a\,b\,c \\ \overline{a}\,\overline{b}\,\overline{c} + \overline{a}\,b\,\overline{c} + \overline{a}\,b\,c + a\,\overline{b}\,\overline{c}. \end{array} \right\}$ reveals that they are negations of each other—they have no clauses in common and together contain all eight possible clauses.

(a) *Define a recursive function* $\mathbb{D}$: PROP $\rightarrow$ PROP *that gives the DeMorgan dual of any* $F \in$ PROP.

$$
\begin{array}{rcl}
\mathbb{D}[\,\texttt{0}\,] & = & \texttt{1} \\
\mathbb{D}[\,\texttt{1}\,] & = & \texttt{0} \\
\mathbb{D}[\,v\,] & = & \texttt{-}\hat{}v \text{ for } v \in IVS \\
\mathbb{D}[\,\texttt{-}\ P\,] & = & \texttt{-}\hat{}\mathbb{D}[\,P\,] \\
\mathbb{D}[\,P\ \texttt{|}\ Q\,] & = & \mathbb{D}[\,P\,]\hat{}\texttt{\&}\hat{}\mathbb{D}[\,Q\,] \\
\mathbb{D}[\,P\ \texttt{\&}\ Q\,] & = & \mathbb{D}[\,P\,]\hat{}\,\texttt{|}\,\hat{}\mathbb{D}[\,Q\,]
\end{array}
$$

(b) *Prove that the DeMorgan dual of a propositional formula is its logical negation:* For all $\sigma \in \text{ENV}$ and $F \in \text{PROP}$, $\mathcal{P}_\sigma[\mathbb{D}[F]] = \neg\mathcal{P}_\sigma[F]$.

The proof is a straightforward induction on words in PROP. The crux of the argument is in the base case for variables, where for any $v \in \mathit{IVS}$ we have

$$\mathcal{P}_\sigma[\mathbb{D}[v]] = \mathcal{P}_\sigma[\texttt{-}\hat{\ }v] = \neg\mathcal{P}_\sigma[v]$$

All steps above are justfied by the definitions of $\mathcal{P}$ or $\mathbb{D}$. An example of the inductive cases, for formulas of the form P `+` Q, is

$$\begin{array}{rcll}
\mathcal{P}_\sigma[\mathbb{D}[P \texttt{ | } Q]] & = & \mathcal{P}_\sigma[\mathbb{D}[P]\hat{\ }\texttt{\&}\hat{\ }\mathbb{D}[Q]]] & \text{(defn. } \mathbb{D}) \\
& = & \mathcal{P}_\sigma[\mathbb{D}[P]] \wedge \mathcal{P}_\sigma[\mathbb{D}[Q]]] & \text{(defn. } \mathcal{P}) \\
& \overset{H}{=} & \neg\mathcal{P}_\sigma[P] \ \wedge \ \neg\mathcal{P}_\sigma[Q] & \text{(I.H. used twice)} \\
& = & \neg(\mathcal{P}_\sigma[P] \ \vee \mathcal{P}_\sigma[Q]) & \text{(DeMorgan's Identity)} \\
& = & \neg\mathcal{P}_\sigma[P\hat{\ }\texttt{|}\hat{\ }Q] & \text{(defn. } \mathcal{P})
\end{array}$$

∎

Exercises 8.6

1. Add an implication operator, '`=>`' to PROP.

2. Use transliteration to add an implication "macro" to PROP. That is, define a language $\text{PROP}_\Rightarrow$ that includes '`=>`' and a translation function $F\colon \text{PROP}_\Rightarrow \to \text{PROP}$ that correctly re-interprets f_1 `=>` f_2 as `(-` f_1 `+` f_2 `)` .

8.7 Substitution

A *substitution* is the simultanesous replacement of formulas for variables in an expression.

Definition 8.3 *Let* $F, P_1, \ldots, P_k \in \text{PROP}$ *and* $v_1, \ldots, v_k \in \text{PVAR}$. *The* substitution

$$F\left[{P_1,\ \ldots,\ P_k \atop v_1,\ \ldots,\ v_k}\right]$$

denotes the result, $\mathcal{S}[F]$*, of a substitution function* $S\colon \text{PROP} \to \text{PROP}$ *defined*

Copyright © 2008 Steven D. Johnson

as follows:

$$\begin{array}{rcl}
\mathcal{S}[\,\texttt{0}\,] &=& 0 \\
\mathcal{S}[\,\texttt{1}\,] &=& 1 \\
\mathcal{S}[\,x\,] &=& \begin{cases} P_j & \textit{if } x = v_j \\ x & \textit{otherwise} \end{cases} \\
\mathcal{S}[\,\texttt{-}\ Q\,] &=& \texttt{-}\hat{}\mathcal{S}[\,P\,] \\
\mathcal{S}[\,Q\ \texttt{|}\ Q'\,] &=& \mathcal{S}[\,Q\,]\hat{}\,\texttt{|}\,\hat{}\mathcal{S}[\,Q'\,] \\
\mathcal{S}[\,Q\ \texttt{\&}\ Q'\,] &=& \mathcal{S}[\,Q\,]\hat{}\texttt{\&}\hat{}\mathcal{S}[\,Q'\,]
\end{array}$$

According to the definition, we often consider the substitution *function*

$$\mathcal{S} \text{ as specified by } \left[{}^{P_1,\ \ldots,\ P_k}_{v_1,\ \ldots,\ v_k}\right]$$

which may be applied to any formula $F \in$ PROP, writing write $\mathcal{S}[\,P\,]$ to denote the result.

Example

Ex 8.7

$$\texttt{p * (-q + r)}\left[{}^{\texttt{q (s+t) p}}_{\texttt{p\ \ \ q\ \ \ r}}\right] \equiv \texttt{q * ((s+t) + p)}$$

In performing substitutions, one must take care to preserve operator precedence. Above, this is done by parenthesizing `(s + t)`*.*

The three theorems that follow verify the fundamental rules for logical manipulations involving substitution. Since we have been using these results all our lives, another way to look at these theorems is that they validate the definitions given so far.

Lemma 8.2 (Substitution Lemma) *Let $\mathcal{S}$ be a substitution and σ an environment. Define an new environment σ' as follows:*

$$\sigma'(v) = \mathcal{P}\sigma\,[\mathcal{S}(v)] \;\text{ for } v \in \mathit{IVS}$$

Then for all PROP*s P,*

$$\mathcal{P}\sigma'\,[P] = \mathcal{P}\sigma\,[\mathcal{S}\,[P]]$$

PROOF: The proof is a straightforward structural induction on PROP. The inductive cases hold because the operations are functions. The interesting base case is the one for a variable $v \in \mathit{IVS}$. In that case, we have

$$\mathcal{P}\sigma'\,[v] = \mathcal{P}\sigma\,[\mathcal{S}\,[p]]$$

which is exactly what we need to make the Theorem true.

∎

The Substitution Lemma states that for the language of propositions, a call-by-value style evaluation—in which variables are bound to expressions' values—is equivalent to a call-by-name style evaluation. This is an exact equivalence because there is no form of looping in the language.

More significantly, the lemma says that our notion of substitution interacts well with our notion of evaluation. For example,

Theorem 8.3 (Tautology Theorem) *Let P be a PROP and $\mathcal{S}$ a substitution, If P is a tautology, then so is $\mathcal{S}[P]$.*

PROOF: Apply the definition of *tautology* and the Substitution Lemma. The details are left as Exercise 2. ∎

For example, the formula

$$Q \equiv \texttt{(p | q \& (p => r)) | -(p | q \& (p => r))}$$

is a tautology because `p | -p` is a tautology and there is a subsitition,

$$\mathcal{S}[\texttt{p}] = \texttt{(p | q \& (p => r))}$$

under which

$$Q \equiv \mathcal{S}[\texttt{p | -p}]$$

Theorem 9.3 gives us one way to abbreviate the analysis of PROPs. The next theorem states that we can analyze PROP *schemes* as well as individual PROPs.

Theorem 8.4 (Substitution Theorem) *If $P \,\mathrm{eq}\, Q$ then for any substitution $\mathcal{S}$, $\mathcal{S}[P] \,\mathrm{eq}\, \mathcal{S}[Q]$*

PROOF: Use the Substitution Lemma. ∎

Thus, not only is formula

$$\texttt{p | (q | r)} \;\mathrm{eq}\; \texttt{(p | q) | r}$$

but the two formula *schemes*,

$$P \texttt{ | (}Q\texttt{ | }R\texttt{)} \quad \text{and} \quad \texttt{(}P\texttt{ | }Q\texttt{) |}R$$

are equivalent for arbitray sub-PROPs P, Q, and R. In fact, we reason about PROP schemes far more often than we reason about PROP individuals.

The following result is very important to the way we do proofs. It says you can "replace equals with equals" and still preserve equivalence.

Theorem 8.5 (Replacement Theorem) *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be substitutions such that, for all $v \in IVS$, $\mathcal{S}_1(v)$ eq $\mathcal{S}_2(v)$. Then for any PROP P,*

$$\mathcal{S}_1[P] \;\mathrm{eq}\; \mathcal{S}_2[P]$$

Copyright © 2008 Steven D. Johnson

PROOF: Let σ be any environment and define σ' to be

$$\sigma'(v) = \mathcal{P}\sigma\,[\mathcal{S}_1[v]]$$

Since $\mathcal{S}_1(v)$ eq $\mathcal{S}_2(v)$, it is also the case that $\sigma'(v) = \mathcal{P}\sigma\,[\mathcal{S}_2[v]]$ for all $v \in IVS$. Using the Substitution Lemma twice, we have

$$\mathcal{P}\sigma\,[\mathcal{S}_1[P]] = \mathcal{P}\sigma'\,[P] = \mathcal{P}\sigma\,[\mathcal{S}_2[P]]$$

as desired.

∎

For example, `p⇒q` eq `¬q ⇒ ¬p` so

`(q ⇒ p) ∧ (p ⇒ q)`   eq   `(q ⇒ p) ∧ (¬q ⇒ ¬p)`

under the substitutions

v	$\mathcal{S}_1[v]$	$\mathcal{S}_2[v]$
p	p	p
q	q	q
r	p ⇒ q	¬q ⇒ ¬p

These results about substitution and replacement do not depend in any fundamental way on the syntax of formulas in PROP. Exercise 3 asks you to define substitution for arithmetic expressions. Exercise 3 asks you to consider a more general definition of substitution, applicable to any language defined in the style developed in this chapter.

The question of interpretation is more important. Is it the case that all recursively defined interpretations allow substitution? One answer is that substitution is so important that only those interpretations that make the Substitution Lemma (Lemma 9.2) true are allowed.

Exercises 8.7

1. Let $F \equiv p \wedge (q \vee r)$. Perform the following substitutions

(a) $F\begin{bmatrix} r, & q, & p \\ p, & q, & r \end{bmatrix}$

(b) $F\begin{bmatrix} p \vee r \\ p \end{bmatrix}$

(c) $F\begin{bmatrix} p \vee r, & q \Rightarrow r \\ p, & r \end{bmatrix}$

(d) $\left(F\begin{bmatrix} p \vee r \\ p \end{bmatrix}\right)\begin{bmatrix} q \\ p \end{bmatrix}$

(e) $\left(F\begin{bmatrix} q \\ p \end{bmatrix}\right)\begin{bmatrix} p \vee r \\ p \end{bmatrix}$

Copyright © 2008 Steven D. Johnson

2. Define substitution for arithmetic terms as defined in Figure 8.1. Show that the Substitution Lemma (Lemma 9.2) holds for their interpretation.

3. Describe in general terms the process of defining substitution for any language that contains variables. Try to write down the appropriate generalized definitions and theorems.

8.8 The Programming Language of Statements

The STMT programming language was first introduced in Chapter 1 and has been referred to often throughout this textbook. In this section we shall apply the definitional style developed in this chapter to write a more rigorous specification of program syntax and meaning. These specifications appear in Definitions 8.4 and 10.3.

Sentences in STMT—programs—are built from a set of keywords,

$$\{\texttt{begin}, \texttt{end}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{while}, \texttt{do}, \texttt{:=}, \texttt{;}\}$$

and phrases coming from:

(a) The language of *arithmetic terms* used in assignment statements. This language and its interpretation are essentially the same as that of Figure 8.1.

(b) *Test expressions* used in `if` and `while` statements. Test expressions are logical combinations of arithmetic comparisons. Their interpretation is straightforward to define and is left as an exercise.

Discussion Points The clauses in Definition 10.3 are subtle. They must be studied carefully.

