Theories of Computation

Theodore S. Norvell

Draft Typeset January 22, 2018

Contents

Outline **ix**

Preface **xi**

0 Correctness of Computing Systems **1**

0 Modelling (Computing) Systems **5**
0.0 Systems and Behavioural specifications 5
0.1 Angle bracket notation 10
0.2 Uses of Specifications 10
0.3 Refinement 12
0.4 Input, Output, Determinism, and Implementability 15

1 Imperative Programming Language **21**
1.0 States 21
1.1 A programming language 22
1.1.0 The **skip** command 22
1.1.1 Assignment commands 22
1.1.2 Parallel assignments 23
1.1.3 Sequential composition 24
1.1.4 The analogy between imperative specifications and matrices 27
1.1.5 Conjunction and Disjunction 29
1.1.6 Alternation 33
1.1.7 **abort** and **magic** 35
1.1.8 Iteration 35
1.2 Properties of the specification operators 37

2 Derivation of nonlooping programs **39**
2.0 Strengthening and monotonicity 39
2.1 Programming with **skip** and assignments 41
2.2 The substitution laws 43
2.3 Weakest prespecification and weakest postspecification 46
2.4 Alternation 47
2.4.0 Nondeterministic alternation 48

3 Derivation of Loops **49**
3.0 The recursive refinement approach 49
3.0.0 While law (incomplete version) 49
3.0.1 Summation of an array 50
3.0.2 Greatest Common Denominator 52
3.1 Loop Termination 54

4 Loop invariants **59**
4.0 Square root 59
4.1 The method of loop invariants 61
4.2 Examples of using loop invariants 62
4.2.0 Square Root by Binary Search 62
4.2.1 Searching for a pattern 65
4.3 Finding invariants 71

5 Data Transformation **73**
5.0 A slightly faster (and smaller) square root 73
5.0.0 An Abstract Binary Search algorithm 75

6 Placeholder **77**

1 Formal Languages and Models of Computation **79**

7 Strings, Languages, and Regular Expressions **83**
7.0 Strings 83
7.0.0 Languages 85
7.1 Regular language 86
7.2 Regular expressions 87
7.2.0 Syntax 87
7.2.1 Semantics 89

7.2.2 Examples and conventions . . . 90
7.3 Matching . . . 94
7.4 Equivalences of regular expressions . . . 95
7.5 Regular languages and looking forward . . . 95
7.6 Reversal . . . 96

8 Finite Recognizers 97
8.0 Nondeterministic Finite State Recognizers . . . 98
8.0.0 Syntax . . . 98
8.0.1 Semantics . . . 100
8.0.2 Systematic state renaming . . . 101
8.1 All regular languages are described by NDFRs . . . 101
8.1.0 Thompson's construction . . . 101
8.1.1 Example of Thompson's construction . . . 103
8.2 Recognition algorithms . . . 104
8.2.0 A one-finger algorithm . . . 104
8.2.1 Relationship to nondeterminism in programming . . . 106
8.2.2 A many-finger algorithm . . . 109
8.3 **Deterministic Finite State Recognizers (DFRs)** . . . 112
8.4 From NDFRs to DFRs . . . 113
8.4.0 Minimal DFRs . . . 116
8.4.1 Recognizing regular expressions with DFRs . . . 116
8.5 Equivalence of regular expressions and (N)DFRs . . . 117
8.5.0 From NDFRs to Regular Expressions . . . 118
8.5.1 Example . . . 121
8.5.2 Summary . . . 122
8.6 Are all language regular? . . . 122
8.7 Regular expressions in practice . . . 124
8.8 Chapter summary . . . 124

9 Reactive Systems 127
9.0 Reactive Systems . . . 127
9.0.0 Finite State Transducer . . . 127
9.0.1 Application to digital circuits . . . 128
9.1 System modelling and StateCharts . . . 128
9.1.0 Reactive systems . . . 128
9.1.1 StateCharts . . . 129
9.1.2 Transitions . . . 129

9.1.3 Conditions . . . 131
9.1.4 Time . . . 132
9.1.5 Hierarchy . . . 133
9.1.6 Concurrency. . . . 134
9.1.7 Communication . . . 135
9.2 System modelling and StateCharts . . . 136
9.2.0 A *Microwave oven* Example . . . 136
9.3 Using StateCharts to model software classes . . . 142
9.3.0 Relationship to other UML diagrams: . . . 143
9.3.1 An Example . . . 143
9.3.2 Another example: . . . 146
9.4 A Case Study — The RunEditor Dialog . . . 148
9.4.0 Statechart . . . 149
9.4.1 Implementation . . . 153
9.4.2 Implementation continued . . . 154

10 Context free grammars and context free parsing. 155
10.0 Grammars and Parsing . . . 155
10.0.0 Unrestricted Grammars . . . 155
10.0.1 Handier Notation . . . 158
10.0.2 Formalizing . . . 160
10.0.3 Context Free Grammars . . . 162
10.1 Examples Of Context Free Grammars . . . 163
10.1.0 Programming language examples . . . 163
10.1.1 Internet applications . . . 170
10.2 Recognition and Parsing . . . 171
10.3 Derivation Trees and Left-most Derivations . . . 172
10.3.0 Ambiguity and expression grammars . . . 175
10.4 Top-Down predictive parsing and recognition. . . . 176
10.4.0 Conceptual view . . . 176
10.4.1 Augmenting the grammar . . . 177
10.4.2 States, steps, and stops . . . 178
10.4.3 Example . . . 180
10.4.4 In a more algorithmic form . . . 180
10.4.5 LL(1) Grammars . . . 181
10.5 Recursive Descent Parsing . . . 183
10.5.0 Recursive Descent parsing of LL(1) grammars . . . 183
10.5.1 Getting results . . . 185

10.6 Dealing with non LL(1) grammars . . . 188
10.6.0 Converting to LL(1) . . . 188
10.7 Bottom-up, Shift-Reduce Parsing . . . 190
10.7.0 State . . . 190
10.7.1 Steps . . . 190
10.7.2 Stops . . . 190
10.7.3 Example using grammar Exp1 . . . 190
10.7.4 LR(1) grammars and Parser Generators . . . 191
10.7.5 Deterministic shift-reduce parsing . . . 192
10.8 Extended BNF (EBNF) . . . 193
10.8.0 Back to Extended BNF (or extended CFGs) . . . 193
10.8.1 EBNF is no more powerful than CFGs . . . 194
10.8.2 Recursive Descent Parsing with EBNF . . . 194
10.8.3 Syntax diagrams (or railroad diagrams) . . . 195
10.9 Regular languages . . . 196
10.9.0 First way . . . 197
10.9.1 Second way . . . 197
10.10Attribute grammars . . . 198
10.11Recognition by "dynamic programming" . . . 198

2 Efficiency 201

11 Computation time 203
11.0 The RAM model of computation . . . 203
11.1 Parameterized RAM . . . 207

12 A Dialogue concerning $P = NP$ and NP-Completeness 209
12.0 First day . . . 209
12.1 Second day . . . 215

3 Reference 225

A Mathematical Background 227
A.0 Sets . . . 227
A.0.0 Sets, elements, equality and subsets . . . 227
A.0.1 Operations on sets . . . 229

A.0.2 Set builder notation . . . 230
A.0.3 Minimum and maximum . . . 233
A.0.4 Sets of consecutive integers . . . 234
A.0.5 Pairs, other tuples, and Cartesian products . . . 234
A.0.6 The size of sets . . . 235
A.0.7 Set models of numbers and what isn't a set . . . 236
A.1 Relations and functions. . . . 239
A.1.0 Binary relations, partial functions, and total functions . 239
A.1.1 Domain and Range . . . 241
A.1.2 Application . . . 241
A.1.3 A digression on terminology . . . 241
A.1.4 Lambda expressions . . . 242
A.2 Sequences . . . 243
A.3 Graphs . . . 245
A.4 Categories . . . 246
A.5 Propositional logic . . . 247
A.5.0 Implication, follows from, and negation (NOT) . . . 247
A.5.1 Conjunction (AND) and disjunction (OR) . . . 248
A.5.2 Equivalence and exclusive-or. . . . 250
A.5.3 Duality . . . 250
A.5.4 Other notations . . . 251
A.6 Predicate Logic . . . 252
A.6.0 Substitution . . . 253
A.6.1 The Quantifiers ∀ and ∃ . . . 256
A.6.2 Laws . . . 264
A.6.3 'Universally true' and 'Stronger Than' . . . 265
A.7 Precedence and associativity . . . 267

Colophon **273**

Outline

- Part 0: Correctness of Computing Systems
  - General notions of specification and correctness.
  - Transformative systems (Hardware and software)
    - Documentation of software components.
    - Documentation of hardware components.
  - Data structures
  - Reactive systems (Hardware and software)
  - Data refinement
  - Objects
  - Hoare logic for sequential systems
  - Hoare logic for concurrent systems (POL)
- Part 2: Languages and Machines
  - Languages
  - Regular expressions
  - Finite recognizers
  - Context free grammars
  - Top-down and bottom-up recognizers
- Part 1: Efficiency of Computing Systems
  - Big Oh notation
  - Time complexity of algorithms

Typeset January 22, 2018

  - Efficient data structures
  - Efficient algorithms
  - Intractable algorithms
  - Complexity of problems
  - Tractable and intractable problems
  - The classes NP and NPC
  - Reductions.

- Part 3: Background
  - Appendix A: Review of Discrete Mathematics

Preface

Engineering is based on models and these models are usually mathematical. Engineers use mathematics in order to create mathematical models of things (systems) and situations (environments). By analyzing the properties of their mathematical models, engineers predict properties of real things in real situations. Even when a full analysis is not done, informal modelling and informal conclusions about the model guide an engineer's intuitions about design. For long established engineering disciplines such as Civil Engineering, Mechanical Engineering, and Electrical Engineering, I think that the preceding statements are not controversial. This book takes as a premise that the same applies to much of Computer Engineering, Software Engineering, and Computer Science.

This book is about some of the mathematical modelling techniques that apply to computing. It focusses particularly on modelling techniques that can be used to reason about the correctness and the efficiency of computer programs and digital hardware designs.

The first part of the book is about theories of correctness. We'll look at a theory that lets us specify the correct behaviour of a system or program — we will consider computer programs to be certain kind of system. This theory is generally known as 'Predicative Programming' and it provides an approach that is very easy to understand but also very general. The theory also gives us tools to decide whether a system is correct with respect to its specification and, importantly, it gives us methods to derive correct programs from their specifications. We also look at the related theories of Hoare-triples and Proof Outline Logic. The latter is particularly useful for analyzing the correctness of parallel programs, which is an important and tricky subject.

The second part of the book deals with sequences. Sets of sequences are called languages and we will look at ways of describing languages and ways of writing programs that recognize and analyse sequences (parsers).

Typeset January 22, 2018

The third part of the book deals with efficiency, in particular, asymptotic time complexity. The key idea is that by being a bit abstract we can compare the efficiency of algorithms without assuming much about their implementations. We do this by considering not the actual time that they take on particular inputs, but rather the way that the time they take increases as the size of the input increases. If one algorithm takes time proportional to the square of the size of its input, while another takes time proportional to the cube of the size of its input, we can see that the first algorithm will be quicker, for large enough inputs, regardless of implementation details and the values of the input. We need to know the details of implementation to know where the cross-over point will be, but not to see that there will be one and which algorithm will come out as the faster in the long run. We can use this approach not only to compare algorithms, but also to compare problems. We can draw the line between easy and hard problems according to whether the best algorithm for the problem is 'fast' or 'slow'. By defining 'fast' to mean that the time increases as a polynomial function of the input size, we are led to the theory of NP-completeness and one of the major unsolved problems of algorithmics, the so-called $P = NP$ problem. Most books that take the study of computational efficiency as far as NP-completeness do so using very abstract models of computation, usually Turing Machines. While this approach may be elegant, it is unnecessary and may be off-putting to the typical student who is used to using programming languages, not Turing Machines. My approach is to use ordinary programming notation and machine models that are closer to common experience to present computational efficiency and complexity.

Part 0

Correctness of Computing Systems

Typeset January 22, 2018

Letter conventions for this part of the book

I'll use variables as follows.

e, f, g, h, w	Specifications (or boolean valued functions)
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{I}$	Boolean expressions
$\mathcal{E}, \mathcal{F}$	Expressions or sequences of expressions
$\mathcal{V}, \mathcal{W}$	Names or sequences of names
Σ	Signatures
b, i, m, o	Behaviours
σ	States
n	Natural numbers

Note: variables used inside angle brackets, $\langle\rangle$, (and certain other places, such as between **if** and **then**,between **while** and **do**, and before and after the := sign) are state variables and do not follow these conventions. For example, the variable i is often used as the name of an integer component of the state. The $\mathcal{V}$ and $\mathcal{W}$, on the other hand, are not names themselves, but mathematical variables that range over names.

Chapter 0

Modelling (Computing) Systems

By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena.
John von Neumann

0.0 Systems and Behavioural specifications

System

A *system* is any object or collection of objects that imposes constraints on some collection of quantities. The collection of quantities is called the system boundary. For example an amplifier might have as part of its system boundary, the voltages on its supply voltage, input wire and output wire, and also its temperature and the amount of heat that it radiates. An airplane might have as part of its system boundary its shape, size, position, velocity, thrust, and the orientation of its various control surfaces (ailerons, rudder). System boundaries must be chosen carefully, as we will often think of a system as a "black box"; that is, we will think only about the system boundary and ignore any quantities within the system entirely. For example, while specifying a system, we will discuss only the quantities at the system boundary.

Any real engineered product will have a very complex system boundary and an important part of building a system model is to chose a subset of the system boundary to model. For example a system model of an airplane, modelling aerodynamic properties, might ignore the shape and size of the airplane and focus on how its position, velocity, thrust, and control surfaces

relate. A system model for an amplifier might ignore the supply voltage, the temperature and the heat given off.

In Engineering, we use names (variables) to represent actual physical quantities, things like voltages, currents etc. For example while

$$V = I \times R$$

is not necessarily true mathematically (take $V = 3$, $I = 4$, and $R = 5$), it is true physically, provided V represents the voltage across a resistor, I represents the current across the same resistor and R represents the resistance of the resistor, expressed in appropriate units. Often these quantities we give names to are ones on system boundaries.

Our description of systems and system boundaries, above, is rather vague and certainly subject to disagreement. To try to be more precise, I'll define mathematical concepts that reflect the aspects of systems and system boundaries that are of most interest for the purposes of this book. These concepts are *signatures, behaviours,* and *behavioural specifications.*

A signature is a list of the names for a system boundary together with information about the mathematical type for each name.

Signature

Definition 0 A *signature* Σ is a partial function from a set of names to some set of nonempty sets.

Variable

Definition 1 The names in the domain of a signature are variously called its *boundary variables,* its *state variables,* or simply its *variables.*[0]

A signature is the mathematical part of a system's boundary. Consider for example, an amplifier. Suppose that we choose to ignore such aspects as heat, current drain, temperature, and focus only on how the voltages on the input and output wires relate. These voltages change over time so each wire can be mathematically represented as a function of time. We can use the real numbers to represent time in seconds and also to represent voltage in

[0] We call these names *variables*, following computing tradition, although, mathematically, they are values. Even within computing, the word *variable* is used in at least three senses: first, for a name that is mapped to a value, second, for a name that is mapped to a memory location, and, third, for a location in a computer's memory. In this book, we use the word in the first sense.

volts, so each of the two wires will be a function from real numbers to real numbers. If we choose to call the input "x" and the output "x'"", then the signature[1] will be

$$\left\{ \text{“}x\text{”} \mapsto \left(\mathbb{R} \xrightarrow{\text{tot}} \mathbb{R} \right), \text{“}x'\text{”} \mapsto \left(\mathbb{R} \xrightarrow{\text{tot}} \mathbb{R} \right) \right\} \quad .$$

There is a lot of information not represented in this signature, information that is not mathematical in nature. For example, that we are using volts rather than say millivolts for voltage, that we are using seconds rather than minutes or milliseconds to measure time, what ground is used as a reference voltage and what instant is used for time zero. Even the facts that the variable "x" represents the input and the variable "x'" represents the output are not explicitly a part of the signature. So signatures leave out a lot of useful information that should be supplied by the engineer, by context, or by convention. Nevertheless, signatures contain the essential *mathematical* information about a system's boundary and so they will do for the purposes of this book.

If we observe a system, we can measure the value of each of the quantities on its boundary. A *behaviour* consists of a particular value for each variable of a signature.

Behaviour

Definition 2 A *behaviour* is a partial function from a set of names. A behaviour b *belongs to* a signature Σ iff

$$\text{dom}(b) = \text{dom}(\Sigma) \wedge \forall n \in \text{dom}(\Sigma) \cdot b(n) \in \Sigma(n) \quad .$$

I'll use the notation $b : \Sigma$ to mean that behaviour b belongs to signature Σ. Belongs to

A *behavioural specification* is a kind of a system model; it is a description of all possible behaviours of a system, as seen from the system boundary. Because in this book we won't look at any kinds of specifications that aren't behavioural, I'll just use the word *'specification'* as a synonymous with 'behavioural specification' from here on.

The purpose of a (behavioural) specification is to first describe a set of conceivable behaviours and second to divide the conceivable behaviours of a

[1] I'm only showing the graph of the signature. The source is a set of names; the target is some set of sets.

system into two categories. Behaviours that the system could actually engage in comprise one category and those that could not comprise the other. We'll use a boolean function to split the behaviours into categories.

Specification

Definition 3 Suppose that Σ is a signature and that f is a boolean valued function, the domain of which includes all behaviours that belong to Σ, then the pair (Σ, f) is a *specification*.

$$b \in \mathrm{dom}(f) \qquad \text{, for all } b : \Sigma$$

Notation 4 Generally, I'll write f_Σ instead of (Σ, f) for the pair, and, at times, I'll omit the subscript and write f, when the signature is clear from context or doesn't particularly matter. Furthermore I'll sometimes write $S(b)$ to mean $f(b)$ where $S = f_\Sigma$.

Example 5 Let

$$\Sigma = \left\{ \text{“}x\text{”} \mapsto \left(\mathbb{R} \xrightarrow{\text{tot}} \mathbb{R}\right), \text{“}x'\text{”} \mapsto \left(\mathbb{R} \xrightarrow{\text{tot}} \mathbb{R}\right) \right\}$$

and

$$f(b) = (\forall t \in \mathbb{R} \cdot b(\text{“}x'\text{”})(t) = b(\text{“}x\text{”})(t) \times 2) \qquad .$$

Then (Σ, f) (which we will also write as f_Σ or just f) is a specification for an amplifier that outputs twice its input signal, at each point in time.

Example 6 Consider a clocked digital circuit where we measure the voltages on wires at discrete points in time. These points in time are defined by the rising edges of the clock. We will take time 0 to be the time when the system is turned on, time 1 to be the first rising clock edge thereafter and so on. The voltage values don't matter except in as much as they represent true or false values. Let's consider a one-input, one-output device. An appropriate signature would be

$$\Sigma = \left\{ \text{“}d\text{”} \mapsto \left(\mathbb{N} \xrightarrow{\text{tot}} \mathbb{B}\right), \text{“}q'\text{”} \mapsto \left(\mathbb{N} \xrightarrow{\text{tot}} \mathbb{B}\right) \right\} \qquad .$$

If we define

$$f(b) = (\forall t \in \mathbb{N} \cdot b(\text{“}q'\text{”})(t+1) = b(\text{“}d\text{”})(t)) \qquad ,$$

then f_Σ is a specification for a simple memory device called a D-flip-flop. The output of a D-flip-flop is the same as its input in the previous clock period. The output at time 0 is not defined by f, but we can see from Σ that it will be either $\mathfrak{true}$ or $\mathfrak{false}$. This specification has two properties that the amplifier example does not. First, it exhibits memory, which is to say that the the output at each time may depend on earlier input. Second, it exhibits nondeterminism, which means that the output is not fully determined by the input.[2]

Example 7 Suppose that

$$\Sigma = \{\text{“}x\text{”} \mapsto A, \text{“}y\text{”} \mapsto A, \text{“}x'\text{”} \mapsto A, \text{“}y'\text{”} \mapsto A\} \quad .$$

We will interpret x and y as representing the initial values of program variables x and y and x' and y' as representing their final values. Let

$$f(b) = (b(\text{“}x'\text{”}) = b(\text{“}y\text{”}) \wedge b(\text{“}y'\text{”}) = b(\text{“}y\text{”})) \quad .$$

Then f_Σ models a command that assigns to x the value of y, in other words the assignment command[3] $x := y$.

Example 8 Let[4]

$$\Sigma = \{\text{“}in\text{”} \mapsto A^*, \text{“}out'\text{”} \mapsto A^*\} \quad ,$$

where A is some nonempty set, and f be a function

$$f(b) = (b\,(\text{“}out'\text{”}) = b\,(\text{“}in\text{”})\,\hat{}\,b\,(\text{“}in\text{”})) \quad .$$

The specification f_Σ will describe an “automaton” that reads an input sequence and writes it out twice. Chapter 9 will deal with automata as models of systems.