- Programs express computation in terms of *assignment*: recording information in a memory. The interpretation of statements reflects this model. A program starts with an initial memory, runs for a while, and then stops, leaving its results in an updated memory. We model memories abstractly with environments mapping program variables to values. Thus, the interpretation function maps from environments to environments.

$$\mathcal{M} : \text{ENV} \times \text{STMT} \xrightarrow{p} \text{ENV}$$

  $\mathcal{M}$ is a *partial* function. For non-terminating programs it does not give a value, as discussed below.

- The interpretation of assignment says to take the environment function σ, remove the ordered pair for program variable V, and replace it with one that binds V to the value of T.

$$[\sigma \setminus \{(V, \sigma(V)\}] \cup \{(v, \mathcal{T}_\sigma[\,T\,]\}$$

Copyright © 2008 Steven D. Johnson

- The compound-statement interpretations says, execute statement S_1, and then execute S_2 using this resulting memory, σ'. It might have been written even more mysteriously as

$$\mathcal{M}_{(\mathcal{M}_\sigma[S_1])}[\,S_2\,]$$

- Unlike all other rules, the rule for `while`-statements does not reduce interpretation to a *proper* sub-sentence. The interpretation of certain statements is undefined. For instance,

$$\begin{aligned}
& \mathcal{M}_\sigma[\,\texttt{while}\ x = x\ \texttt{do}\, x := x\,] \\
& \quad \boxed{\mathcal{P}_\sigma[\,x = x\,] = \cdots = \mathit{true}} \\
\overset{(d)}{==} \quad & \mathcal{M}_{\sigma'}[\,\texttt{while}\ x = x\ \texttt{do}\, x := x\,] \\
& \quad \boxed{\text{where } \sigma'(x) = \mathcal{M}_\sigma[\,x := x\,] = \cdots = \sigma} \\
= \quad & \mathcal{M}_\sigma[\,\texttt{while}\ x = x\ \texttt{do}\, x := x\,] \\
& \vdots
\end{aligned}$$

  Of course, this is just what we want our model to describe: a non-terminating program does not produce a “final” memory.

Let us use Definition 10.3 prove an interesting fact about compound statements.

Proposition 8.6 *The compound operator, ‘`;`’ is associative in the sense that for all $\sigma \in$ ENV and statements S_1, S_2, and S_3,*

$$\begin{aligned}
& \mathcal{M}_\sigma[\,\texttt{begin begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end ;}\ S_3\ \texttt{end}\,] \\
= \quad & \mathcal{M}_\sigma[\,\texttt{begin}\ S_1\ \texttt{; begin}\ S_2\ \texttt{;}\ S_3\ \texttt{end end}\,]
\end{aligned}$$

PROOF: Apply Definition 10.3.

∎

Thus, it is not ambiguous to write `begin` S_1 `;` S_2 `;` S_3 `end` because it doesn't matter how `begin`-`ends` are associated.

The next relatively simple fact is the first step in reducing program correctness statements to purely logical conditions. Recall that the notation `{`P`} S {`Q`}` says, “If statement S starts with a memory in which property P holds (and if it terminates), property Q holds in the final memory. In more precise terms, for all $\sigma \in$ ENV,

$$\mathcal{P}_\sigma[\,P\,] \text{ implies } \mathcal{P}_{\sigma'}[\,Q\,] \text{ where } \sigma' = \mathcal{M}_\sigma[\,S\,]$$

Proposition 8.7 $\{P\}\,V$ `:=` $T\,\{Q\}$ *iff* $P \Rightarrow Q^T_V$.

PROOF: ($\Rightarrow$) Let $\sigma \in$ ENV, and assume $\{P\}\,V$ `:=` $T\,\{Q\}$ is true. Then $\mathcal{P}_\sigma[\,P\,] \Rightarrow \mathcal{P}_{\sigma'}[\,Q\,]$ where $\sigma' = \mathcal{M}_\sigma[\,V := T\,]$. By Definition 10.3, $\sigma'(V) = \mathcal{T}_\sigma[\,T\,]$. Hence, by the Substitution Lemma (Lemma 9.2, suitably generalized), $\mathcal{P}_{\sigma'}[\,Q\,]$ is logically equivalent to $\mathcal{P}_\sigma[\,Q^T_V\,]$. Therefore, $\mathcal{P}_\sigma[\,P \Rightarrow Q^T_V\,]$ is true.

($\Rightarrow$) By the same argument as above.

∎

Definition 8.4 *The languages of arithmetic terms (*TERM*), comparisons (*COMP*), tests (*TEST*) and program statements (*STMT*) are defined according to the grammar below. All operators associate to the right with precedence* $\boxed{-}_1 \succ \boxed{*} \succ \boxed{+} = \boxed{-}_2$ *and*

$$
\begin{array}{lll}
\langle\text{TERM}\rangle & ::= & \langle\text{NUMERAL}\rangle \\
 & ::= & \langle\text{IDENTIFIER}\rangle \\
 & ::= & \texttt{-}_1\ \langle\text{TERM}\rangle \\
 & ::= & \texttt{(}\ \langle\text{TERM}\rangle\ \texttt{)} \\
 & ::= & \langle\text{TERM}\rangle\ \texttt{*}\ \langle\text{TERM}\rangle \\
 & ::= & \langle\text{TERM}\rangle\ \texttt{+}\ \langle\text{TERM}\rangle \\
 & ::= & \langle\text{TERM}\rangle\ \texttt{-}_2\ \langle\text{TERM}\rangle
\end{array}
\qquad
\begin{array}{lll}
\langle\text{COMP}\rangle & ::= & \langle\text{TERM}\rangle\ \texttt{=}\ \langle\text{TERM}\rangle \\
 & ::= & \langle\text{TERM}\rangle\ \texttt{<}\ \langle\text{TERM}\rangle \\
\langle\text{TEST}\rangle & ::= & \langle\text{COMP}\rangle \\
 & ::= & \texttt{-}_3\ \langle\text{COMP}\rangle \\
 & ::= & \texttt{(}\ \langle\text{COMP}\rangle\ \texttt{)} \\
 & ::= & \langle\text{COMP}\rangle\ \texttt{\&}\ \langle\text{COMP}\rangle \\
 & ::= & \langle\text{COMP}\rangle\ \texttt{|}\ \langle\text{COMP}\rangle
\end{array}
$$

$$
\begin{array}{llll}
\langle\text{STMT}\rangle & ::= & \langle\text{IDENTIFIER}\rangle\ \texttt{:=}\ \langle\text{TERM}\rangle & (\text{assignment}) \\
 & ::= & \texttt{begin}\ \langle\text{STMT}\rangle\ \texttt{;}\ \langle\text{STMT}\rangle\ \texttt{end} & (\text{compound}) \\
 & ::= & \texttt{if}\ \langle\text{TEST}\rangle\ \texttt{then}\ \langle\text{STMT}\rangle\ \texttt{else}\ \langle\text{STMT}\rangle & (\text{conditional}) \\
 & ::= & \texttt{while}\ \langle\text{TEST}\rangle\ \texttt{do}\ \langle\text{STMT}\rangle & (\text{repetition})
\end{array}
$$

Definition 8.5 *Given interpretations* $\mathcal{T}\colon \text{ENV} \times \text{TERM} \to \mathbb{N}$ *and* $\mathcal{P}\colon \text{ENV} \times \text{TEST} \to \{\mathit{true}, \mathit{false}\}$*, the operational meaning of programs in* STMT *(Definition 8.4) is given by the partial function* $\mathcal{M}\colon \text{ENV} \times \text{STMT} \to \text{ENV}$ *defined as follows:*

$$(a)\quad \mathcal{M}_\sigma[\, V\ \texttt{:=}\ T\,] = [\sigma \setminus \{(V, \sigma(V)\}] \cup \{(V, \mathcal{T}_\sigma[\,T\,]\}$$

$$(b)\quad \mathcal{M}_\sigma[\,\texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\,] = \mathcal{M}_{\sigma'}[\,S_2\,] \text{ where } \sigma' = \mathcal{M}_\sigma[\,S_1\,]$$

$$(c)\quad \mathcal{M}_\sigma[\,\texttt{if}\ Q\ \texttt{then}\ S_1\ \texttt{else}\ S_2\,] = \begin{cases} \mathcal{M}_\sigma[\,S_1\,] & \text{if } \mathcal{P}_\sigma[\,Q\,] = \mathit{true} \\ \mathcal{M}_\sigma[\,S_2\,] & \text{if } \mathcal{P}_\sigma[\,Q\,] = \mathit{false} \end{cases}$$

$$(d)\quad \mathcal{M}_\sigma[\,\texttt{while}\ Q\ \texttt{do}\, S\,] = \begin{cases} \sigma, & \text{if } \mathcal{P}_\sigma[\,Q\,] = \mathit{false} \\ \mathcal{M}_{\sigma'}[\,\texttt{while}\ Q\ \texttt{do}\ S\,] & \text{if } \mathcal{P}_\sigma[\,Q\,] = \mathit{true} \\ \quad \text{where } \sigma' = \mathcal{M}_\sigma[\,S\,], & \end{cases}$$

Example

Ex 8.8 In Proposition 4.1 (p. ??) it was shown that

```
{z + xy = AB}
while x ≠ 0 do
   begin
   x := x − 1;
   z := z + y
   end;
end {z = AB}
```

The proof of invariance involved reasoning about the values of identifiers before (x, y and z) and after (x', y' and z') executing the loop body. Proposition 8.7 says that this kind of temporal reasoning may be reduced to "pure logic":

$$\{z + xy = AB \ \wedge \ x \neq 0\}\,\texttt{begin}\ \ x \,\texttt{:=}\, x - 1\,\texttt{;}\ \ z \,\texttt{:=}\, z + 1\ \ \texttt{end}\,\{z + xy = AB\}$$

eq (by Prop. 8.7)

$$\{z + xy = AB \ \wedge \ x \neq 0\}\ x \,\texttt{:=}\, x - 1\ \left\{\big(z + xy = AB\big)\big[{}^{z\,+\,1}_{\ z}\big]\right\}$$

eq (by Prop. 8.7)

$$\big(z + xy = AB \ \wedge \ x \neq 0\big) \ \Rightarrow \ \Big(\big(z + xy = AB\big)\big[{}^{z\,+\,y}_{\ z}\big]\Big)\big[{}^{x\,-\,1}_{\ x}\big]$$

Performing these substitutions, we get

$$\big(z + xy = AB \ \wedge \ x \neq 0\big) \ \Rightarrow \ \Big(\big(z + xy = AB\big)\big[{}^{z\,+\,y}_{\ z}\big]\Big)\big[{}^{x\,-\,1}_{\ x}\big]$$

eq

$$\big(z + xy = AB \ \wedge \ x \neq 0\big) \ \Rightarrow \ \big((z + y) + xy = AB\big)\big[{}^{x\,-\,1}_{\ x}\big]$$

eq

$$\big(z + xy = AB \ \wedge \ x \neq 0\big) \ \Rightarrow \ \big[(z + y) + (x - 1)y = AB\big]$$

eq (simplifying $(x + y) + (x - 1)y$)

$$\big(z + xy = AB \ \wedge \ x \neq 0\big) \ \Rightarrow \ \big(z + xy = AB\big)$$

Which is tautologically true.

Exercises 8.8

1. Define interpretations for arithmetic and test expressions. $\mathcal{T}\colon$ ENV × TERM → $\mathbb{N}$ $\mathcal{P}\colon$ ENV × TEST → $\{\mathit{true},\ \mathit{false}\}$.

Copyright © 2008 Steven D. Johnson

2. In Example 8.8 the subformula

$$\Big(\big(z + xy = AB\big)\big[{}^{z+y}_{z}\big]\Big)\big[{}^{x-1}_{x}\big]$$

is derived. Is this the same as

$$\big(z + xy = AB\big)\big[{}^{z+y,\; x-1}_{z,\;\; x}\big]$$

3. Suppose that, for all $\sigma \in$ ENV, $T \in$ TERM and $C \in$ COMP, $\mathcal{T}_\sigma[\,T\,] = 42$ and $\mathcal{P}_\sigma[\,C\,] = \mathit{true}$. Describe how programs in STMT behave.

8.9 *Discussions

8.9.1 Parenthesized Expressions

In Example 8.2 it is claimed that fully parenthesized expressions are unambiguous. This fact seems intuitively obvious but proving it rigourously requires delving into that intuition. The essential property of this language is that its parentheses are "balanced."

As in the example, Let alphabet $A = \mathbb{N} \cup \{ \#, \$, (,) \}$. For $w \in A^+$, define $\Delta[w]$ to be the number of left parentheses in w minus the number of right parentheses. For example, $\Delta[((5 \$ 2()] = 2$ and $\Delta[)3()7(] = 0$.

The language L_2 of 8.2 was defined

	$L_2 \subseteq A^+$			$\mathcal{V}: L_2 \to \mathbb{N}$
1.	$\langle L_2 \rangle$	::=	$\mathbb{N}$	$\mathcal{V}[n] = n$ for $n \in \mathbb{N}$
2*a*.		|	$(\langle L_2 \rangle \# \langle L_2 \rangle)$	$\mathcal{V}[(u \# v)] = \mathcal{V}[u] + \mathcal{V}[v]$
2*b*.		|	$(\langle L_2 \rangle \$ \langle L_2 \rangle)$	$\mathcal{V}[(u \$ v)] = \mathcal{V}[u] \times \mathcal{V}[v]$

Proposition 8.8 *Let L_2 be the language defined in Example 8.2. Then for all $w \in L_2$, $\Delta[w] = 0$.*

PROOF: The proof is by structural induction on L_2. In the base case, if $k \in \mathbb{N}$ then k contains no parentheses, so $\Delta[k] = 0$. For the induction case, assume $\Delta[u] = \Delta[v] = 0$. Then

$$\Delta[(u*v)] = 1 + \Delta[u] + \Delta[v] - 1 = 0$$

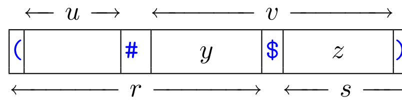
And similarly for $\Delta[(u+v)]$.

∎

Proposition 8.8 captures only part of the quality of being balanced. How can we capture the notion of being *properly* balanced? The answer is not obvious, but a little experimentation will convince you that the next proposition is what we need:

Proposition 8.9 *Let L_2 be the language defined in Example 8.2, and let $w \in L_2$. If $w = r \# s$ (or $r \$ s$) for any two words $r, s \in V^+$, then $\Delta[r] > 0$ and $\Delta[s] < 0$.*

PROOF: The proof is by structural induction on L_2 and the base case holds vacuously. For the induction case, assume $w = (u \# v)$ and that the induction hypothesis holds for u and v. Now suppose that w also equals $r \$ s$, so that we have the following:



Copyright © 2008 Steven D. Johnson

By Proposition 8.8 $\Delta[u] = 0$, and by the induction hypothesis, $\Delta[y] > 0$. Hence,

$$\Delta[r] = \Delta[\texttt{(}u \,\texttt{\#}\, y] = 1 + \Delta[y] > 0$$

Again by Proposition 8.8, $\Delta[w] = 0$, so it must be the case that

$$\Delta[s] = \Delta[z\texttt{)}] = -\Delta[r] < 0$$

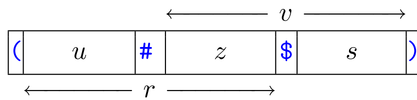
A similar argument holds if the '$' occurs to the right of the '#'.

∎

We are now in a position to prove that each word in L_2 has a unique parse.

Proposition 8.10 *Let L_2 be the language defined in Example 8.2, and let $w \in L_2$, $w \notin \mathbb{N}$. Then there is exactly one pair of words, $u, v \in L_2$ such that $w = \texttt{(}u \,\texttt{\#}\, v\texttt{)}$ (or $\texttt{(}u \,\texttt{\$}\, v\texttt{)}$).*

PROOF: We will assume that there are two such pairs and reach a contradiction. Without loss in generality, assume that $w = \texttt{(}u \,\texttt{\#}\, v\texttt{)}$ and $w = \texttt{(}r \,\texttt{\$}\, s\texttt{)}$ and u, v, r, $s \in L_2$, so that we have the following:



By Proposition 8.8, $r \in L_2$ implies $\Delta[z] < 0$ while $u \in L_2$ implies $\Delta[z] > 0$. This is a contradiction, so either u and v or r and s do not exist.

∎

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Chapter 9

Formal Logic

In this Chapter we shall apply the techniques of Chapter 8 to the language of logic. One purpose is to verify, in a mathematically rigorous manner, that the rules of reasoning we are taught, use, and believe are valid. It should come as no surprise that they are, but it is worthwhile to contemplate how much of the mathematics (or anything else) we learn without looking very deeply into what it means.

The propositions, lemmas, and theorems in Sec. 9.1 all have to do validating the basic formula manipulations—their syntax—used in mathematical reasoning in terms of a precisely defined notion of what these formulae mean—their semantics.

A *calculus* is a system for reasoning symbolically in semantically valid ways. "The Calculus" (of continuous functions on $\mathbb{R}$) is a deeply developed and elegant example. A first course in The Calculus spends some time on semantics, what differentiation and integration mean and how they are related, and some time on symbolic manipulation rules, such as partial differentiation and integration by parts. When you apply The Calculus, you are performing symbolic manipulations without thinking about what they mean (unless they don't work).

This chapter builds a calculus for logical reasoning, which is extended in Chapter 10 to a reasoning system for programs. The emphasis is *validity*, proving that the symbolic rules are correct with respect to the underlying semantics. Toward the end of Chapter 10, we will begin to explore applications to program refinement.

9.1 Propositional Logic

Definition 9.1 *A* well formed formula*, or WFF, is a* term *(Definition* **??***) in the data type of logical operations,* $\mathcal{B} = \langle \{T, F\};\ \wedge,\ \vee,\ \Rightarrow,\ \neg;\ ; T, F\rangle$

A *WFF* represents a possible combination of elementary propositions. For this

reason the individual variable symbols are often called *propositional variables*.

$$IVS = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \dots\},$$

We will use capital letters P, Q, R, ..., to refer to *WFF*s. We will also use a parenthesized infix notation to make *WFF*s easier to read. Following Section 8.3, we define the precedence of operations to be

↑ (*highest*)
¬ (*rank*= 1)
∧ (*rank*= 2, *associative*)
∨ (*rank*= 2, *associative*)
⇒ (*rank*= 2, *right-associative*)
(*lowest*)
↓ (*lowest*)

For example,

`p ∨ q ∧ r`	means `p ∨ (q ∧ r)`
`p ⇒ q ⇒ r`	means `p ⇒ (q ⇒ r)`
`¬ p ∨ q ⇒ r`	means `((¬ p) ∨ q) ⇒ r`

Our informal description of *WFF* in Section 2 can now be replaced by a rigorous one. Although we can certainly still use truth tables to evaluate propositions, our definition of *WFF*s' meanings is of environments. A given environment $\sigma\colon VAR \to \{T, F\}$ corresponds to one row of a truth table.

$$\text{ENV} = \{\sigma \mid \sigma\colon IVS \to \{T, F\}\}$$

Definition 9.2 *A WFF Q is a* tautology *iff for every* $\sigma \in \text{ENV}$, $\mathcal{T}\sigma\,[Q] = T$. *A WFF Q is a* contradiction *iff for every* $\sigma \in \text{ENV}$, $\mathcal{T}\sigma\,[Q] = F$.

Definition 9.3 *Two WFFs P and Q are* logically equivalent*, written* $P \,\text{eq}\, Q$*, iff for every* $\sigma \in \text{ENV}$, $\mathcal{T}\sigma\,[P] = \mathcal{T}\sigma\,[Q]$.

As a first exercise, let us validate the intuitive correspondence between logical equivalence (eq) and bi-implication (⇔). The correspondence is "intuitive" because we learned it at an early age and have been using it ever since. Proposition 9.1 merely confirms what is believed. Conversely, the proof of Prop. 9.1 helps confirm that our definitions are sensible.

Proposition 9.1 *If P and Q are WFFs, then for all* $\sigma \in \text{ENV}$

$$P \text{ eq } Q \;\text{ iff }\; (P{\Rightarrow}Q){\wedge}(Q{\Rightarrow}P) \textit{ is a tautology.}$$

PROOF: The proof is a straightforward application of the definition of $\mathcal{T}$; induction is not required, although we do need to know that $\mathcal{T}$ is a function (not a relation or partial function).

Copyright © 2008 Steven D. Johnson

IF PART: Let $\sigma \in$ ENV and assume that P eq Q, that is, by Definition 9.3, $\mathcal{T}\sigma\,[P] = \mathcal{T}\sigma\,[Q]$. Then

$$\begin{aligned}
&(P \Rightarrow Q)\wedge(Q\Rightarrow P) \\
&= (\mathcal{T}\sigma\,[P] \Rightarrow \mathcal{T}\sigma\,[Q]) \wedge (\mathcal{T}\sigma\,[Q] \Rightarrow \mathcal{T}\sigma\,[P]) && \text{(Defn. of } \mathcal{T}) \\
&= (\mathcal{T}\sigma\,[P] \Rightarrow \mathcal{T}\sigma\,[P]) \wedge (\mathcal{T}\sigma\,[P] \Rightarrow \mathcal{T}\sigma\,[P]) && \text{(Assumption that } \mathcal{T}\sigma\,[P] = \mathcal{T}\sigma\,[Q]) \\
&= T && \text{(Meanings of } \Rightarrow, \wedge \text{ in } \mathcal{B})
\end{aligned}$$

Thus, by Defintion 9.2, $(P\Rightarrow Q)\wedge(Q\Rightarrow P)$ is a tautology.

ONLY-IF PART: Let $\sigma \in$ ENV and assume that $(P \Rightarrow Q)\wedge(Q\Rightarrow P)$ is a tautology. Hence

$$\begin{aligned}
T &= \mathcal{T}\sigma\,[(P \Rightarrow Q)\wedge(Q\Rightarrow P)] \\
&= (\mathcal{T}\sigma\,[P] \Rightarrow \mathcal{T}\sigma\,[Q]) \wedge (\mathcal{T}\sigma\,[Q] \Rightarrow \mathcal{T}\sigma\,[P]) \qquad \text{(Defn. of } \mathcal{T})
\end{aligned}$$

For operations $\Rightarrow$ and $\wedge$ in $\mathcal{B}$, this equality can hold only if $\mathcal{T}\sigma\,[P] = \mathcal{T}\sigma\,[Q]$. Therefore, by Definition 9.3, P *eq* Q.

∎

Recall (Definition **??**) that a *substitution*,

$$W\left[{}^{p,\ \ q,\ldots}_{P,\ \ Q\ldots}\right]$$

denotes a recursively defined function $\mathcal{S}$: *WFF* $\rightarrow$ *WFF* simultaneously replacing propositional variables p, q, etc. with *WFF*s P, Q, etc., respectively. For simplicity, let us restrict ourselves to single-variable substitutions, and write

$$W^p_Q \text{ instead of } W\left[{}^{p}_{Q}\right]$$

and use $\mathcal{S}\,[P]$ for the result of a substitution.

The following three results verify the fundamental rules for logical manipulations involving substitution.

Lemma 9.2 (Substitution Lemma) *Let $\mathcal{S}$ be a substitution and σ an environment. Define an new environment σ' as follows:*

$$\sigma'(v) = \mathcal{T}\sigma\,[\mathcal{S}(v)] \text{ for } v \in IVS$$

Then for all WFFs P,

$$\mathcal{T}\sigma'\,[P] = \mathcal{T}\sigma\,[\mathcal{S}\,[P]]$$

PROOF: The proof is a straightforward structural induction on *WFF*. The inductive cases hold because the operations are functions. The interesting base case is the one for a variable $v \in IVS$. In that case, we have

$$\mathcal{T}\sigma'\,[v] = \mathcal{T}\sigma\,[\mathcal{S}\,[p]]$$

which is exactly what we need to make the Theorem true.

∎

The Substitution Lemma states that for the language of propositions, a call-by-value style evaluation—in which variables are bound to expressions' values—is equivalent to a call-by-name style evaluation. This is an exact equivalence because there is no form of looping in the language.

More significantly, the lemma says that our notion of substitution interacts well with our notion of evaluation. For example,

Theorem 9.3 (Tautology Theorem) *Let P be a WFF and $\mathcal{S}$ a substitution, If P is a tautology, then so is $\mathcal{S}[P]$.*

PROOF: Apply the definition of *tautology* and the Substitution Lemma. The details are left as Exercise 2.

∎

For example, the formula

$$Q \equiv \texttt{(p} \vee \texttt{q} \wedge \texttt{(p} \Rightarrow \texttt{r))} \vee \neg\texttt{(p} \vee \texttt{q} \wedge \texttt{(p} \Rightarrow \texttt{r))}$$

is a tautology because $\texttt{p} \vee \neg\texttt{p}$ is a tautology there is a subsitition,

$$\mathcal{S}[\texttt{p}] = \texttt{(p} \vee \texttt{q} \wedge \texttt{(p} \Rightarrow \texttt{r))}$$

under which

$$Q \equiv \mathcal{S}[\texttt{p} \vee \neg\texttt{p}]$$

From this point on, we will again use the triple-equals symbol '≡', as above, to refer to the equality of *words*.