[2] At this point our definition of a system model does not distinguish between inputs and outputs and does not require any notion of time, so the properties of having memory and of being nondeterministic can't be defined formally. Later this will be rectified.

[3] In C , C++, or Java, this would be written

```
x = y;
```

In this book, following the tradition starting with the Algol language, I use := for the assignment operator and save the semi-colon for a more important purpose.

[4] For any set A, A^* is the set of finite sequences with items from A. The concatenation of sequences x and y is written $x; y$.

0.1 Angle bracket notation

As the examples above show, the notation for specifications is a bit awkward. Let's use the following convention for boolean functions of behaviours: We'll write the function as a boolean expression with the variables from the signature as free variables. We'll write the boolean expression in angle brackets. The angle brackets are there to remind us that we are dealing with a function, rather than a boolean expression. This convention is perhaps best further explained by example.

Example 9 With this convention the specifications from the above four examples are, respectively

$$\begin{aligned}
&\langle \forall t \in \mathbb{R} \cdot x'(t) = x(t) \times 2 \rangle_\Sigma && , \\
&\langle \forall t \in \mathbb{N} \cdot q'(t+1) = d(t) \rangle_\Sigma && , \\
&\langle x' = y \wedge y' = y \rangle_\Sigma && \text{, and} \\
&\langle out' = in \hat{} in \rangle_\Sigma && ,
\end{aligned}$$

where, in each case, Σ is as given in the corresponding example above.

Angle-bracket notation

We'll call this convention "angle-bracket notation".

0.2 Uses of Specifications

Specifications are useful for a number of purposes

- **Documentation:** Given a system that exists, or for which we have a sufficiently complete understanding, we may wish to describe all the ways it may behave. A specification can do that.
- **Requirements Specification:** A specification can be used to describe all the ways that it is acceptable for a system, which may not yet have been built, to behave.
- **Testing:** Once the system has been built, its actual behaviour can be compared with its intended specification. If the system behaves in a way that is not acceptable to the specification, then an error has been

detected. For example, if a behaviour $b : \Sigma$ is observed, and the system specification is g_Σ, then $\neg g(b)$ indicates an error.

- **Verification:** A system meets its specification just if every behaviour that the system could engage in is acceptable to the specification. Suppose specification g_Σ describes a system (i.e., it is the documentation of the system) and that f_Σ is a requirements specification. The formula

$$\forall b : \Sigma \cdot g(b) \Rightarrow f(b)$$

says that the the system meets the requirements f_Σ. So if we prove the above formula, we have verified that the system meets the requirements. If $\forall b : \Sigma \cdot g(b) \Rightarrow f(b)$, we say specification g_Σ refines specification f_Σ.

- **Design, derivation, or synthesis:** With verification, we start with two specifications and try to show refinement. When designing, we start with a specification f_Σ and try to find a system design whose specification g_Σ refines f_Σ. As we will see later, the process of design can often by done by a series of small steps, so that we start with a specification f_Σ and then find a series of specifications

$$f_\Sigma \quad f1_\Sigma \quad f2_\Sigma \quad \cdots \quad g_\Sigma$$

such that each refines the previous and such that the last corresponds to a system we can obviously build. This method is called *step-wise refinement*. In practice, each specification in the sequence differs only in part from the previous one and we can usually show that it refines the previous one looking only at the part that differs. We will look at this idea more closely when we look at top-down design of software systems in Chapter 2.

- **Analysis:** Given a design consisting of a set of components, joined together somehow, we might ask what the system as a whole does. If we know specifications of the components and how the components are joined together, then we can calculate a specification of the whole.

- **Equivalence testing:** Given two systems, we might ask if they are (behaviourally) equivalent. For example if we replace an expensive part with a less expensive part, we might wonder if that will change

the overall behaviour of the system. If the specifications are f_Σ and g_Σ we can ask whether

$$\forall b : \Sigma \cdot f(b) = g(b) \qquad .$$

If the answer is 'yes', then f_Σ and g_Σ are behaviourally equivalent and we can be sure that one can be replaced by the other. (If the answer is 'no', then further analysis may be needed.)

0.3 Refinement

Suppose that you work for a major operator of vending machines. You need to order 100 coffee dispensing machines for use at various locations. The machine will dispense about 200 ml of coffee for 1 dollar into a paper cup that holds about 220 ml. Now if the machine dispenses too much, it costs you extra money for the ingredients and more effort to replenish supplies; also the cup could overflow, which will make customers unhappy. On the other hand if the machine dispenses too little, customers will be unhappy. It might be nice to specify that the amount dispensed must be exactly 200 ml. However, it is unrealistic to expect a manufacturer to produce a machine that dispenses 200 ml exactly, every time, never a microliter more, never a microliter less. Thus you specify that the amount should be between 200 and 203 ml. Thinking of each cup of coffee as a behaviour, and considering only the amount dispensed as the system boundary, we have a signature of

$$\Sigma = \{\text{“}amount'\text{”} \mapsto \mathbb{R}\} \qquad .$$

We can write this requirement as a specification

$$f_\Sigma = \langle 200 \leq amount' \leq 203 \rangle_\Sigma \qquad .$$

Now suppose that the manufacturer delivers machines that actually dispenses between 201 and 202 ml. The actual machines' behaviour is described by

$$g_\Sigma = \langle 201 \leq amount' \leq 202 \rangle_\Sigma \qquad .$$

Obviously, although the specification of the machines delivered, g_Σ, is not exactly the same as your requirement, f_Σ, you can have no complaint against the manufacturer.

Now suppose that the machines delivered were actually described by any of the following specifications

$$\begin{aligned} g1_\Sigma &= \langle 201 \leq amount \leq 204 \rangle_\Sigma \quad , \\ g2_\Sigma &= \langle 199 \leq amount \leq 202 \rangle_\Sigma \quad , \text{ or} \\ g3_\Sigma &= \langle 199 \leq amount \leq 204 \rangle_\Sigma \quad . \end{aligned}$$

Clearly in each of these cases the delivered product does not meet out specification. There is some relationship $\sqsubseteq$ such that

$$\begin{aligned} f_\Sigma &\sqsubseteq g_\Sigma \quad \text{and} \\ f_\Sigma &\sqsubseteq f_\Sigma \end{aligned}$$

but

$$\begin{aligned} f_\Sigma &\not\sqsubseteq g1_\Sigma \quad , \\ f_\Sigma &\not\sqsubseteq g2_\Sigma \quad , \text{ and} \\ f_\Sigma &\not\sqsubseteq g3_\Sigma \quad . \end{aligned}$$

This relationship is essentially backwards implication, but it is backward implication for all possible behaviours. I.e. we have

$$\begin{aligned} &(\forall b : \Sigma \cdot f(b) \Leftarrow g(b)) \quad \text{and (obviously)} \\ &(\forall b : \Sigma \cdot f(b) \Leftarrow f(b)) \end{aligned}$$

but

$$\begin{aligned} &\neg\,(\forall b : \Sigma \cdot f(b) \Leftarrow g1(b)) \quad , \\ &\neg\,(\forall b : \Sigma \cdot f(b) \Leftarrow g2(b)) \quad , \text{ and} \\ &\neg\,(\forall b : \Sigma \cdot f(b) \Leftarrow g3(b)) \quad . \end{aligned}$$

For example as evidence that

$$\neg\,(\forall b : \Sigma \cdot f(b) \Leftarrow g1(b))$$

we can consider

$$b = \{\text{“}amount\text{”} \mapsto 203.5\}$$

We call this relationship between specifications *refinement* and say that f_Σ *is refined by* g_Σ.

Refinement

Definition 10 Specification f_Σ *is refined by* specification g_Σ iff

$$\forall b : \Sigma \cdot f(b) \Leftarrow g(b) \qquad .$$

$\sqsubseteq$

Notation 11 We write $f_\Sigma \sqsubseteq g_\Sigma$ to mean that f_Σ is refined by g_Σ. When Σ is clear from context, we may write $f \sqsubseteq g$.

Example 12 Suppose that Alice needs a subroutine that computes the square root of a positive number to about 4 decimal places. Calling the input x and the root x', we can use the following specification

$$\begin{aligned} \Sigma &= \{\text{“}x\text{”} \mapsto \mathbb{R}, \text{“}x'\text{”} \mapsto \mathbb{R}\} \\ f_\Sigma &= \left\langle x > 0 \Rightarrow x - 0.01 \leq x'^2 \leq x + 0.01 \right\rangle_\Sigma \qquad . \end{aligned}$$

This specification deserves a bit more explanation.Why the $x > 0 \Rightarrow$? Well Alice only cares what the subroutine will do in cases where x is positive, not when x is nonpositive. Thus it doesn't matter to Alice, what the behaviour is when x is nonpositive. To Alice, any behaviour where x is nonpositive should acceptable. This is exactly what you can say about a specification of the form $\langle x > 0 \Rightarrow P \rangle_\Sigma$; without even looking at the details of P, you can see that this boolean expression will evaluate to $\mathtt{true}$ (i.e., acceptable), whenever $x \leq 0$. Note that for each input value (i.e., value for x) there are an infinite number of values for x' that will combine with the value for x to make an acceptable behaviour. Now suppose that Alice assigns the job of writing the actual code for the subroutine to Bob. The subroutine he writes is unlikely to be capable of behaving in all the ways that Alice's specification deems acceptable. But that's ok. All we need is that each behaviour that Bob's subroutine can exhibit is acceptable to Alice's specification. Suppose that Bob creates a subroutine, whose behaviour is described by the following[5]

$$g_\Sigma = \left\langle (x < 0 \Rightarrow x' = 0.0) \wedge \left(x \geq 0 \Rightarrow x' = \left\lfloor \sqrt[+]{\lfloor x \times 10,000 \rfloor} \right\rfloor /100 \right) \right\rangle_\Sigma$$

Now the question of whether Bob's subroutine meets Alice's specification boils down to the question of whether or not

$$f_\Sigma \sqsubseteq g_\Sigma \qquad ,$$

[5]The notation $\lfloor x \rfloor$, for real number x means the largest integer not larger than x.

that is whether or not

$$\forall x, x' \in \mathbb{R} \cdot \left(\Leftarrow \begin{pmatrix} (x > 0 \Rightarrow x - 0.01 \leq x'^2 \leq x + 0.01) \\ (x < 0 \Rightarrow x' = 0.0) \\ \wedge \quad \left(x \geq 0 \Rightarrow x' = \left\lfloor \sqrt[+]{\lfloor x \times 10,000 \rfloor} \right\rfloor / 100 \right) \end{pmatrix} \right) \quad .$$

After a bit of simplification, the question comes down to

$$\forall x \in \mathbb{R} \cdot x > 0 \Rightarrow x - 0.01 \leq \left(\left\lfloor \sqrt[+]{\lfloor x \times 10,000 \rfloor} \right\rfloor / 100 \right)^2 \leq x + 0.01 \quad ,$$

which is in fact true.