Theorem 9.3 gives us one way to abbreviate the analysis of *WFF*s. The next theorem states that we can analyze *WFF schemes* as well as individual *WFF*s.

Theorem 9.4 (Substitution Theorem) *If $P \,\mathrm{eq}\, Q$ then for any substitution $\mathcal{S}$, $\mathcal{S}[P] \,\mathrm{eq}\, \mathcal{S}[Q]$*

PROOF: Use the Substitution Lemma.

∎

Thus, not only is

$$\texttt{p} \vee \texttt{(q} \vee \texttt{r)} \;\mathrm{eq}\; \texttt{(p} \vee \texttt{q)} \vee \texttt{r}$$

but the two schemes, $P \vee (Q \vee R)$ and $(P \vee Q) \vee R$, are equivalent for arbitray sub-*WFF*s P, Q, and R. In fact, we reason about *WFF* schemes far more often than we reason about *WFF* individuals.

The following result is very important to the way we do proofs. It says you can "replace equals with equals" and still preserve equivalence.

Theorem 9.5 (Replacement Theorem) *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be substitutions such that, for all $v \in IVS$, $\mathcal{S}_1(v) \;\mathrm{eq}\; \mathcal{S}_2(v)$. Then for any WFF P,*

$$\mathcal{S}_1[P] \;\mathrm{eq}\; \mathcal{S}_2[P]$$

Copyright © 2008 Steven D. Johnson

PROOF: Let σ be any environment and define σ' to be

$$\sigma'(v) = \mathcal{T}\sigma\,[\mathcal{S}_1[v]]$$

Since $\mathcal{S}_1(v)$ eq $\mathcal{S}_2(v)$, it is also the case that $\sigma'(v) = \mathcal{T}\sigma\,[\mathcal{S}_2[v]]$ for all $v \in IVS$. Using the Substitution Lemma twice, we have

$$\mathcal{T}\sigma\,[\mathcal{S}_1[P]] = \mathcal{T}\sigma'\,[P] = \mathcal{T}\sigma\,[\mathcal{S}_2[P]]$$

as desired. ∎

For example, `p⇒q` eq `¬q ⇒ ¬p` so

`(q ⇒ p) ∧ (p ⇒ q)` eq `(q ⇒ p) ∧ (¬q ⇒ ¬p)`

under the substitutions

v	$\mathcal{S}_1[v]$	$\mathcal{S}_2[v]$
p	p	p
q	q	q
r	p ⇒ q	¬q ⇒ ¬p

These results about substitution and replacement do not depend in any fundamental way on the data type $\mathcal{B}$. Exercise 3 asks you to generalize the theorems in this section to an arbitrary data type.

Exercises 9.1

1. The proof of Lemma 9.2 does not show any of the inductive cases, claiming that they are "straightforward." Give the details of the cases for:

(a) If $P, Q \in \mathit{WFF}$ then so is $P{\wedge}Q$.

(b) If $P, Q \in \mathit{WFF}$ then so is $P{\vee}Q$.

(c) If $P, Q \in \mathit{WFF}$ then so is $P{\Rightarrow}Q$.

(d) If $P \in \mathit{WFF}$ then so is $\neg P$.

2. Prove the Tautology Theorem, Theorem 9.3.

3. Let TERM be the language of *terms* over data type

$$\mathcal{A} = \langle A; f_1, \ldots, f_n;\ p_1, \ldots, p_m;\ c_1, \ldots, c_r\rangle$$

and define the notion of *term equivalence* to be:

> *For terms N, M ∈* TERM*, N is equivalent to M, written N eq M, means that for all σ ∈* ENV*,* $\mathcal{T}\sigma[N] = \mathcal{T}\sigma[M]$.

(a) Prove a version of the Substitution Lemma (Lemma 9.2) for TERM.

(b) Prove a version of the Substitution Theorem (Theorem 9.4) for TERM.

(c) Prove a version of the Replacement Theorem (Theorem 9.5) for TERM.

9.2 Formal Proofs

Theorems 9.2, 9.4 and 9.5 lay the foundation for logical reasoning at a purely syntactic level.

Definition 9.4 (Terminology) *Let $P \in WFF$ and $\mathcal{S}$ a substitution. The term*

$$\mathcal{S}\,[P] \equiv P\left[\begin{smallmatrix} Q_1, & \cdots, & Q_n \\ p_1, & \cdots, & p_n \end{smallmatrix}\right]$$

*is called an $\mathcal{S}$-*instance of P*.*

If $\mathcal{S}$ is determined by the discussion context, we simply say *instance* rather than $\mathcal{S}$-instance.

We have used lower-case variables, p, q, r, etc., for individual variable symbols and upper-case variables P, Q, R, etc., for WFFs. The expression

$$P \wedge (P \Rightarrow Q)) \Rightarrow Q$$

denotes an instance of

$$p \wedge (p \Rightarrow q)) \Rightarrow q$$

substituting P for p and Q for q.

Observe also that an instance of a substitution is an instance, for example,

$P\left[\begin{smallmatrix} R \\ p \end{smallmatrix}\right] \wedge (P\left[\begin{smallmatrix} R \\ p \end{smallmatrix}\right] \Rightarrow Q\left[\begin{smallmatrix} W \\ r \end{smallmatrix}\right]) \Rightarrow Q\left[\begin{smallmatrix} W \\ r \end{smallmatrix}\right]$ is also an instance of $p \wedge (p \Rightarrow q)) \Rightarrow q$

Let $\Gamma \subseteq WFF$ be a set of propositional formulas In the definition below, formulas in Γ are the starting assumptions of a logical argument[1]

Definition 9.5 *Given a set of* assumptions*, $\Gamma \subset WFF$, and a set of* inference rules*, $IR\colon WFF^k \overset{p}{\to} WFF$, A* formal proof *is a sequence*

$$D = \langle P_1,\, P_2,\, \ldots,\, P_n \rangle$$

In which each P_j is either

(a) *An instance of $A \in \Gamma$, or*

(b) *The an instance $IR(Q_1, \ldots, Q_k)$ of some inference rule <u>and</u> each $Q_1, \ldots, Q_k$ is occurs prior to P_j in $\mathcal{D}$.*

Definition 9.6 *If a formal proof $\langle P_1, P_2, \ldots, P_n \rangle$ from assumptions Γ exists, we say that the last formula, P_n, is* deducible from Γ*. A deducible formula is called a* theorem in formal system $\langle \Gamma, IR \rangle$*.*

The proof-sequence $D = \langle P_1, P_2, \ldots, P_n, Q \rangle$ can be thought of as a construction sequence (in the sense of Section 7.4) for Q from the base set Γ. Note however, that the IR's are typically *partial* functions. This generalization has no impact on the basic relationships among inductively defined sets, structural induction principles and recursive definitions developed in Chapter 4.

[1] It is traditional to use capital Greek letters in this context. Later the variable Δ will be introduced for the same purpose.

Example

Ex 9.1 [2] *There are just three assumptions and one inference rule:*

$$\begin{array}{lll} \Gamma & = \{\ P \Rightarrow (Q \Rightarrow P), & (A1) \\ & \quad (P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R)), & (A2) \\ & \quad (\neg Q \Rightarrow \neg P) \Rightarrow ((\neg Q \Rightarrow P) \Rightarrow Q) & (A3) \\ & \ \} & \\ MP & :\ p,\ p \Rightarrow q \mapsto q & (MP) \end{array}$$

Show that $p \Rightarrow p$ *is deducible from* Γ *and* MP.

(1)	$(p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))$	$\mathrm{A2}\left[{}^{p,\ p,\ p}_{P,\ Q,\ R}\right]$
(2)	$p \Rightarrow ((p \Rightarrow p) \Rightarrow p)$	$\mathrm{A1}\left[{}^{p,\ p}_{P,\ Q}\right]$
(3)	$(p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p)$	$MP[\,(2),\ (1)\,]\,]$
(4)	$p \Rightarrow (p \Rightarrow p)$	$\mathrm{A1}\left[{}^{p,\ p}_{P,\ Q}\right]$
(5)	$p \Rightarrow p$	$MP[\,(4),\ (3)\,]\,]$

As may be evident from Example 9.1, finding a formal proof is essentially a goal-directed activity; starting with the conclusion and applies MP "backwards" to find an applicable assumption. The justification of each step explicity gives the substitution from which each instance is is derived. Since this information is determined from the instance P_i and the assumption $A \in \Gamma$, it is usually omitted in the justification, which simply specifies which assumption is being used.

Example

Ex 9.2 *Using the same* Γ *and* MP *as in Example 9.1, show that* $(\neg p \Rightarrow p) \Rightarrow p$ *is deducible.*

(1)	$(\neg p \Rightarrow ((\neg p \Rightarrow \neg p) \Rightarrow \neg p))$ $\Rightarrow ((\neg p \Rightarrow (\neg p \Rightarrow \neg p)) \Rightarrow (\neg p \Rightarrow \neg p))$	(A2)
(2)	$\neg p \Rightarrow ((\neg p \Rightarrow \neg p) \Rightarrow \neg p)$	(A1)
(3)	$(\neg p \Rightarrow (\neg p \Rightarrow \neg p)) \Rightarrow (\neg p \Rightarrow \neg p)$	$MP[\,(2),\ (1)\,]$
(4)	$\neg p \Rightarrow (\neg p \Rightarrow \neg p)$	(A1)
(5)	$\neg p \Rightarrow \neg p$	$MP[\,(4),\ (3)\,]$
(6)	$(\neg p \Rightarrow \neg p) \Rightarrow ((\neg p \Rightarrow \neg p) \Rightarrow p)$	(A3)
(7)	$(\neg p \Rightarrow p) \Rightarrow p$	$MP[\,(5),\ (6)\,]$

[2] This example is taken from the book *A First Course in Formal Logic and its Applications in Computer Science,* by R.D. Dowsing, V.J. Rayward-Smith and C.C. Walter, Blackwell Scientific Publications, Oxford, 1986, pp.26–29.

Copyright © 2008 Steven D. Johnson

Verify that Steps 1, 2, 4 and 6 properly instantiate the assumptions used in each step, for example,

$$(1) \equiv \mathrm{A2}\left[\begin{smallmatrix}[\neg p], & [\neg p \Rightarrow \neg p], & [\neg p] \\ P, & Q, & R\end{smallmatrix}\right]$$

Steps 1–5 in Example 9.1 are the same as in the proof in Example 9.2, except that they are applied to different substitution instances. We would like for existing proofs to be "reusable," so let us add We should add to Definition 9.5 a provision for using already-proven results:

(a) Any instance of assumption $A \in \Gamma$.

(b) Any instance $IR(Q_1, \ldots, Q_k)$ of some inference rule *provided* each $Q_1, \ldots, Q_k$ is occurs prior to P_j in $\mathcal{D}$.

(c) Any instance of a theorem $T \in \mathit{WFF}$ that is deducible from Γ and IR.

9.2.1 Deducability and Validity

Definition 9.7 *Suppose* Γ *is any set of assumptions,* R *any set of inference rules, and* $Q \in \mathit{WFF}$ *any formula. If* Q *is* deducible *from* Γ *and* R*, we say* Q *is* provable*, written*

$$\Gamma \vdash_R Q$$

If the set of inference rules is fixed by the context of the discussion, the subscript R is omitted. If $\Gamma = \emptyset$, it too is dropped and we just write $\vdash Q$.

Definition 9.8 Q *is a* consequence *of* $\Gamma = \{a_1, \ldots, a_n\}$*, if, for all* $\sigma \in$ ENV*,*

$$\big(\mathcal{T}\sigma\,[a_1] \wedge \cdots \wedge \mathcal{T}\sigma\,[a_n]\big) \Rightarrow \mathcal{T}\sigma\,[Q]$$

The notation for consequence *is*

$$\Gamma \models Q$$

'$\models$' gives us a semantic notion of consequence; '$\vdash$' is a syntactic notion of provability. Naturally, we would want our formal proofs to be valid, that is, if one can prove a formula P, it should be the case that the assumptions *imply* the truth of P. We might also want our proof system to powerful, that is, if P is a tautology, there should exist a formal proof of P. These two qualities specified in the next two definitions.

Definition 9.9 *A proof system* $\langle\Gamma, R\rangle$ *is* sound *when* $\Gamma \vdash_R Q$ *implies* $\Gamma \models Q$*.*

Definition 9.10 *A proof system* $\langle\Gamma, R\rangle$ *is* complete *when* $\Gamma \models Q$ *implies* $\Gamma \vdash_R Q$

The system used in Examples 9.1 and 9.2 is sound and, perhaps surprisingly, complete.