Example 13 Recall Example 6:

$$\Sigma = \left\{ \text{“}d\text{”} \mapsto \left(\mathbb{N} \xrightarrow{\text{tot}} \mathbb{B} \right), \text{“}q'\text{”} \mapsto \left(\mathbb{N} \xrightarrow{\text{tot}} \mathbb{B} \right) \right\}$$
$$f_\Sigma = \langle \forall t \in \mathbb{N} \cdot q'(t+1) = d(t) \rangle_\Sigma \quad .$$

This specification allows two behaviours where the input, d, is $\mathfrak{false}$ at all times. In one behaviour $q'(0)$ is $\mathfrak{false}$ and in the other $q'(0)$ is $\mathfrak{true}$. Suppose that I put out a contract for devices that behave as f_Σ, and suppose that you promised to deliver 1000 such devices for 100 dollars. After agreeing to these terms, I pay the money and receive the parts and find that in fact, they are described by the specification

$$g_\Sigma = \langle q'(0) = \mathfrak{false} \wedge (\forall t \in \mathbb{N} \cdot q'(t+1) = d(t)) \rangle_\Sigma \quad .$$

Do I have a reason to sue you? All behaviours that the delivered flip-flops engage in are deemed acceptable by the specification, f_Σ. So we have $f_\Sigma \sqsubseteq g_\Sigma$ and I would not have any claim against you. On the other hand, if I had required a D-flip-flop satisfying g_Σ and you delivered flip-flops whose actual range of behaviour was described by f_Σ then I should demand my money back. We do not have $g_\Sigma \sqsubseteq f_\Sigma$.

0.4 Input, Output, Determinism, and Implementability

So far we've talked about behaviours without reference to which variables are controlled by the system and which are controlled from outside the system.

We call the variables controlled by the system its *output variables* and those that are controlled from outside its *input variables*.

It may seem surprising that we've gotten as far as we have without distinguishing inputs from outputs; in particular, the definition of refinement does not require inputs and outputs to be distinguished, nor do the notions of documentation, specification, analysis, design, or equivalence.

The convention used in this book, as you may have noticed, is to write input variables without any decoration, like this

$$x, y, d, q$$

and output variables with 'prime marks' like this

$$x', y', d', q' \qquad .$$

Each behaviour belonging to a signature can be divided into two aspects: an input aspect and an output aspect. For example if

$$b = \{\text{“}x\text{”} \mapsto 0, \text{“}y\text{”} \mapsto 1, \text{“}x'\text{”} \mapsto 2, \text{“}y'\text{”} \mapsto 3\}$$

is a behaviour, its input aspect is

$$\{\text{“}x\text{”} \mapsto 0, \text{“}y\text{”} \mapsto 1\}$$

while its output aspect is

$$\{\text{“}x\text{”} \mapsto 2, \text{“}y\text{”} \mapsto 3\}$$

I'm going to write $i \dagger o$ for the combination of two behaviours i and o. The combined behaviour $i \dagger o$ is a behaviour which has i as its input aspect and o as its output aspect. So for example

$$\{\text{“}x\text{”} \mapsto 0, \text{“}y\text{”} \mapsto 1\} \dagger \{\text{“}x\text{”} \mapsto 2, \text{“}y\text{”} \mapsto 3\}$$

gives the behaviour

$$\{\text{“}x\text{”} \mapsto 0, \text{“}y\text{”} \mapsto 1, \text{“}x'\text{”} \mapsto 2, \text{“}y'\text{”} \mapsto 3\}$$

Given a behaviour b, we write $\overleftarrow{b}$ for its input aspect and $\overrightarrow{b}$ for its output aspect so that in general

$$b = \overleftarrow{b} \dagger \overrightarrow{b}$$

and we'll use the same notations for signatures so that

$$\Sigma = \overleftarrow{\Sigma} \dagger \overrightarrow{\Sigma}$$

Now that we can distinguish between input and output, we can make some important definitions. Suppose we have a system exactly described by a specification f_Σ. For any particular input (stimulus) there is a set of possible outputs (responses).

Response set

Definition 14 For a given specification f_Σ and an input $i : \overleftarrow{\Sigma}$, the *response set* is given by

$$\text{resp}(f_\Sigma, i) \triangleq \left\{ o : \overrightarrow{\Sigma} \mid f(i \dagger o) \right\}$$

The sizes of the response sets tell us important things about the system's potential response to particular inputs.

Determined
Underdetermined
Overdetermined

Definition 15 Given a specification f_Σ we say

- f_Σ is *determined*, for input i, iff $|\text{resp}(f_\Sigma, i)| = 1$.
- f_Σ is *underdetermined*, for input i, iff $|\text{resp}(f_\Sigma, i)| > 1$.
- f_Σ is *overdetermined*,for input i, iff $|\text{resp}(f_\Sigma, i)| = 0$.

We can also describe two important properties of systems in terms of the sizes of the response sets over all inputs: determinist ism and implementability.

Determininstic
Nondeterminininstic

Definition 16 We say that f_Σ is *deterministic*, if it is determined for all inputs; that is if

$$\forall i : \overleftarrow{\Sigma} \cdot |\text{resp}(f_\Sigma, i)| = 1$$

When f is not deterministic it is *nondeterministic*.

Determinism can also be expressed by

$$\forall i : \overleftarrow{\Sigma} \cdot \exists! o : \overrightarrow{\Sigma} \cdot f(i \dagger o)$$

where the notation $\exists!$ means "there exists exactly one".

While many books on systems, treat only deterministic systems, the theory presented here will work equally well for deterministic and nondeterministic systems. Allowing nondeterministic systems as well as deterministic ones has a number of nice properties.

- We can specify a range of outputs. For example, if it is not important what the exact output is, we can specify a range of acceptable outputs. For example
$$\langle \sin x - 0.001 \leq y' \leq \sin x + 0.001 \rangle$$
- We can avoid specifying the output for input cases we do not care about. For example
$$\left\langle 0 \leq x < \frac{\pi}{2} \Rightarrow \sin x - 0.001 \leq y' \leq \sin x + 0.001 \right\rangle$$
- We can omit quantities from the system boundary. For example, consider a pseudo random number generator. If we know the seed, we can perfectly predict its output. However if we are not interested in knowing exactly what the output is, we can omit the seed from the system boundary and model the generator as $\langle y' \in \{0, ..100\} \rangle$.
- We can freely combine specifications with conjunction, disjunction, implication, and negation. If we dealt only with deterministic systems, there would be severe restrictions about using any of these operators.

We won't usually worry about the sizes of the response sets, as long as they are not of size zero. Empty response sets mean that the system must do something impossible: it must produce an output chosen from an empty set of choices. We need to be careful about specifications that have one or more empty response sets. We give these a name: *unimplementable*.

Implementable
Unimplementable

Definition 17 f_Σ is *implementable* if there is at least one acceptable output for each possible input; that is
$$\forall i : \overleftarrow{\Sigma} \cdot |\text{resp}(f_\Sigma, i)| > 0$$

In other words f_Σ is overdetermined for no inputs. If f_Σ is not implementable, we call it *unimplementable*; which is to say that it is overdetermined for at least one input

$$\exists i : \overleftarrow{\Sigma} \cdot \text{resp}(f_\Sigma, i) = \emptyset$$

.

Another way to express implementability that we will often use is

$$\forall i : \overleftarrow{\Sigma} \cdot \exists o : \overrightarrow{\Sigma} \cdot f(i \dagger o)$$

Similarly, a specification is unimplementable exactly if

$$\exists i : \overleftarrow{\Sigma} \cdot \forall o : \overrightarrow{\Sigma} \cdot \neg f(i \dagger o)$$

Implementability turns out to be tremendously important for the simple reason that all physical devices are implementable. Furthermore an unimplementable specification can not be refined by an implementable specification and hence can not be realized by a physical device. If someone hands you an unimplementable specification and asks you to design an implementation of it, they are asking for something impossible. This is useful to know; you should turn down the job!

Let's look at the two claims in the last paragraph more closely. First I claimed that physical systems will always be implementable. The reason is that a physical system will always behave in some way. If I apply a particular input, there will always be some kind of behaviour and thus some output.[6] The second claim is that an unimplementable specification can not be refined by an implementable one. I'll leave the proof of this as an exercise.

An unimplementable specification constrains its inputs in some way. It says that certain inputs are not possible. An example is

$$\langle x' < x \rangle_\Sigma$$

[6] Later we will face the question of what the output of a program is, if the program goes into an infinite loop. Some formalisms (such as Z [[ref]], which is otherwise very similar to the one presented in this book) take the point of view that an infinite loop has no output. This is an intuitively appealing point of view. However, it turns out that that approach requires a more complicated definition of refinement and more complicated definitions for certain kinds of system composition. We will take the point of view that all actual systems —even ones that take an infinite amount of time— have an output.

where

$$\Sigma = \{\text{“}x\text{”} \mapsto \mathbb{N}, \text{“}x'\text{”} \mapsto \mathbb{N}\} \qquad .$$

Recall that $\mathbb{N} = \{0, 1, 2, \cdots\}$ is the set of natural numbers. The specification says that the output must be less than the input. When the input is 2, the output can be 0 or 1; when the input is 1, the output must be 0; but when the input is 0, there is no possible output. The specification says that the input must not be 0. But it is not up to the system to determine its input values.

Chapter 1

Imperative Programming Language

In the next few chapters we will apply the theory of behavioural specifications, as presented in Chapter 0, to imperative programming of noninteractive systems. As we do so, new notations and concepts will be introduced. A few of these are particular to the application, but a great many of these notations and concepts can be applied to other areas such as interactive systems and hardware development.

The key observation is that a command in a programming language can be considered to be a specification. The specification relates two states: an initial state and a final state. This chapter looks at how we can understand programs from a simple programming language as specifications.

1.0 States

We can think of statements in imperative programs as being transformations on states. A state is simply a mapping from names to values. Assume Σ is a signature that maps only input variables,[0] then Σ is called a *state space* and any behaviour $b : \Sigma$ is a *state.* An imperative specification is then a specification with signature $\Sigma \dagger \Sigma$. Throughout this chapter the omitted subscripts on specifications will be assumed to be $\Sigma \dagger \Sigma$.

State space

State

[0] I.e., identifiers without prime marks.

1.1 A programming language

We will consider a simple programming language with commands of the following form

skip	Skip
$\mathcal{V} := \mathcal{E}$	Assignment
$f; g$	Sequential Composition
if $\mathcal{A}$ **then** f **else** g	Alternation
while $\mathcal{A}$ **do** f	Iteration
(f)	Grouping

The letters in this table are as follows: $\mathcal{V}$ a sequence of one or more program variable names; $\mathcal{E}$ is a sequence of expressions; f and g are commands; $\mathcal{A}$ is a boolean expression. For now, we won't worry about how program variables are declared, instead, we'll assume that, for each example, there is a fixed set of program variables with known types, as given by Σ. We'll also assume that each expression is of an appropriate type.

1.1.0 The skip command

skip

The **skip** command[1] means make no change to the state. Thus we have as a definition

$$\mathbf{skip}(i \dagger o) \triangleq (i = o)$$

For example, suppose that $\mathrm{dom}(\Sigma) = \{\text{“}x\text{”}, \text{“}y\text{”}, \text{“}z\text{”}\}$, then

$$\mathbf{skip} = \langle x' = x \wedge y' = y \wedge z' = z \rangle$$

1.1.1 Assignment commands

Assignment

Suppose that $\mathrm{dom}(\Sigma) = \{\text{“}x\text{”}, \text{“}y\text{”}, \text{“}z\text{”}\}$ and $\mathcal{E}$ is an expression involving the variables x, y, and z. An assignment $x := \mathcal{E}$ means 'change the value of x to the initial value of expression $\mathcal{E}$, while leaving the other variables unchanged'.[2] The final value of x (i.e. the value of "x" in the final state) is

[1] In C, C++, and Java, the skip command can be written as a semicolon not preceded by an expression "`;`"or as an empty pair of braces "`{}`".

[2] In C, C++, and Java assignment commands are written as "`x = `$\mathcal{E}$`;`".

the initial value of $\mathcal{E}$. The final values of y and z are the same as their initial values. Viewed as a specification, an assignment statement $x := \mathcal{E}$ is

$$\langle x' = \mathcal{E} \wedge y' = y \wedge z' = z \rangle$$

For example $x := y + z$ is

$$\langle x' = y + z \wedge y' = y \wedge z' = z \rangle$$

When $\mathrm{dom}(\Sigma)$ is something other than $\{\text{“}x\text{”}, \text{“}y\text{”}, \text{“}z\text{”}\}$, the interpretation of the assignment statement changes to match.

If you want a single all-purpose definition, here we go: Suppose Σ is a signature with $\mathcal{V}$ in its domain. Suppose g be a function that maps behaviours over Σ to values in $\Sigma(\mathcal{V})$ according to the expression $\mathcal{E}$;[3] then $\mathcal{V} := \mathcal{E}$ is a specification $f_{\Sigma\dagger\Sigma}$ where

$$f(i \dagger o) \triangleq (o(\mathcal{V}) = g(i) \wedge (\forall \mathcal{W} \in \mathrm{dom}(\Sigma) \cdot \mathcal{W} \neq \mathcal{V} \Rightarrow o(\mathcal{W}) = i(\mathcal{W})))$$

You can pronounce $\mathcal{V} := \mathcal{E}$ as “$\mathcal{V}$ is assigned $\mathcal{E}$” or as “$\mathcal{V}$ becomes $\mathcal{E}$”.[4]

1.1.2 Parallel assignments

To the left of the assignment operator we can have a finite sequence of variables while to the right we can have a finite sequence of expressions of equal length and corresponding types. For example[5]

$$x, y := 1, 2$$

Assuming that $\mathrm{dom}(\Sigma) = \{\text{“}x\text{”}, \text{“}y\text{”}, \text{“}z\text{”}\}$ we have

$$(x, y := 1, 2) = \langle x' = 1 \wedge y' = 2 \wedge z' = z \rangle$$

[3] For example if $\mathcal{E}$ is $x + y$ then $g(i) = i(\text{“}x\text{”}) + i(\text{“}y\text{”})$

[4] Some people make the mistake of saying “$\mathcal{V}$ equals $\mathcal{E}$”. I've heard people say “x equals $x+1$”, which is utter nonsense. This bad habit probably originated because early languages such as Fortran used the notation $\mathcal{V}$ `=` $\mathcal{E}$ for assignment. (The first version of Fortran, introduced in 1956, had no equality operator.) Unfortunately more recent languages such as C, C++, and Java have copied this notation. The notation $\mathcal{V} := \mathcal{E}$ was established with IAL (the International Algebraic Language, also known as Algol58) in 1958.

[5] In C, C++, and Java, there is no exact equivalent to the parallel assignment. In general we can write

```
{T x0=E; y=F ; x=x0 ; }
```

where `T` is the type of `x`.

Note that both expressions are evaluated in the same initial state. From an operational point-of-view, we can think of the changes to the two variables happening after both expressions have been evaluated. For example

$$(x, y := y, x) = \langle x' = y \wedge y' = x \wedge z' = z \rangle$$

In general

$$(\mathcal{V}, \mathcal{W} := \mathcal{E}, \mathcal{F}) = (\mathcal{W}, \mathcal{V} := \mathcal{F}, \mathcal{E})$$

1.1.3 Sequential composition

We can combine two or more specifications to make a new specification. We can also modify a specification in some way to make a new specification. Functions that take one or more specifications as inputs and produce a specification are called composition operators. Composition operators are to specifications as arithmetic operators, such as $+$, $-$, $\times$, and $\div$, are to numbers. The first kind of composition operator we will consider is sequential composition.

Imperative algorithms are recipes for doing sequences of actions. So far we've seen assignments and **skip** which are basic actions. Sequential composition allows us to sequence actions. If f and g are commands then $f; g$ is a command to do f and then to do g.[6] More generally, f and g can be any specifications with signatures $\Sigma \dagger \Sigma$; they don't need to be commands; if f and g are commands, then $f; g$ is a command, otherwise it is not.

What is the formal meaning of $f; g$? Let's assume, for the moment, that both f and g are implementable and that f is deterministic. Starting from a state i the command f with transform the system to the (unique) state m such that $f(i \dagger m)$. Then the command g will transform the system to some state o such that $g(m \dagger o)$. So we can say

$$(f; g)\,(i \dagger o) = g(m \dagger o), \text{ where } m \text{ is that state such that } f(i \dagger m)$$

Okay, but what if f is not deterministic? Suppose f is underdetermined for i, then m can be any state such that $f(i \dagger m)$. We have

$$(f; g)\,(i \dagger o) = (\exists m : \Sigma \cdot f(i \dagger m) \wedge g\,(m \dagger o))$$

[6] In C, C++, and Java, we could write this as $\{f\ g\}$. Note there is a significant difference between how C, C++, and Java use the semicolon and how it is used in this book: In C, C++, and Java, the semicolon is used to terminate some kinds of commands (expression commands, return commands, break, continue, etc.) and declarations. In this book the semicolon is used as a binary operator on specifications: sequential composition.

It turns out that this definition also makes sense in the cases where f and or g are not implementable.[7] Thus we will take this as the definition of sequential composition

Sequential composition

Definition 18 The *sequential composition* of $f_{\Sigma\dagger\Sigma}$ and $g_{\Sigma\dagger\Sigma}$ is a specification $h_{\Sigma\dagger\Sigma}$ where

$$h\,(i \dagger o) = (\exists m : \Sigma \cdot f(i \dagger m) \wedge g\,(m \dagger o)) \qquad \text{, for all } i, o : \Sigma \qquad (0)$$

Notation 19 We denote the sequential composition of $f_{\Sigma\dagger\Sigma}$ and $g_{\Sigma\dagger\Sigma}$ by $f_{\Sigma\dagger\Sigma}; g_{\Sigma\dagger\Sigma}$.

If the specifications to be composed are given using angle-bracket notation, we can give a simple definition of sequential composition. We'll assume $\Sigma = \{\text{“}x\text{”} \mapsto S, \text{“}y\text{”} \mapsto T, \text{“}z\text{”} \mapsto U\}$

$$\langle \mathcal{A} \rangle \,;\, \langle \mathcal{B} \rangle = \langle \exists \dot{x} \in S, \dot{y} \in T, \dot{z} \in U \cdot \mathcal{A}[x', y', z' : \dot{x}, \dot{y}, \dot{z}] \wedge \mathcal{B}[x, y, z : \dot{x}, \dot{y}, \dot{z}] \rangle$$

(Recall that $\mathcal{A}[x', y', z' : \dot{x}, \dot{y}, \dot{z}]$ means the expression $\mathcal{A}$ with variables x', y', and z' replaced by variables $\dot{x}$, $\dot{y}$, and $\dot{z}$. See Section A.6.0 in appendix A for details.) The expression $\mathcal{A}[x', y', z' : \dot{x}, \dot{y}, \dot{z}]$ relates the initial to the middle state, while the expression $\mathcal{B}[x, y, z : \dot{x}, \dot{y}, \dot{z}]$ relates the middle state to the final state.

Example 20 For this example, the operand specifications are both deterministic. Take $\Sigma = \{\text{“}x\text{”} \mapsto \mathbb{Z}, \text{“}y\text{”} \mapsto \mathbb{Z}, \text{“}z\text{”} \mapsto \mathbb{Z}\}$. Take

$$f = (x := x + 1) = \langle x' = x + 1 \wedge y' = y \wedge z' = z \rangle$$

and

$$g = (y := 2 \times x) = \langle x' = x \wedge y' = 2 \times x \wedge z' = z \rangle$$

[7] We won't have much reason to use sequential composition to combine unimplementable specifications, but let's consider what happens if we do. If $\text{resp}(f, i) = \emptyset$ then $\text{resp}((f; g), i) = \emptyset$. The case where $\text{resp}(g, m) = \emptyset$ is more interesting. In that case m is, in a sense, rejected by g. It is up to the command f to find a middle state m that will not be rejected by g. This corresponds to "backtracking", such as is found in the Icon, SNOBOL, and PROLOG programming languages.

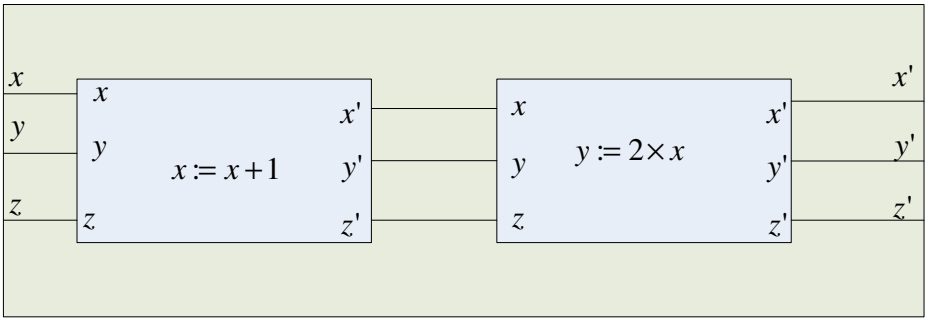


Figure 1.0: The sequential composition of $x := x + 1$ with $y := 2 \times x$.