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

Proposition 9.6 *The formal logic*

$$\begin{array}{lll} \Gamma & = \{ P \Rightarrow (Q \Rightarrow P), & (A1) \\ & \quad (P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R)), & (A2) \\ & \quad (\neg Q \Rightarrow \neg P) \Rightarrow ((\neg Q \Rightarrow P) \Rightarrow Q) & (A3) \\ & \} & \\ MP & : p,\ p \Rightarrow q \mapsto q & (MP) \end{array}$$

Is both sound and complete with respect to propositional logic.

PROOF: We shall not delve into the details of proving either soundness or completeness, except to note:

(a) In this instance and in general, proving *soundness* is a relatively straightforward structural induction involving applications of the Substitution and Replacement Theorems, 9.4 and 9.5.

(b) *Completeness* is harder to prove because the argument must consider all *interpretations*, $\mathcal{I}\colon \mathit{WFF} \rightarrow \{0, 1\}$.

(c) Of course we need to know that all the usual logical operators are accounted for; that is, they can be expressed with '¬' and '⇒':

$$\begin{array}{lcl} P \wedge Q & \text{eq} & \neg(P \Rightarrow \neg Q) \\ P \vee Q & \text{eq} & \neg P \Rightarrow Q \\ P \Leftrightarrow Q & \text{eq} & (P \Rightarrow Q) \wedge (Q \Rightarrow P) \\ & \text{etc.} & \end{array}$$

∎

Proving has to do with *implication*, and the "provability" relation '⊢' differentiates the formal notion of proof from the *operation* '⇒'. The next theorem establishes the semantic connection.

Theorem 9.7 (Deduction Theorem) *If* $\Gamma = \{A_1, \ldots, A_n\}$ *and inference rules* R *form a sound logic, then*

$$\Gamma \vdash_R Q \textit{ implies that } (A_1 \wedge \cdots \wedge A_n) \Rightarrow Q \textit{ is a tautology.}$$

Theorem 9.7 does say that the assumptions are tautologies, only the implication. Γ typically contains formulas of three kinds:

(a) *Axioms* are the basis of reasoning. These should be valid tautologies, true for any instance. Assumptions $A1$, $A2$ and $A3$ in Examples 9.1 and 9.2 of this kind.

Copyright © 2008 Steven D. Johnson

(b) *Premises* are assumptions to a specific theorem. to prove a theorem of the form $\Gamma \vdash (P \wedge Q) \Rightarrow R$, Theorem 9.7 says you can take a shortcut by adding P and Q as assumptions.

$$\Gamma \cup \{P, Q\} \vdash R$$

(c) *Results*. As noted earlier, if $\Gamma \vdash R$, it is valid to add R to Γ.

9.2.2 A More Useful Propositional Calculus

The examples in the previous section use a minimal set of axioms and a single inference rule, claimed to be a sound and complete Propositional Calculus. It is the kind of system one would use to for in-depth studies of logic itself, but it is not useful for *doing* logic. This section specifies a more useful calculus for the latter purpose. We begin by introducing a compact notation for specifying inference rules.

Inference Rule Notation.

As motivation, consider this description of the inference rule *MP*:

$$\frac{\Gamma \vdash (P \Rightarrow Q) \;;\; \Gamma \vdash P}{\Gamma \vdash Q} \quad MP$$

It could be read in two ways:

(i) "If $P \Rightarrow Q$ and P are provable, then Q is provable," *or*

(ii) "If you want to prove Q, it suffices to prove P and $P \Rightarrow Q$."

The capitalized variables P and Q refer to substitution instances of $MP\big[(p \Rightarrow q),\ p\big]$. The formula below the line is a *proof goal* that decomposes into two *proof subgoals*. A generalization of the provablility notation '$\Gamma \vdash_R Q$' replaces the single conclusion Q with a a *set* of *WFF*s, Δ:

$$\Gamma \vdash_R \Delta \qquad \text{meaning} \qquad \boxed{\text{At least one } \gamma \in \Delta \text{ is deducible from } \Gamma.}$$

Our inference rules will have the form

$$\frac{\Gamma' \vdash \Delta' \quad \cdots}{\Gamma \vdash \Delta} \quad name$$

In which the proof-goal below the line decomposes to one or more subgoals above the line, in which formulas may be added or deleted from Γ or Δ. As a further abbreviation, we write

$$\Gamma, P \vdash Q, \Delta \;\text{ rather than }\; \Gamma \cup \{P\} \;\vdash\; \{Q\} \cup \Delta$$

A set of rules called *System G* is shown in Figure 9.1. It has several interesting features:

(a) The Axiom says that if $\Gamma \cap \Delta \neq \emptyset$ then $\Gamma \vdash \Delta$. In other words any instance of $p \Rightarrow p$ is a theorem.

(b) There are two inference rules for each logical operator, one for operators appearing as assumptions, the other dealing with operators appearing in conclusions.

(c) In every rule, the subgoals above the line are "simpler" than the goals below, in the sense that they remove an operator from every subgoal.

(d) The rules are "reversible;" they can be applied in either direction.

Let us use these rules to prove $((p \Rightarrow q) \wedge p \Rightarrow q$. The proof on the left starts with the goal at the bottom and applies inference rules until the subgoals are axioms. The same proof is shown on the right in the form of a deduction.

$$\text{Ax:}\ \frac{}{p, q \vdash q}\ (1) \qquad \text{Ax:}\ \frac{}{p \vdash p, q}\ (2)$$

$$\Rightarrow\vdash:\ \frac{}{(p \Rightarrow q), p \vdash q}\ (3)$$

$$\wedge\vdash:\ \frac{}{(p \Rightarrow q) \wedge p \vdash q}\ (4)$$

$$\vdash\Rightarrow:\ \frac{}{\vdash ((p \Rightarrow q) \wedge p) \Rightarrow q}\ (5)$$

1.	$p, q \vdash q$	Ax.
2.	$p \vdash p, q$	Ax.
3.	$(p \Rightarrow q), p \vdash q$	$\Rightarrow\vdash$
4.	$(p \Rightarrow q) \wedge p \vdash q$	$\wedge\vdash$
5.	$\vdash ((p \Rightarrow q) \wedge p) \Rightarrow q$	$\vdash\Rightarrow$

The fact that each inference rule of System G consumes an operator symbol implies that, eventually, there will be no operators left and hence no more applicable rules. The subgoals ultimately reduce to the form $\{p_1, \ldots, p_k\} \vdash \{q_1, \ldots, q_m\}$. If any $p_i = q_j$ we have reached an axiom; otherwise, there is nothing left to do.

If there are no applicable rules to apply, let us say that deduction *succeeds* if all leaves of the resulting *proof tree* are axioms. Otherwise it *fails*, and we write $\Gamma \nvdash \Delta$.

It is a pleasant fact that, no matter the order that the rules may be applied, either all deductions succeed or they all fail. In other words, there is no risk of choosing the wrong rule.

Theorem 9.8 *System G is sound. If a deduction on* $\Gamma \vdash \Delta$ *succeeds then*

$$\bigwedge_{P \in \Gamma} \Rightarrow \bigvee_{Q \in \Delta}$$

is a tautology. That is, if $\Gamma \vdash \Delta$ *then* $\Gamma \models \Delta$.

PROOF: The proof is a structural induction on *WFF* and reduces to showing that the Axioms of System G are tautologies and the inference rules are sound.

∎

Copyright © 2008 Steven D. Johnson

System G		
Axioms	$\Gamma, P \vdash P, \Delta$ (Ax.)	
	assumption simplifcation	*conclusion simplification*
$\neg$ *rules*	$\neg\vdash: \dfrac{\Gamma \vdash P, \Delta}{\Gamma, [\neg P] \vdash \Delta}$	$\vdash\neg: \dfrac{\Gamma, P \vdash \Delta}{\Gamma \vdash [\neg P], \Delta}$
$\vee$ *rules*	$\vee\vdash:) \dfrac{\Gamma, P \vdash \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, [P \vee Q] \vdash \Delta}$	$\vdash\vee: \dfrac{\Gamma \vdash P, Q, \Delta}{\Gamma \vdash [P \vee Q], \Delta}$
$\wedge$ *rules*	$\wedge\vdash: \dfrac{\Gamma, P, Q \vdash \Delta}{\Gamma, [P \wedge Q] \vdash \Delta}$	$\vdash\wedge: \dfrac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash [P \wedge Q], \Delta}$
$\Rightarrow$ *rules*	$\Rightarrow\vdash: \dfrac{\Gamma \vdash P, \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, [P \Rightarrow Q] \vdash \Delta}$	$\vdash\Rightarrow: \dfrac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash [P \Rightarrow Q], \Delta}$

Figure 9.1: A Propostional Calculus, System G

Theorem 9.9 *If a deduction for* $\Gamma \vdash \Delta$ *fails, then* $\Gamma \not\models \Delta$.

PROOF: Such a deduction contains a leaf $\{p_1, \ldots, p_k\} \vdash \{q_1, \ldots, q_m\}$ for which no $p_i = q_j$. Define an environment σ in which, for all $1 \leq i \leq k$, $\sigma(p_i) = \mathit{true}$ and all $1 \leq j \leq m$, $\sigma(q_i) = \mathit{false}$. Under σ, the original formula is *false* and therefore not a tautology.

∎

It follows from these results that System G is both sound and complete.

Theorem 9.10 $\Gamma \vdash \Delta$ *iff* $\Gamma \models \Delta$.

Exercises 9.2

1. Using the formal system defined for Examples 9.1 and 9.2, give proofs for the following theorems:

(a) $\neg\neg P \Rightarrow P$

(b) $\neg P \Rightarrow (P \Rightarrow Q)$

(c) $(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$

(d) $(Q \Rightarrow P) \Rightarrow (\neg P \Rightarrow \neg Q)$

Copyright © 2008 Steven D. Johnson DRAFT MATERIAL (v08r0) January 26, 2009

2. Using System G, give proofs for the following theorems:

(a) $\neg\neg P \Rightarrow P$

(b) $\neg P \Rightarrow (P \Rightarrow Q)$

(c) $(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$

(d) $(Q \Rightarrow P) \Rightarrow (\neg P \Rightarrow \neg Q)$

9.3 First-order Predicates

The propositional logic gives us a way to reason about pure statements of fact. However, the truth of some sentences depends on the context in which they are made. For example, consider the statement

$$(n \neq 0) \quad \Rightarrow \quad \frac{x}{n} \cdot n = x$$

This is a true sentence in the domain of real or rational numbers. If, however, we are thinking about integers (as we do in computer arithmetic) the law may no longer hold because of round-off errors. Thus, for our purposes, the notion of "context" will be represented by a data type

$$\mathrm{A} = \langle A; f_1, \ldots, f_n;\ p_1, \ldots, p_m;\ c_1, \ldots, c_r \rangle$$

We are going to define a language of *first-order formulas over* $\mathcal{A}$, in which we can make assertions about elements of A.

Definition 9.11 *The language FOF of* first-order formulas over $\mathcal{A}$ *is defined inductively as follows.*

1. *if $\hat{p}$ is an n-place predicate symbol and t_1, ..., t_n are terms, then*

$$\hat{p}(t_1,\ \ldots,\ t_n)$$

is a formula.

2a. *if G and H are formulas, then so are*

(i) $\neg\ G$
(ii) $G\ \wedge\ H$ *(iv)* $G\ \Rightarrow\ H$
(iii) $G\ \vee\ H$ *(v)* $G\ \Leftrightarrow\ H$

2b. *if G is a formula and v is an individual variable symbol, then the following are formulas:*

(i) $\forall\ v\colon\ G$
(ii) $\exists\ v\colon G$

3. *n .e.*

Copyright © 2008 Steven D. Johnson

Implication associates to the right (all other operators may associate in any way), and the precedence in parsing is

$$\textit{(highest)}\quad \binom{\exists}{\forall},\ \neg,\ \wedge,\ \vee,\ \Rightarrow,\ \Leftrightarrow,\quad \textit{(lowest)}$$

Before defining what these formulas mean, let us discuss what they *ought* to mean. In the base case, each term in the formula $\hat{p}(t_1,\ \ldots,\ t_n)$ represents a particular element of the set of values, A, in the data type. So we will need the term evaluator of Chapter 4 to assign a meaning to these formulas.

The rules in 2a are pretty easy. They are just the standard logical combinations of subformulas that we have seen before.

Rule 2b is new and different. How should we interpret the formula $\forall \mathtt{x}\colon G$? You might have said, "this formula is true if G is true for any possible value of $\mathtt{x}$." We have used environments to give values to variables, so we want to phrase this definition in terms of environment.[3] We might say, "formula $\forall \mathtt{x}\colon G$ is true in environment σ if G is true no matter what $\sigma(\mathtt{x})$ is." Let us introduce some notation for this idea.