We have

$$
\begin{array}{l}
\quad f; g \\
= \text{“Definitions of } f \text{ and } g\text{”} \\
\quad x := y + z; y := 2 \times x \\
= \text{“Definition of assignment”} \\
\quad \langle x' = y + z \wedge y' = y \wedge z' = z \rangle \, ; \langle x' = x \wedge y' = 2 \times x \wedge z' = z \rangle \\
= \text{“Definition of sequential composition”} \\
\quad \left\langle \begin{array}{ll} \exists \dot{x} \in \mathbb{Z}, \dot{y} \in \mathbb{Z}, \dot{z} \in \mathbb{Z} \cdot & (x' = y + z \wedge y' = y \wedge z' = z)\,[x', y', z' : \dot{x}, \dot{y}, \dot{z}] \\ \wedge & (x' = x \wedge y' = 2 \times x \wedge z' = z)\,[x, y, z : \dot{x}, \dot{y}, \dot{z}] \end{array} \right\rangle \\
= \text{“Making the substitutions”} \\
\quad \langle \exists \dot{x} \in \mathbb{Z}, \dot{y} \in \mathbb{Z}, \dot{z} \in \mathbb{Z} \cdot \dot{x} = y + z \wedge \dot{y} = y \wedge \dot{z} = z \wedge x' = \dot{x} \wedge y' = 2 \times \dot{x} \wedge z' = \dot{z} \rangle \\
= \text{“One point”} \\
\quad \langle x' = y + z \wedge y' = 2 \times (y + z) \wedge z' = z \rangle \\
= \text{“Definition of assignment”} \\
\quad x, y := y + z, 2 \times (y + z)
\end{array}
$$

Figure 1.0 illustrates this example.

1.1.4 The analogy between imperative specifications and matrices

We can think of imperative specifications as being very much like square matrices. Imperative specifications describe transformations on states, while matrices specify transformations on vectors. Just as matrix multiplication composes transformations represented by matrices, sequential composition composes transformations on states.

Example 21 To illustrate this analogy, let's consider Example 20 again. To f and g we can associate matrices

$$F = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$G = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So that $F\,(x, y, z)^{\mathrm{T}} = (y + z, y, z)^{\mathrm{T}}$ and $G\,(x, y, z)^{\mathrm{T}} = (x, 2 \times x, z)^{\mathrm{T}}$. The sequential composition of F and G is given by

$$GF = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

so that $GF(x, y, z)^{\mathrm{T}} = (y + z, 2 \times (y + z), z)^{\mathrm{T}}$. Note that the (most common) convention for matrices is to work from right to left (the first transformation applied is on the right) whereas for programming we work from left to right (the first transformation applied is on the left).

We can see that **skip** is the unit or identity element of sequential composition in the sense that

$$\mathbf{skip}; f = f = f; \mathbf{skip}$$

So **skip** is analogous to the identity matrix.

Just as matrix multiplication is associative[8], so is sequential composition

$$(f; g)\,; h = f; (g; h)$$

In general $f; g$ is not the same as $g; f$, although it may be for certain f and g. For example we have

$$(x := 0; y := 0) = (y := 0; x := 0)$$

but

$$(x := y; y := 0) \neq (y := 0; x := y)$$

This is to say that sequential composition is not commutative, in general. The same is true of matrix multiplication.

Just as some matrices (but not all) have inverses, some specifications (but not all) have inverses. For example, if the type of x is $\mathbb{Z}$ and

$$f = (x := x + 1)$$

then f's inverse is

$$\breve{f} = (x := x - 1)$$

as

$$f; \breve{f} = \mathbf{skip} = \breve{f}; f$$

Specification inversion is not used much in programming.

There are three ways that specifications are more general than matrices:

- Matrices are limited to state spaces that are homogeneous and numerical. I.e. each dimension has the same type and that type is a type of number.[9]
- Matrices are deterministic.
- Matrices specify linear transformations.

[8] That is

$$A(BC) = (AB)\,C$$

[9] If you know what a "ring" is, then you may recognize that I'm not being entirely truthful here. Generally matrices contain the elements of rings or even semirings, that is algebras that have both an "addition" operator and a "multiplication" operator. The elements of a ring may be numbers, but also might not be.

These differences make it hard to carry the analogy very far. For example, while we can add matrices, we can not (in general) add specifications, since the underlying data types may not have an addition operation. And there are operations on specifications that don't make sense on matrices. Next we look at two such operations.

1.1.5 Conjunction and Disjunction

Before tackling the if-then-else operator, it is useful to consider two simpler ways of combining specifications.

Definition 22 Consider two specifications f and g. The *disjunction* of f and g is the specification $f \vee g$ where

$$(f \vee g)(b) \triangleq f(b) \vee g(b) \qquad \text{, for all } b$$

The *conjunction* of f and g is the specification $f \wedge g$ where

$$(f \wedge g)(b) \triangleq f(b) \wedge g(b) \qquad \text{, for all } b$$

These definitions are even simpler using angle-bracket notation

$$(\langle \mathcal{A} \rangle \vee \langle \mathcal{B} \rangle) = \langle \mathcal{A} \vee \mathcal{B} \rangle$$
$$(\langle \mathcal{A} \rangle \wedge \langle \mathcal{B} \rangle) = \langle \mathcal{A} \wedge \mathcal{B} \rangle$$

Similarly we can define implication and negation on specifications:

$$(\langle \mathcal{A} \rangle \Rightarrow \langle \mathcal{B} \rangle) = \langle \mathcal{A} \Rightarrow \mathcal{B} \rangle$$
$$(\langle \mathcal{A} \rangle \Leftarrow \langle \mathcal{B} \rangle) = \langle \mathcal{A} \Leftarrow \mathcal{B} \rangle$$

and

$$\neg \langle \mathcal{A} \rangle = \langle \neg \mathcal{A} \rangle$$

It's important to remember that, in each case, the result of combining specifications with these operators is again a specification. It doesn't make sense, for example, to say that

$$\langle a \leq b' \rangle \Rightarrow \langle a < b' \rangle$$

is false. It is a specification that is true of some behaviours and false of others.

All the usual laws of propositional logic extend to specifications. So, for example, we have the following distributive law

$$f \wedge (g \vee h) = (f \wedge g) \vee (f \wedge h)$$

Disjunction is used to indicate that a system can behave in accordance with either of two specifications..

Example 23 For example, if we have

$$f = \left\langle x \geq 0 \Rightarrow x' = \sqrt[+]{x} \right\rangle$$
$$g = \left\langle x \geq 0 \Rightarrow x' = \sqrt[-]{x} \right\rangle$$

f specifies that the output should be the positive square root of the input, while g specifies that the output should be the negative root of the input. If we don't care which root is computed, we can use $f \vee g$ as the specification.

Conjunction is used to build a specification of a system that must behave simultaneously according to several specifications. Conjunction is often used to separately specify different parts of the output.

Example 24 For this example, the signature is

$$\{a \mapsto \mathbb{Z}, b \mapsto \mathbb{Z}, q \mapsto \mathbb{Z}, r \mapsto \mathbb{Z}\}$$

Suppose f specifies the value of q' to be the integer quotient of two integer numbers

$$f = \langle b \neq 0 \Rightarrow q' = \lfloor a \div b \rfloor \rangle$$

while

$$g = \langle b \neq 0 \Rightarrow r' = a \bmod b \rangle$$

specifies r' to be the remainder of the same two numbers. The conjunction $f \wedge g$ specifies a system that computes both the quotient and the remainder. After a bit of simplification

$$f \wedge g = \langle b \neq 0 \Rightarrow q' = \lfloor a \div b \rfloor \wedge r' = a \bmod b \rangle$$

Figure 1.1 illustrates this conjunction.

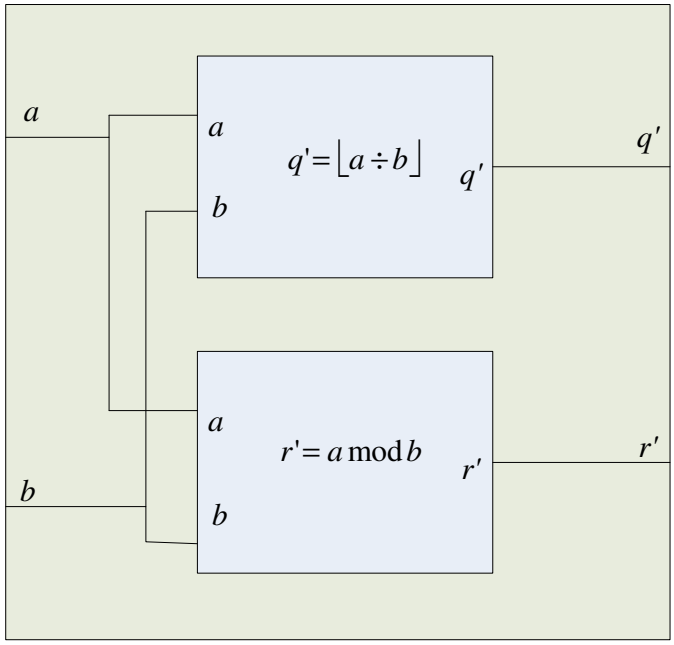


Figure 1.1: Conjunction of specifications with disjoint outputs.

Conjunction can also be used to combine specifications that constrain output under different circumstances.