Definition 9.12 *If σ and σ' are environments and v is an individual variable symbol, then σ and σ' are said to be* equivalent modulo v, *written $\sigma \overset{v}{\simeq} \sigma'$, if they agree everywhere except perhaps at v. In other words,*

$$w \neq v \textit{ implies } \sigma(w) = \sigma'(w)$$

With this bit of notation, we can assign a meaning to first-order formulas.

Definition 9.13 *The function $\mathcal{F}\colon \textsc{env} \times FOF \to A$ is defined over FOF (Definition 9.11 as follows:*

1. $\mathcal{F}\sigma\,[\hat{p}(t_1,\ \ldots,\ t_n)] = p\big(\mathcal{T}\sigma[t_1],\ \ldots,\ \mathcal{T}\sigma[t_n]\big)$

2a.
(i) $\mathcal{F}\sigma\,[\neg G] = \neg\mathcal{F}\sigma\,[G]$
(ii) $\mathcal{F}\sigma\,[G \wedge H] = \mathcal{F}\sigma\,[G]\ \wedge\ \mathcal{F}\sigma\,[H]$
(iii) $\mathcal{F}\sigma\,[G \vee H] = \mathcal{F}\sigma\,[G]\ \vee\ \mathcal{F}\sigma\,[H]$
(iv) $\mathcal{F}\sigma\,[G \Rightarrow H] = \mathcal{F}\sigma\,[G]\ \Rightarrow\ \mathcal{F}\sigma\,[H]$
(v) $\mathcal{F}\sigma\,[G \Leftrightarrow H] = \mathcal{F}\sigma\,[G]\ \Leftrightarrow\ \mathcal{F}\sigma\,[H]$

2b. $\mathcal{F}\sigma\,[\exists v\colon G] = T$ *iff for some* $\sigma' \overset{v}{\simeq} \sigma$, $\mathcal{F}\sigma'\,[G] = T$
$\mathcal{F}\sigma\,[\forall v\colon G] = T$ *iff for every* $\sigma' \overset{v}{\simeq} \sigma$, $\mathcal{F}\sigma'\,[G] = T$

Consider the formula

$$\exists q\colon a = bq$$

[3] Textbooks in logic often use the term "interpretation" rather than "environment" in this context.

It says, "b divides a evenly." In other words, it is a statement about a and b but *not* about the quantified variable q. Let us give this formula a name and acknowledge that it depends on two parameters:

$$\textsc{Divides}(A, B) \ \equiv\ \exists q\colon A = Bq$$

The triple-equal symbol '$\equiv$' says that we are defining a *formula* called DIVIDES in terms of two sub-formulas A and B. There is, of course, a corresponding *predicate*,

$$divides(x, y) \ = \ \begin{cases} T \text{ if } x = yq \text{ for some positive integer } q \\ F \text{ otherwise} \end{cases}$$

giving the meaning of DIVIDES.

As we have just seen, first-order formulas denote predicates on their unquantified variables. A variable that occurs unquantified in a formula G is said to be *free in* G. A variable that occurs quantified in G is said to be *bound in* G. It is possible for different occurances of a variable to be both free and bound in the same formula.

Definition 9.14 *If P is a first-order formula, the* free variables *of P is the subset $FREE[P] \subseteq IVS$ defined as follows.*

1. *for a term t, $FREE[t] = \{v \in IVS \mid v \text{ occurs in } t\}$*

2a. *$FREE[\neg G] = FREE[G]$*
$FREE[G \ \wedge \ H] = FREE[G] \cup FREE[H]$
and similarly for the other logical connectives

2b. *$FREE[\forall v\colon G] = FREE[G] \setminus \{v\}$*
$FREE[\exists v\colon G] = FREE[G] \setminus \{v\}$

Of course, we want and expect the laws of substitutivity to hold for first-order formulas. However we must redefine what substitution means for quantified formulas.

First, substitution should not affect quantified variables or their bound occurances. Second, we must take care to avoid "capturing" a free variable in the process of substitution. For example, if we try to subsititute $5q$ for A in the formula DIVIDES(A, B), we get

$$\exists q\colon 5q = Bq$$

Whether or not this formula is true, it does not mean "B divides $5q$ evenly. What we want is something like

$$\exists x\colon 5q = Bx$$

Thus, to assure that free variable capture does not occur, we can systematically replace the quantified variable symbols of our formula with variables that are unused anywhere else.

$$\exists v\colon P \mapsto \exists v'.P\left[{}^{v'}_{v}\right]$$

In the following definition, it is assumed that this has already been done.

Definition 9.15 *The substitution mapping* $\mathcal{S}\colon IVS \to FOF$ *extends to a substitution function* $\mathcal{S}^*\colon \mathcal{P}(IVS) \times FOF^* \to FOF$*, on the subset* FOF^* *of first-order formulas whose bound variables are not free in any* $\mathcal{S}(v)$*.* $\mathcal{S}^*$ *is defined as follows:*

1. *For terms,*

$$\begin{array}{rl} (i) & \text{for constant symbols, } \mathcal{S}^*B[\hat{c}] = \hat{c} \\ (ii) & \text{for variable symbols, } \mathcal{S}^*B[v] = \begin{cases} v & \text{if } v \in B \\ \mathcal{S}^*(v) & \text{if } v \notin B \end{cases} \\ (iii) & \mathcal{S}^*B[\hat{f}(t_1, \ldots, t_n)] \\ & = \hat{f}(\mathcal{S}^*B[t_1], \ldots, \mathcal{S}^*B[t_1]) \end{array}$$

2a. *if* G *and* H *are formulas,*

$$\begin{array}{rl} (i) & \mathcal{S}^*B[\neg G] = \neg\, \mathcal{S}^*B[G] \\ (ii) & \mathcal{S}^*B[G \wedge H] = \mathcal{S}^*B[G] \wedge \mathcal{S}^*B[H] \\ (iii\text{–}v) & \text{similarly for } \vee,\ \Rightarrow, \text{ and } \Leftrightarrow \end{array}$$

2b. *if* G *is a formula and* v *is an individual variable symbol, then*

$$\begin{array}{rl} (i) & \mathcal{S}^*B[\forall\, v\colon\ G] = \forall\, v\colon \mathcal{S}^*B'[G] \\ (ii) & \mathcal{S}^*B[\exists\, v\colon G] = \exists\, v\colon \mathcal{S}^*B'[G] \end{array}$$

where $B' = B \setminus \{v\}$*.*

Thus, in the "body" of a quantified formula, substitution $\mathcal{S}$ is applied to every variable *except* the one bound by the quantifier.

Recall that

$$\textsc{Divides}(A, B) \;\equiv\; \exists q\colon A = Bq$$

We ran into a substitution problem in the formula $\textsc{Divides}(5q, B)$. To solve this problem, the quantified variable q is first changed to an unused variable symbol; then $5q$ is substituted for A:

$$\exists q\colon A = Bq\left[{}^{5q}_{A}\right] \;\mapsto\; \exists z\colon A = Bz\left[{}^{5q}_{A}\right] \;\mapsto\; \exists z\colon 5q = Bz$$

With this modification to substitution, the Substitution Lemma (9.2), Tautology Theorem (9.3), Substitution Theorem (9.4) and Replacement Theorem (9.5) extend to *FOF*.

9.4 Predicate Calculus

The previous section lays the groundwork for a formal system for reasoning in *FOF*, which adds quantified formulas to *WFF*. Inference rules of two kinds are needed:

(a) Rules for equality, so that one can reason about objects in the abstract type $\mathcal{A}$. Figure 9.2 introduces equalities that are valid for the type $\mathcal{A}$. Typically, these equalities are specified as *identities* for the type (e.g. the Boolean Identities). The inference rule says, in essence, that if $N = M$ is valid for data type $\mathcal{A}$ ($\mathcal{A} \models N = M$, then replacement of N by M is valid in a proof.

(b) Quantifier rules, both for "getting inside" quantified formulas and for introducing quantified formulas in the conclusions. In Figure 9.3, $\exists$-elimination and $\forall$-introduction involve introducing a new variable, unused anywhere else, to serve as a "dummy constant."

  (i) If $\exists v\colon P[v]$ is an assumption, then a name v' is introduced to represent one of the values for which P is true. Conversely, if $P[v']$ is deducible for an *arbitrary* $v' \in A$, the one may conclude $\forall v\colon P(v)$.

  (ii) If $\forall v\colon P(v)$ is an assumption, then $P(t)$ may be concluded for any *term*, t. And if $P(t)$ is deducible, so is $(\exists v\colon P[v]$.

Exercises 9.4

1. Let $P(x)$ be and assertion about x. Write a First-Order Formula that says, "There is a unique x such that $P(x)$."

2. Write a First-Order Formula stating the Principla of Induction for Nat.

3. Let $P(x)$ and $Q(x)$ be an assertions about x. Prove the following in System G.

  (a) $\forall x\colon P(x) \Rightarrow P(a)$

  (b) $(\forall x\colon P(x) \Rightarrow Q(x) \wedge P(a) \Rightarrow Q(a)$

  (c) $\forall x\colon P(x) \Rightarrow \exists y\colon P(y)$

  (d) $\forall x\colon P(x) \vee \exists y\colon \neg P(y)$

  (e) $\neg\forall x\colon P(x) \Leftrightarrow \exists x\colon \neg P(x)$

  (f) $\neg\exists x\colon P(x) \Leftrightarrow \forall x\colon \neg P(x)$

  (g) $\forall x\colon P(x) \wedge \forall x\colon Q(x) \Leftrightarrow \forall x\colon P(x) \wedge Q(x)$

  (h) $\exists x\colon P(x) \vee \exists x\colon Q(x) \Leftrightarrow \exists x\colon P(x) \vee Q(x)$

Copyright © 2008 Steven D. Johnson

System G	
Axioms	$\vdash N = N$
equality rules	$\dfrac{\Gamma, [N = M], P_v^N \vdash Q_v^N, \Delta}{\Gamma, [N = M], P_v^M \vdash Q_v^M, \Delta}$

Figure 9.2: Equality rules for System G

System G						
		quantifier elimination			*quantifier introduction*	
$\exists$*rules*	$\exists\vdash$:	$\dfrac{\Gamma, P[v'] \vdash \Delta}{\Gamma, \exists v\colon P[v] \vdash \Delta}$	$v' \notin \text{FREE}(\Gamma \cup \Delta)$	$\vdash\exists$:	$\dfrac{\Gamma \vdash P(t), \Delta}{\Gamma \vdash \exists v\colon P(v), \Delta}$	$t \in \text{TERM}$
$\forall$*rules*	$\forall\vdash$:	$\dfrac{\Gamma, P[t] \vdash \Delta}{\Gamma, \forall v\colon P[v] \vdash \Delta}$	$t \in \text{TERM}$	$\vdash\forall$:	$\dfrac{\Gamma \vdash P(v'), \Delta}{\Gamma \vdash \forall v\colon P(v)\Delta}$	$v' \notin \text{FREE}(\Gamma \cup \Delta)$

Figure 9.3: Quantifier rules for System G

Chapter 10

Proving Programs

In Chapter 8 we studied a simple programming language of recursive expressions. In this chapter we apply the same techniques to the sequential language STMT introduced in Chapter 1.3. STMT is a language of assignments and commands represents a family of languages that also contains Java, C, C#, and many others. The reason becomes apparent as one learns about computer architecture. Sequential languages reflect the way computer processors operate.

Our goal in this chapter is to take advantage of our language-specification techniques in devising a system not just for modeling a language, but also for reasoning about programs, specifically for proving them correct. We will do this by showing that *programs* and *assertions* interact in ways that can be precisely described by extending the logical formalism System G developed in Chapter 9. In this way we can always reduce an assertion about program correctness to a purely logical formula. If this formula it true, the program is correct.

10.1 The Language of Statements

Our programming language uses predicate formulas to express tests in conditionals and loops. However, we shall restrict these to be simple predicates, involving logical combinations of primitive tests, but no quantifiers. A test like

> `if` *there exists an* x *such that* $P[x]$
> `then` *Command A*
> `else` *Command B*

would, in general, involve some kind of loop. We don't want our language to have such hidden loops.

Definition 10.1 *The set QFF of* quantifier free first-order formulas *is that subset of FOF whose formulas do not contain the quantifier symbols '*$\forall$*' and '*$\exists$*.'*

Our definition of the Language of Statements uses Backus-Naur form (Section 8.5).

Definition 10.2 *The language of* statements over data type $\mathcal{A}$ *is defined inductively as follows:*

$$\begin{array}{lcll} \langle\text{STMT}\rangle & ::= & \langle\text{IVS}\rangle \texttt{ := } \langle\text{TERM}\rangle & \text{(assignment)} \\ & \mid & \texttt{begin } \langle\text{STMT}\rangle \texttt{ ; } \langle\text{STMT}\rangle \texttt{ end} & \text{(compound)} \\ & \mid & \texttt{if } \langle\text{QFF}\rangle \texttt{ then } \langle\text{STMT}\rangle \texttt{ else } \langle\text{STMT}\rangle & \text{(conditional)} \\ & \mid & \texttt{while } \langle\text{QFF}\rangle \texttt{ do } \langle\text{STMT}\rangle & \text{(repetition)} \end{array}$$

10.1.1 Operational Interpretation of Statements

In our earlier language of expressions, programs reflected the notion of a function, taking values in the form of actual parameters and producing values. Recall that the interpretation function,

$$\mathcal{E}_\delta \colon \text{ENV} \times \mathbf{E} \to A$$

The language of statements, in contrast, is more reflective of the architecture of a computer. It expresses computation in terms of *assignment*: the recording of information in a memory. Or mathematical interpretation of statements should bear this out. A program starts with an initial memory, runs for a while, and then stops, leaving its results in an updated memory. Thus, the interpretation function maps from environments to environments.

Definition 10.3 *The interpretation of statements is given by the partial function*

$$\mathcal{M} \colon \text{ENV} \times \text{STMT} \to \text{ENV}$$

which is defined as follows:

$$(a) \quad \mathcal{M}\sigma\,[v \texttt{ := } t] = \sigma \setminus \{(v, \sigma(v)\} \cup \{(v, \mathcal{E}\sigma[t]\}$$

$$(b) \quad \mathcal{M}\sigma\,[\texttt{begin } S_1 \texttt{ ; } S_2 \texttt{ end}] = \mathcal{M}\sigma'\,[S_2] \text{ where } \sigma' = \mathcal{M}\sigma\,[S_1]$$

$$(c) \quad \mathcal{M}\sigma\,[\texttt{if } Q \texttt{ then } S_1 \texttt{ else} S_2] = \begin{cases} \mathcal{M}\sigma\,[S_1] & \text{if } \mathcal{I}\sigma[Q] = \mathit{true} \\ \mathcal{M}\sigma\,[S_2] & \text{if } \mathcal{I}\sigma[Q] = \mathit{false} \end{cases}$$

$$(d) \quad \mathcal{M}\sigma\,[\texttt{while } Q \texttt{ do } S] = \begin{cases} \quad \sigma, & \text{if } \mathcal{I}\sigma[Q] = \mathit{false} \\ \mathcal{M}\sigma'\,[\texttt{while } Q \texttt{ do } S] & \text{if } \mathcal{I}\sigma[Q] = \mathit{true} \\ \mathit{where}\ \sigma' = \mathcal{M}\sigma\,[S]\,, & \end{cases}$$

The interpretation of assignment says to take the function σ, and replace the ordered pair for program variable v with one that binds v to the value of t.

The compound-statement interpretations says, execute statement S_1, and then execute S_2 using this resulting memory, σ'.

Copyright © 2008 Steven D. Johnson

The rule for `while`-statements, unlike all the other rules, does not reduce interpretation to some proper sub-statement. We can infer from this that interpretation of certain programs does not terminate.

We will not use this operational interpretation very much, but now that it has been defined, let us prove an interesting fact about compound statements.

Proposition 10.1 *The compound operator, '`;`' is associative in the sense that for all environments σ and statements S_1, S_2, and S_3,*

$$\begin{aligned} &\mathcal{M}\sigma\,[\texttt{begin}\ S_1\ \texttt{;}\ \texttt{begin}\ S_2\ \texttt{;}\ S_3\ \texttt{end}\ \texttt{end}] \\ = &\mathcal{M}\sigma\,[\texttt{begin}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \texttt{;}\ S_3\ \texttt{end}] \end{aligned}$$

PROOF: Apply Definition 10.3. ∎

Thus, it is not ambiguous to write

$$\texttt{begin}\ S_1\texttt{;}\ S_2\texttt{;}\ S_3\ \texttt{end}$$

because it doesn't matter how `begin`-`ends` are associated.

10.1.2 Axiomatic Interpretation of Statements

The operational interpretations of statements tells us what programs do, but as programmers we are perhaps more interested in knowing just what we can say a particular program. Our goal in this section is to develop an alternative formalism for this purpose.

Definition 10.4 *Let P and Q be first order formulas and let S be a word in the language of statements. The* program correctness assertion

$$\texttt{\{}P\texttt{\}}\ S\ \texttt{\{}Q\texttt{\}}$$

Is a predicate which is true iff for all $\sigma \in$ ENV,

$$\mathcal{I}\sigma[P] \text{ implies } \mathcal{I}\sigma'[Q] \text{ where } \sigma' = \mathcal{M}\sigma\,[S]$$

We call formula P a *precondition* of statement S; and we call formula Q a *postcondition.*

In words, if property P holds for the initial memory, then after S executes, property Q holds for the final memory. It is important to notice the implicit assumption that S terminates.

10.1.3 Reasoning Rules for Statements

Here is a fundamentally important result.

Proposition 10.2 *For all environments σ, the partial correctness assertion*

$$\texttt{\{}P\texttt{\}}\ v := t\ \texttt{\{}Q\texttt{\}}$$

holds iff

$$P \Rightarrow Q^t_v$$

PROOF: Since substitution is performed on just one variable, we will write Q_v^t instead of $Q\left[{t \atop v}\right]$. According to Definition 10.4, {P} v := t {Q} holds iff

$$\mathcal{I}\sigma[P] \text{ implies } \mathcal{I}\sigma'[Q]$$

where $\sigma' = \mathcal{M}\sigma[v := t]$. By Definition 10.3, σ' can be defined as

$$\sigma'(w) = \begin{cases} \mathcal{E}\sigma[t] & \text{if } w = v \\ \sigma w & \text{if } w \neq v \end{cases}$$

By the (version of the) Substitution Lemma (for first-order formulas), $\mathcal{I}\sigma'[Q] = \mathcal{I}\sigma[Q_v^t]$ so the implication above holds whenever

$$P \Rightarrow Q_v^t$$

is a tautology.

∎

We will express facts like this as *inference rules*, just as we did with logic in the previous chapter. In fact, these rules simply extend System G from that chapter with rules to reason about programs.

$$\frac{P \Rightarrow Q_v^t}{\{P\}\ v\ \texttt{:=}\ t\ \{Q\}} \quad \text{ASSIGNMENT RULE}$$

One should read such a rule like this:

> *If you want to prove* $\{P\}\ v\ \texttt{:=}\ t\ \{Q\}$*, then it suffices to show that* $P \Rightarrow Q_v^t$.

In general, there may be more than one statement above the line. What lies below the line represents a goal hat we are trying to prove. What lies above the line tell us what fact, or set of facts in general, that we must show in order to establish the goal.

Example

Ex 10.1 *Prove:* $\{z + xy - y = AB\}\ x\ \texttt{:=}\ x - 1\ \{z + xy = AB\}$

Performing the substitution we get

$$z + zy - y = AB \quad \Rightarrow \quad z + (x - 1)y = AB$$

But this is clearly true, since $z + (x - 1)y = z + xy - y$.

There is a reasoning rule for each phrase-type of the language of statements. These are shown in Figure 10.1 and discussed individually below. The assignment rule has already been discussed.

10.1.4 The Compound Rule

The rule

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ \texttt{begin}\ S_1\ ;\ S_2\ \texttt{end}\ \{R\}} \qquad \text{COMPOUND RULE}$$

says that you can break down the proof of a compound statement by finding an *intermediate assertion* to place between them. As we shall see in the next section it is never a problem to determine what this assertion should be.

Example

Ex 10.2 *Prove:* $\{z + xy = A \wedge y > 0\}$
$\texttt{begin}\ x := x - 1\ ;\ z := z + y\ \texttt{end}$
$\{z + xy = A\}$

Let intermediate assertion $Q \equiv (z + y) + xy = A$. You may be able to see shy this Q was chosen, but if not, it will become clear in a moment. The compound rule says that to prove the desired program correctness assertion, it suffices to prove the following two assertions:

$$\{z + xy = A \wedge y > 0\}\ x := x - 1Q\ \{$$

$\}$ and

$$\{Q\}\ GETSzz + y\ \{z + xy = A\}$$

The second of these is trivial. By the assignment rule, it reduces to

$$Q\ IMP\big(z + xy = A\big)_z^{z+y}$$

After performing the substitution we get the tautology

$$(z + y) + xy = A \quad \Rightarrow \quad (z + y) + xy = A$$

To prove the first correctness assertion, we again use the assignment rule. We are to assume that $z + xy = A$ and $y > 0$ and prove Q_x^{x-1}. After performing the substitution we can derive

$$(z + y) + (x - 1)y = z + y + xy - y = z + zy = A$$

by our assumption of the precondition.

10.1.5 The Conditional Rule

The rule

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}} \qquad \text{CONDITIONAL RULE}$$

says that you can argue separately about the two branches, while using the fact that the test B succeeded, in the case of the `then`-part, or failed, in the case of the `else` part.

10.1.6 The Repetition Rule

The rule

$$\frac{\{P \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{I \wedge \neg B\}} \quad \textsc{Repetition Rule}$$

Is a formal statement of Theorem 4.2 from Chapter 4.

10.1.7 The Relaxation Rule

The specific assertions within programs do not always match exactly with the subgoals generated by the rules. The relaxation rule says that once we have proven a program correctness assertion, we can replace the precondition by a stronger statement and we can replace the postcondition by a weaker statement:

$$\frac{P' \Rightarrow P \quad \{P\}\ S\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}} \quad \textsc{Relaxation rule}$$

An instance of this rule is the invariant rule for `while` statements:

$$\frac{P \Rightarrow I \quad \{I \wedge B\}\ S\ \{I\} \quad (I \wedge \neg B) \Rightarrow Q}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{Q\}} \quad \textsc{Invariant Rule}$$

10.2 Using the Rules

Given an assertion $\{P\}\ S\ \{Q\}$, how do we go about determining whether it is true? The only thing to do is look for a rule that applies. Since there is just one rule for each kind of phrase in the language, there always just one choice. Since each inference rule reduces an argument about a statement to sub-arguments about its parts, we will eventually reduce the program correctness assertion to a collection of purely logical formulas. These are called the program's *verification conditions*. If the verification conditions are all true, so is the original correctness assertion. In other words, the program is correct.

Unfortunately, knowing which rule to apply is not always enough. consider the assertion

$$\{x = 2\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \{y = 4\}$$

Obviously, we need to use the compound rule, but in order to use it, we must come up with an intermediate assertion Q such that

$$\frac{\{x = 2\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{x = 4\}}{\{x = 2\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \{y = 4\}} \quad \text{COMPOUND}$$

We cannot choose Q at random, but we can almost always find the right formula. Just two of the four rules, repetition and compound, require us to invent an assertion in this way.

For `while` statements, we must find an invariant formula. In general, this is hard to do all but one person: the person who wrote the loop. In fact, a loop's invariant reveals the essence of what the loop is doing. Let us, therefore, *require* the program to contain an invariant for every loop. This is not unreasonable, since, as we have just claimed, with practice you can write down invariant the instant you conceive of the loop. Furthermore, there are often ways to derive the invariant from the surrounding program specification.

From now on, a `while` statement must have the form

$$\texttt{while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S$$

and our reasoning rule becomes

$$\frac{P \Rightarrow I \quad \{I \wedge B\}\ S\ \{I\} \quad (I \wedge \neg B) \Rightarrow Q}{\{P\}\ \texttt{while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S\ \{Q\}} \quad \text{WHILE RULE}$$

The situation for compound statements isn't nearly so bad. For any compound statement we can determine the intermediate assertion by analyzing the component statements.

Technique 10.3 *To prove* $\{P\}$ `begin` S_1 `;` v `:=` t `end` $\{R\}$*, choose* Q *to be* R^t_v.

If we do this we get the following proof structure:

$$\frac{\{P\}\ S_1\ \{R^t_v\} \qquad \dfrac{R^t_v \;\Rightarrow\; R^t_v}{\{R^t_v\}\ v\ \texttt{:=}\ t\ \{R\}}}{\{P\}\ \texttt{begin}\ S_1\ \texttt{;}\ v\ \texttt{:=}\ t\ \texttt{end}\ \{R\}}$$

Thus, the right-hand verification condition reduces to a trivial tautology.

Technique 10.4 *To prove* $\{P\}$ `begin` v `:=` t `;` S_2 `end` $\{R\}$ when P and t do not contain the variable v*, choose* Q *to be* $P \wedge (v = t)$.