Example 25 Consider

$$f = \langle a < 0 \Rightarrow a' = -1 \rangle$$
$$g = \langle a > 0 \Rightarrow a' = 1 \rangle$$

Note that f only constrains the output value for inputs with $a < 0$, while g only constrains the output value for inputs with $a > 0$. The conjunction $f \wedge g$ of the specifications specifies a value for the output in both cases, but not for the case of $a = 0$. If

$$h = \langle a = 0 \Rightarrow a' = 0 \rangle$$

then $f \wedge g \wedge h$ is the deterministic specification

$$\langle (a < 0 \Rightarrow a' = -1) \wedge (a > 0 \Rightarrow a' = 1) \wedge (a = 0 \Rightarrow a' = 0) \rangle$$

Figure 1.2 illustrates this conjunction

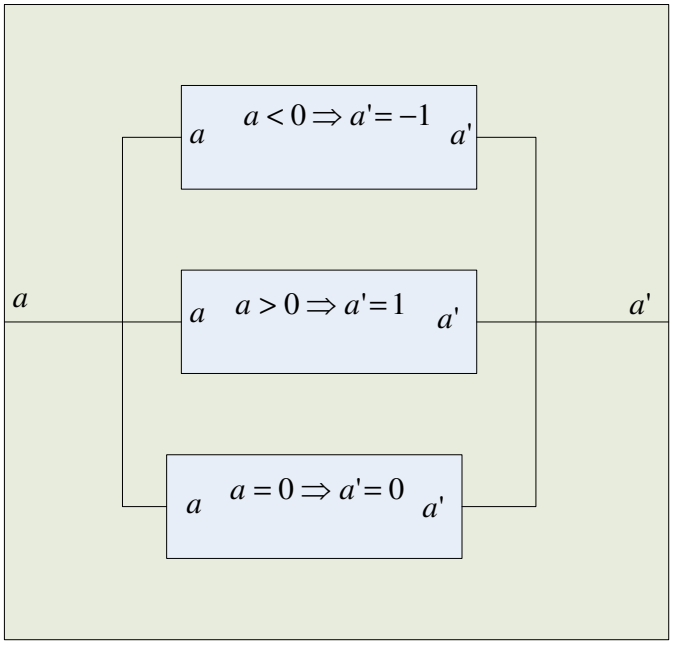


Figure 1.2: Conjunction of specifications that control the same output variable.

We have to be a bit careful with conjunction as it can lead to unimplementable specifications.

Example 26 Consider

$$f = \langle a \leq 0 \Rightarrow a' = -1 \rangle$$
$$g = \langle a \geq 0 \Rightarrow a' = 1 \rangle$$

This time f and g both constrain the output when $a = 0$ and actually contradict each other. The conjunction is

$$\begin{aligned} f \wedge g &= \langle (a \leq 0 \Rightarrow a' = -1) \wedge (a \geq 0 \Rightarrow a' = 1) \rangle \\ &= \langle (a < 0 \Rightarrow a' = -1) \wedge (a > 0 \Rightarrow a' = 1) \wedge (a = 0 \Rightarrow a' = 0 \wedge a' = 1) \rangle \\ &= \langle (a < 0 \Rightarrow a' = -1) \wedge (a > 0 \Rightarrow a' = 1) \wedge (a = 0 \Rightarrow a' = 0 \wedge a' = 1) \rangle \\ &= \langle (a < 0 \Rightarrow a' = -1) \wedge (a > 0 \Rightarrow a' = 1) \wedge (a = 0 \Rightarrow \mathfrak{false}) \rangle \\ &= \langle (a < 0 \Rightarrow a' = -1) \wedge (a > 0 \Rightarrow a' = 1) \wedge (a \neq 0) \rangle \end{aligned}$$

This specification specifies that its input will not be such that $a = 0$ and hence is an unimplementable specification.

1.1.6 Alternation

Two-way alternation Let $\mathcal{A}$ be a boolean expression with free variables that are a subset of the variables of Σ. The specification $\langle \mathcal{A} \rangle$ accepts or rejects a behaviour depending only on the input aspect of the behaviour. For each such expression, we have a binary operator on specifications **if** $\mathcal{A}$ **then** _ **else** _ defined by

$$\textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else } g \triangleq (\langle \mathcal{A} \rangle \wedge f) \vee (\neg \langle \mathcal{A} \rangle \wedge g)$$

This is called a two-way alternation or a two-way **if.**

Example 27 Supposing $dom(\Sigma) = \{\text{“}x\text{”}, \text{“}y\text{”}\}$, what is **if** $x \geq 0$ **then** $y := x$ **else** $y := -x$ in angle-bracket form

$$\begin{array}{l} \textbf{if } x \geq 0 \textbf{ then } y := x \textbf{ else } y := -x \\ = \\ (\langle x \geq 0 \rangle \wedge (y := x) \vee \langle x < 0 \rangle \wedge (y := -x)) \\ = \\ (\langle x \geq 0 \rangle \wedge \langle y' = x \wedge x' = x \rangle \vee \langle x < 0 \rangle \wedge \langle y' = -x \wedge x' = x \rangle) \\ = \\ \langle x \geq 0 \wedge y' = x \wedge x' = x \vee x < 0 \wedge y' = -x \wedge x' = x \rangle \end{array}$$

Exercise 28 Show that

$$\textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else } g = (\langle \mathcal{A} \rangle \Rightarrow f) \wedge (\neg \langle \mathcal{A} \rangle \Rightarrow g)$$

Exercise 29 Show that

$$\begin{array}{l} \textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else if } \mathcal{B} \textbf{ then } g \textbf{ else } h \\ = (\langle \mathcal{A} \rangle \wedge f) \vee (\neg \langle \mathcal{A} \rangle \wedge \langle \mathcal{B} \rangle \wedge g) \vee (\neg \langle \mathcal{A} \rangle \wedge \neg \langle \mathcal{B} \rangle \wedge h_\Sigma) \end{array}$$

Multiway alternation A generalization of the two-way alternation, that is sometimes useful, is this: Let $\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_{n-1}$ be n boolean expressions

with free variables that are a subset of the variables of Σ and let f_0, f_1, ..., f_{n-1}, g be specifications. Now

$$\begin{array}{llll} \textbf{if} & \mathcal{A}_0 & \textbf{then} & f_0 \\ \Box & \mathcal{A}_1 & \textbf{then} & f_1 \\ \vdots & \vdots & \vdots & \vdots \\ \Box & \mathcal{A}_{n-1} & \textbf{then} & f_{n-1} \\ \textbf{else} & g & & \end{array}$$

is a specification that is defined to be equal to

$$\begin{array}{ll} & (\langle \mathcal{A}_0 \rangle \wedge f_0) \\ \vee & (\langle \mathcal{A}_1 \rangle \wedge f_1) \\ \vdots & \vdots \\ \vee & (\langle \mathcal{A}_{n-1} \rangle \wedge f_{n-1}) \\ \vee & (\neg \langle \mathcal{A}_0 \rangle \wedge \neg \langle \mathcal{A}_1 \rangle \wedge \cdots \wedge \neg \langle \mathcal{A}_{n-1} \rangle \wedge g) \end{array}$$

Note that the order of the expression/specification pairs is not important: I.e. we can switch $\mathcal{A}_i$ and f_i with $\mathcal{A}_j$ and f_j for any i, j. Thus

$$\textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else if } \mathcal{B} \textbf{ then } g \textbf{ else } h$$

is not (in general) the same as

$$\textbf{if } \mathcal{A} \textbf{ then } f \,\Box\, \mathcal{B} \textbf{ then } g \textbf{ else } h$$

With the former, when $\mathcal{A}$ and $\mathcal{B}$ are both true of the input, the output is determined by f, whereas, with the latter, the output could be determined by either f or g.

It may be that g can not be reached. This happens when $\mathcal{A}_0 \vee \mathcal{A}_1 \vee \cdots \vee \mathcal{A}_{n-1}$ is universally true. In such a case, it doesn't matter what g is, and we can omit the "**else** g".[10]

[10] The nondeterministic if-command generalizes Dijkstra's if-fi command as well as the if-then-else.With Dijkstra's command, the g is always **abort** (see section 1.1.7), and is implicit. (Notationally, the "**else** g" part is replaced with "**fi**", in Dijkstra's notation.)

With the if-then-else construct in languages such as Algol, Pascal, C, C++, and Java, the "**else** g" part is optional, with the missing specification assumed to be **skip**.

The question then arises, of whether we should allow the else-clause of the if-construct to be omitted and, if so, whether the missing else-clause should be assumed to be "**else skip**" or "**else abort**". For this book, I'm going to come down squarely on the fence: I'll either include an else-clause or ensure that the missing clause is unreachable. Any other choice would create an inconsistency with one or the other established notation.

1.1.7 abort and magic

For a given signature Σ we have two specifications at the extremes. We call these **abort** and **magic**.

Definition 30 **abort** is equivalent to $\langle \mathfrak{true} \rangle$. **magic** is equivalent to $\langle \mathfrak{false} \rangle$.

abort accepts every possible behaviour; for any input, it allows any output. The system that is described by abort is completely unreliable in that it could produce any output whatsoever. It is easy to implement **abort**, as all specifications refine it

$$\textbf{abort} \sqsubseteq f, \text{ for all } f$$

magic, on the other hand, accepts no behaviours at all. Note that it is overdetermined for every input and hence (as there is always at least one possible input) is unimplementable. One curious property of **magic** is that it refines any other specification at all.

$$f \sqsubseteq \textbf{magic}, \text{ for all } f.$$

magic is not part of the programming language.

There is no harm in including **abort** in a programming language. The programmer can use **abort** at points in their code that they expect to be unreachable. The implementer of the language could implement **abort** to behave in any way they want; one particularly pragmatic way is to stop the program and alert the operating system, the user, and/or the developer. We can define a programming construct

$$\textbf{assert } \mathcal{A} \text{ to mean } \textbf{if } \mathcal{A} \textbf{ then skip else abort}$$

1.1.8 Iteration

An important property of iteration is that an iteration is equivalent to its unrolling: A loop **while** $\mathcal{A}$ **do** h that will iterate 4 times should be equivalent to $h; h; h; h$. In general we don't know ahead of time how many times to unroll the loop, but we can say a loop $w =$ **while** $\mathcal{A}$ **do** h should be such that

$$w = \textbf{if } \mathcal{A} \textbf{ then } (h; w) \textbf{ else skip}$$

Using this equation, we can unroll the loop as many times as we want. For example we can derive that

$$w = \textbf{if } \mathcal{A} \textbf{ then } (h; \textbf{if } \mathcal{A} \textbf{ then } (h; \textbf{if } \mathcal{A} \textbf{ then } (h; w) \textbf{ else skip}) \textbf{ else skip}) \textbf{ else skip}$$

Unfortunately, defining **while** $\mathcal{A}$ **do** h to be that x such that

$$x = \textbf{if } \mathcal{A} \textbf{ then } (h; x) \textbf{ else skip} \tag{1}$$

makes a poor definition of the while loop because x occurs on both sides of the equation. By analogy, defining ϕ to be the solution to the equation

$$x = 1/x + 1$$

is a poor definition of the real number ϕ: there are two solutions. Consider the case of $h =$ **skip** and $\langle \mathcal{A} \rangle = \langle \mathfrak{true} \rangle$. In this case equation (1) simplifies to $x = x$, which obviously has many solutions (every specification is a solution).