In this case the choice of Q trivializes the left-hand verification condition to $P \Rightarrow (P \wedge t = t)$:

$$\dfrac{\dfrac{P \;\Rightarrow\; (P \wedge v = t)^t_v}{\{P\}\ v \texttt{ := } t\ \{P \wedge v = t\}} \qquad \{P \wedge v = t\}\ S_1\ \{R\}}{\{P\}\ \texttt{begin}\ v \texttt{ := } t \texttt{ ; } S_2\ \texttt{end}\ \{R\}}$$

If the precondition P or term t do contain the program variable v, this tactic fails miserably. For example, consider the assertion

$$\{x = 2\}\ \texttt{begin}\ x \texttt{ := } 3 \texttt{ ; } S_2\ \texttt{end}\ \{R\}$$

The intermediate assertion would be $(x = 2) \wedge (x = 3)$, which is contradictory.

Technique 10.5 *To prove*

$$\{P\}\ \texttt{begin}\ S_1 \texttt{ ; while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S_2\ \texttt{end}\ \{R\}$$

choose intermediate assertion Q to be I.

See if you can develop a technique for dealing with conditional statements.

Example

Ex 10.3 *Consider the program correctness assertion*

$\{x = A \wedge y = B \wedge A \geq B\}$
while $y \neq 0$ do $\{(x - y = A - B) \wedge (x \geq y)\}$
  begin
  $x := x - 1;$
  $y := y - 1$
  end
$\{x = A - B\}$

Figure 10.3 develops a proof showing the structure of the proof tree, reducing the assertion to purely logical verification conditions, whose proofs are sketched.

Exercises 10.2

1. $\{x = A \wedge y = B \wedge A \geq B\}$
while $y \neq 0$ do $\{(x - y = A - B) \wedge (x \geq y)\}$
  begin
  $x := x - 1;$
  $y := y - 1$
  end
$\{x = A - B\}$

Copyright © 2008 Steven D. Johnson

2. $\{x = A \land y = B\}$
```
begin
z := 1;
while y ≠ 0 do {z · x^y = A^B}
  begin
  z := z * x;
  y := y − 1
  end
end
{z = A^B}
```

3. $\{x = A \land y = B\}$
```
begin
q := 0;
r := x;
while r ≥ y do {q · y + r = A}
  begin
  q := q + 1;
  r := r − y
  end
end
{(q · y + r = A) ∧ (r < y)}
```

4. $\{x = A\}$
```
begin
z := 1;
while x ≠ 1 do {z · x! = A!}
  begin
  z := z * x;
  x := x − 1
  end
end
{z = A!}  [z is "A factorial."]
```

5. Assume *even?* is a primitive test for even numbers.

```
{x = A ∧ y = B}
begin
z := 1;
while y ≠ 0 do {z · x^y = A^B}
  if even?(y)
    then begin y := y/2; x := x * x end
    else begin y := y − 1; z := z * x end
end
{z = A^B}
```

6. $\{x = A \land y = B\}$

```
begin
z := 1;
while y ≠ 0 do {z · x^y = A^B}
  begin
  while even?(y) do {z · x^y = A^B ∧ y ≠ 0}
    begin
    y := y/2
    x := x * x
    end;
  z := z * x;
  y := y − 1
  end
end
{z = A^B}
```

7. NOTE: $GCD(x, y)$ stands for the greatest common divisor of x and y.

```
{x = A ∧ y = B}
begin
while x ≠ y do
  {GCD(x, y) = GCD(A, B)}
  if x < y
    then y := y − x
    else x := x − y
end
{x = GCD(A, B)}
```

Copyright © 2008 Steven D. Johnson

$$\frac{P \Rightarrow Q_v^t}{\{P\}\ v\ \texttt{:=}\ t\ \{Q\}} \qquad \text{ASSIGMENT RULE}$$

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \{R\}} \qquad \text{COMPOUND RULE}$$

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}} \qquad \text{CONDITIONAL RULE}$$

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{I \wedge \neg B\}} \qquad \text{REPETITION RULE}$$

Figure 10.1: Basic reasoning rules for the language of statements

$$\frac{P' \Rightarrow P \qquad \{P\}\ S\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$ Relaxation rule

$$\frac{\{P\}\ S\ \{Q^t_v\}}{\{P\}\ \texttt{begin}\ S\ \texttt{;}\ v\ \texttt{:=}\ t\ \texttt{end}\ \{Q\}}$$ Assignment-right

$$\frac{\{P \wedge (v = t)\}\ \mathrm{S}\ \{Q\}}{\{P\}\ \texttt{begin}\ v\ \texttt{:=}\ t\ \texttt{;}\ S\ \texttt{end}\ \{Q\}}$$ Assignment-left *—If v does not occur in P or t*

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$ Conditional Rule

$$\frac{\{P\}\ S_1\ \{I\} \qquad \{I\}\ \texttt{while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S_2\ \{Q\}}{\{P\}\ \texttt{begin}\ S_1\ \texttt{;}\ \texttt{while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S_2\ \texttt{end}\ \{Q\}}$$ initialization rule

$$\frac{P \Rightarrow I \quad \{I \wedge B\}\ S\ \{I\} \quad (I \wedge \neg B) \Rightarrow Q}{\{P\}\ \texttt{while}\ B\ \{\texttt{inv:}I\}\ \texttt{do}\ S\ \{Q\}}$$ While Rule

Figure 10.2: Derived reasoning rules for the language of statements

$$
\begin{array}{rl}
\textsc{Pre} & \text{is } \; x = A \;\wedge\; y = B \;\wedge A \geq B \\
\textsc{Post} & \text{is } \; x = A - B \\
\textsc{Inv} & \text{is } \; x - y = A - B \;\wedge\; x \geq y \\
\textsc{Test} & \text{is } \; y \neq 0 \\
\textsc{Body} & \text{is } \; \texttt{begin } x := x - 1 \texttt{ ; } y := y - 1 \texttt{ end}
\end{array}
$$

(1) $\{\textsc{Pre}\}$ `while` $\textsc{Test}$ `do` $\{\textsc{Inv}\}$ $\textsc{Body}$ $\{\textsc{Post}\}$
(2) $\textsc{Pre} \;\Rightarrow\; \textsc{Inv}$
(3) $\{\textsc{Inv} \;\wedge\; \textsc{Test}\}\; \textsc{Body} \;\{\textsc{Inv}\}$
(4) $\textsc{Inv} \;\wedge\; \neg\,\textsc{Test} \;\Rightarrow\; \textsc{Post}$
(5) $\{\textsc{Inv} \;\wedge\; \textsc{Test}\}\; x := x - 1 \;\{\textsc{Inv}\left[\begin{array}{c} y-1 \\ y \end{array}\right]\}$
(6) $\textsc{Inv} \;\wedge\; \textsc{Test} \;\Rightarrow\; \textsc{Inv}\left[\begin{array}{c} y-1 \\ y \end{array}\right]\left[\begin{array}{c} x-1 \\ x \end{array}\right]$

Verification Conditions

- (2) $x = A \;\wedge\; y = B \;\Rightarrow\; x - y = A - B$
- This is true by substitution of equals.

- (4) $(x - y = A - B) \;\wedge\; (x \geq y) \;\wedge\; \neg\,(y \neq 0) \;\Rightarrow\; x = A - B$
- The "not" of $\textsc{Test}$ means that $y = 0$. If $\textsc{Inv}$ also holds, we have

$$A - B = x - y = x - 0 = x$$

Thus, $\textsc{Post}$ follows from $\textsc{Inv}$ and $\textsc{Test}$.

- (6) $(x-y = A-B) \wedge (x \geq y) \wedge y \neq 0 \Rightarrow (x-1)-(y-1) = A-B \wedge (x-1) \geq (y-1)$.
- Assume that $x - y = A - B$ and that $y \neq 0$. Then

$$(x - 1) - (y - 1) = x - 1 - y + 1 = x - y = A - B$$

Also, if $x \geq y$ then by subtracting one from both sides, $(x - 1) \geq (y - 1)$.

Figure 10.3: Proof of Example 10.3

Copyright © 2008 Steven D. Johnson Draft Material (v08r0) January 26, 2009

Index

,, 9

ˆ, 11
acyclic, 88
algebra, 26
alphabet, 10, 11
antecedent, 21
antisymmetric, 87
assertion, 15
assignment rule, 186, 193
assignment statement, 14
associates, 144
asymmetric, 87

Backus-Naur form, 149
Backus-Naur notation, 15, 114
balanced parentheses, 162
base case, 48, 53
base set, 114
binomial coefficient, 41
bipartite graph, 78
bit, 25
BNF, 15, 149

calculus, 165
cardinality, 37
characteristic function, 84
Choice
  Principle of, 40
choose number, 41
composition
  of relations, **82**
compound rule, 189, 193
compound statement, 15
concatenation, 11
conditional rule, 187, 193
conditional statement, 14
consequent, 21
construction sequence, 121
constructive argument, 89
constructive proof, 93
constructor function, 114, 116
contains, 6
context free languages, 149
contingency, 24
contradiction, 24
cycle, 88
cyclic, 88

DAG, 98
decidable, 70
decision tree, 38
derivation, 26, 121
diagonalization, 66
difference
  of sets, 7
directed acyclic graph, 98
directed graph, 78
disjoint, 6
disjunctive normal form, 29
DNF, 29
domain, 5
dual, 27
duality, 27

ε, 12
element
  of a set, 3
ellipses, 4
empty word, 12
envelope, 68
environment, 146, 178
equality
  of sets, 6

equivalence class, 100
equivalence relation, 99
exponential function, 70

function, 78
  2-place, 84
  well defined, 83
functional form, 128

Gauss, Karl Friedrich, 49
graph
  bipartite, 78
  directed, 78
  of a relation, 77

homomorphism, 98

identifier, 14
image, 126
  of a function, 81
induction
  leicographic, 131
  mathematical, 47
  principle of, 47
induction hypothesis, 48
induction step, 48, 53
inductive set
  definition scheme, 116
inductive set definition
  definition by stages, 126
inductively set definition
  simultaneous, 127
infeasible, 72
infix notation, 84
injection, 84
integers, 4
intermediate assertion, 187
interpretation, 141
intersection
  of sets, 7
invariant, 60, 136
iteration, 136

labeling a relation, 90
language, 10, 12
  of statements, 14
length
  of a path, 88
lexicographic induction, 131
lexicographic ordering, 131
logically equivalent, 24
lower bound, 68

member
  (of a set), 3
  of a set, 3

$\mathbb{N}$, 4
$\emptyset$, 6
n-tuple, 8
natural numbers, 4, 117
negative logic, 27
numeral, 10, 13
numerical induction, 47

onto, 83
ordered pair, 7
orders of infinity, 72

$\mathcal{P}(A)$, 7
parse tree, 140
parsing, 140
parsing diagram, 140
partial function, 81
partial order, 102
partition, 101
path, 88
Pigeon-Hole Principle, 63
positive logic, 27
postcondition, 185
power set, 7
precedence, 145, 166
precondition, 185
predicate, 84
prefix notation, 84
preimage, 81
preorder, 102
product (of sets), 7
program variable, 14
proof
  by contradiction, 66
proportionality
  constant, 68

Copyright © 2008 Steven D. Johnson

factor, 68
proposition, 19
propositional formula, 19
propositional formulas, 150
propositional variable, 166

$\mathbb{Q}$, 6
quantification, 5
quotient set, 101

$\mathbb{R}$, 6
rank (of a function), 84
rational numbers, 6
real numbers, 6
reflexive, 87
relational form, 128
relaxation rule, 188
repetition rule, 193
repetition statement, 15
ROBDD, 107
Rule of Products, 40

scheme, 155, 168
formula, 155
self reference, 15, 113
set, 3
set builder notation, 5
size
of a set, 37
spanning tree, 94
statements
reasoning rules for, 193
statements, language of, 14
structure, 89
subsidiary derivation, 34
substitution, 153, 167
substitution function, 153
surjection, 83
symbol, 10
symmetric, 87
syntax, 5
concrete, 10

tautologically valid, 21
tautology, 24
threshold, 68
total order, 103
transitive, 87
transitive closure, 126
transliteration, 152
tree, 62
binary, 62
trivial argument, 22
truth table, 22

undecidable, 70
undefined, 81
union
of sets, 7
universe of discourse, 5
upper bound, 68

vacuous argument, 21
variable
propositional, 166
Venn diagram, 58

$\mathbb{W}$, 4
well defined function, 83
well formed formula, 165
WFF, 165
whole numbers, 4
"without loss of generality", 54
witness, 68
word, 11

$\mathbb{Z}$, 4
$\mathbb{Z}_n$, 6

Copyright © 2008 Steven D. Johnson