So how do we define the while loop? If $w0$ and $w1$ are two different solutions to equation (1) with $w0 \sqsubseteq w1$, then $w1$ is rejecting some behaviours without any reason that can be derived from $\mathcal{A}$ and h. We'll take the point of view **while** $\mathcal{A}$ **do** h accepts all behaviours accepted by any solution to equation (1). With this definition behaviours are only rejected if there is a good reason.

Definition 31 **while** $\mathcal{A}$ **do** h is defined to be that specification w such that

- $w = \textbf{if } \mathcal{A} \textbf{ then } (h; w) \textbf{ else skip}$ and
- for any specification x, such that $x = \textbf{if } \mathcal{A} \textbf{ then } (h; x) \textbf{ else skip}$, we have $w \sqsubseteq x$.

This definition says that the while loop is the least refined solution to equation (1).[11]

[11] You might wonder if this definition is sufficient. Is it the case that for any given $\mathcal{A}$ and h, is there one and only one specification w that fits the definition. It is fairly easy to see that this is the case. Let X be the set of all x such that

$$x = \textbf{if } \mathcal{A} \textbf{ then } (h; x) \textbf{ else skip}$$

The disjunction of any two members of X will also be in x. In fact the disjunction of any set (even an infinite set) of solutions is also a solution. The disjunction of all members of X will be a solution and is refined by any other solution. Thus the disjunction of all members of X is the unique least-refined solution of (1).

1.2 Properties of the specification operators

Definition 32

- A unary operator $\circledast$ on specifications is said to *preserve implementability* iff for all implementable specifications f , such that $\circledast f$ is well defined, $\circledast f$ is implementable.
- A binary operator $\circledast$ on specifications is said to *preserve implementability* iff for all implementable specifications f and g, such that $f \circledast g$ is well defined, $f \circledast g$ is implementable.
- In general, an n-ary operator $\circledast$ on specifications is said to *preserve implementability* iff for all implementable specifications f_0, f_1, ..., f_{n-1}, such that $\circledast(f_0, f_1, ..., f_{n-1})$ is well defined, $\circledast(f_0, f_1, ..., f_{n-1})$ is implementable.

In Example 26 we saw that conjunction does not preserve implementability.

Exercise 33 Show that disjunction preserves implementability.

Exercise 34 Define that a specification f_Σ is *independent* of a state variable v if $f(b_0) = f(b_1)$ whenever b_0 and b_1 agree on all variables other than v. Define that two specifications f and g are *output disjoint* if, for each output variable v', either f or g is independent of v'. Show that $f \wedge g$ is implementable if f and g are output disjoint and both f and g are implementable.

Exercise 35 Show that **if** $\mathcal{A}$ **then** _ **else** _ preserves implementability.

Exercise 36 Show that sequential composition preserves implementability.

Exercise 37 Show that **while** $\mathcal{A}$ **do** _ preserves implementability.

With numbers we say that $+$ is monotonic with respect to $\leq$ since $a \leq b$ implies that $a + c \leq b + c$ for all a, b, c.

For specifications, refinement is the key relation and so we define monotonicity with respect to refinement.

Definition 38

- A unary operator $\circledast$ is *monotonic* iff for all f_0, f_1 such that $f_0 \sqsubseteq f_1$, we have $\circledast f_0 \sqsubseteq \circledast f_1$.
- A binary operator $\circledast$ is *monotonic in its left operand* iff for all f_0, f_1, g such that $f_0 \sqsubseteq f_1$, we have $f_0 \circledast g \sqsubseteq f_1 \circledast g$.
- A binary operator $\circledast$ is *monotonic in its right operand* iff for all f_0, f_1, g such that $f_0 \sqsubseteq f_1$, we have $g \circledast f_0 \sqsubseteq g \circledast f_1$.
- In general, an n-ary operator is *monotonic in operand i* iff for all $f_0, f_1, g_0, g_1, ..., g_{i-1}, g_{i+1}, ..., g_{n-1}$ such that $f_0 \sqsubseteq f_1$, we have $\circledast(g_0, g_1, ..., g_{i-1}, f_0, g_{i+1}, ..., g_{n-1}) \sqsubseteq \circledast(g_0, g_1, ..., g_{i-1}, f_1, g_{i+1}, ..., g_{n-1})$

Exercise 39 Show that disjunction and conjunction are both monotonic in both operands

Exercise 40 Show that **if** $\mathcal{A}$ **then** _ **else** _ is monotonic in both operands.

Exercise 41 Show that sequential composition is monotonic in both operands.

Exercise 42 Show that **while** $\mathcal{A}$ **do** _ is monotonic.

Chapter 2

Derivation of nonlooping programs

In this chapter we look at techniques for deriving commands from specifications. That is we start with a specification f and derive a command g such that $f \sqsubseteq g$. We will develop commands step-by-step so that each step is fairly small and is easily verified. It should be said off the top that this is not a mechanical process. Far from it. Creativity is required.

The plan of the chapter is this. We'll look at some of the *laws of programming* for each of the programming constructs and how to apply these laws to deriving programs from specifications.

2.0 Strengthening and monotonicity

We start with some very general laws.

As explained in Section A.6.3, a boolean expression $\mathcal{B}$ is stronger than a boolean expression $\mathcal{A}$ exactly if $\mathcal{B} \Rightarrow \mathcal{A}$ is universally true (i.e., true regardless of the values of the variables).

When a specification is in angle-bracket form we can refine by replacing the expression with a stronger one. I.e. we have

Theorem 43 (Strengthening) $\langle \mathcal{A} \rangle \sqsubseteq \langle \mathcal{B} \rangle$ exactly if $\mathcal{B}$ is stronger than $\mathcal{A}$.

From this we have the following corollaries.

Corollary 44 (Strengthening)

$$\begin{aligned} \langle \mathcal{A} \vee \mathcal{B} \rangle &\sqsubseteq \langle \mathcal{A} \rangle \\ \langle \mathcal{B} \Rightarrow \mathcal{A} \rangle &\sqsubseteq \langle \mathcal{A} \rangle \\ \langle \mathcal{A} \rangle &\sqsubseteq \langle \mathcal{A} \wedge \mathcal{B} \rangle \end{aligned}$$

Corollary 45 (Monotonicity) If $\langle \mathcal{A} \rangle \sqsubseteq \langle \mathcal{B} \rangle$

$$\begin{aligned} \langle \mathcal{A} \wedge \mathcal{C} \rangle &\sqsubseteq \langle \mathcal{B} \wedge \mathcal{C} \rangle \\ \langle \mathcal{A} \vee \mathcal{C} \rangle &\sqsubseteq \langle \mathcal{B} \vee \mathcal{C} \rangle \\ \langle \mathcal{C} \Rightarrow \mathcal{A} \rangle &\sqsubseteq \langle \mathcal{C} \Rightarrow \mathcal{B} \rangle \end{aligned}$$

Corollary 46 (Antimonotonicity) If $\langle \mathcal{B} \rangle \sqsubseteq \langle \mathcal{A} \rangle$ then

$$\begin{aligned} \langle \neg \mathcal{A} \rangle &\sqsubseteq \langle \neg \mathcal{B} \rangle \\ \langle \mathcal{A} \Rightarrow \mathcal{C} \rangle &\sqsubseteq \langle \mathcal{B} \Rightarrow \mathcal{C} \rangle \end{aligned}$$

More generally we have the following monotonicity laws regardless of the form of the specification.

Theorem 47 If f, g, and h are specifications such that $f \sqsubseteq g$, we have

- $f \wedge h \sqsubseteq g \wedge h$
- $f \vee h \sqsubseteq g \vee h$
- $h \Rightarrow f \sqsubseteq h \Rightarrow g$
- $f; h \sqsubseteq g; h$
- $h; f \sqsubseteq h; g$
- **if** $\mathcal{A}$ **then** f **else** $h \sqsubseteq$ **if** $\mathcal{A}$ **then** g **else** h
- **if** $\mathcal{A}$ **then** h **else** $f \sqsubseteq$ **if** $\mathcal{A}$ **then** h **else** g
- **while** $\mathcal{A}$ **do** $f \sqsubseteq$ **while** $\mathcal{A}$ **do** g

2.1 Programming with skip and assignments

Earlier we saw that **skip** can refine specifications of the form $\langle x' = x \wedge y' = y \rangle$. But **skip** will also refine other specifications, such as $\langle k > 0 \Rightarrow k' \geq 0 \rangle$; if k is initially greater than 0, then, after not changing, k will still be greater than 0 and thus greater or equal to 0. If $\mathcal{A}$ is an expression, we'll write $\widetilde{\mathcal{A}}$ for the expression we get by erasing all the prime marks in $\mathcal{A}$. So if $\mathcal{A}$ is the expression $k > 0 \Rightarrow k' \geq 0$ then $\widetilde{\mathcal{A}}$ would be the expression $k > 0 \Rightarrow k \geq 0$. This particular expression, $k > 0 \Rightarrow k \geq 0$, has the property that it is true for all values of its variables. A boolean expression with this property is called *universally true*. In general we have the following law

universally true

Theorem 48 (The erasure law for skip) $\langle \mathcal{A} \rangle \sqsubseteq \mathbf{skip}$ exactly if $\widetilde{\mathcal{A}}$ is universally true.

I won't prove this law in general, but consider the case where the variables are x and y.

$$
\begin{array}{ll}
 & \langle \mathcal{A} \rangle \sqsubseteq \mathbf{skip} \\
= & \text{“Definitions of refinement and skip”} \\
 & \forall x, y, x', y' \cdot x' = x \wedge y' = y \Rightarrow \mathcal{A} \\
= & \text{“One-point”} \\
 & \forall x, y \cdot \mathcal{A}[x', y' : x, y] \\
= & \text{“Definition of erasure”} \\
 & \forall x, y \cdot \widetilde{\mathcal{A}} \\
= & \text{“Definition of universally true”} \\
 & \widetilde{\mathcal{A}} \text{ is universally true}
\end{array}
$$

The substitution notation is covered in Section A.6.0.

If we take this same line of reasoning and apply it to the assignment statement $x := \mathcal{E}$, where $\mathcal{E}$ does not contain any primed variables, we get

$$
\begin{aligned}
& \langle \mathcal{A} \rangle \sqsubseteq x := \mathcal{E} \\
= & \text{“Definitions of refinement and assignment} \\
& \forall x, y, x', y' \cdot x' = \mathcal{E} \wedge y' = y \Rightarrow \mathcal{A} \\
= & \text{“One-point”} \\
& \forall x, y \cdot \mathcal{A}[x', y' : \mathcal{E}, y] \\
= & \text{“}y' \text{ does not occur free in } \mathcal{E}\text{”} \\
& \forall x, y \cdot \mathcal{A}[x' : \mathcal{E}][y' : y] \\
= & \text{“}x' \text{ does not occur free in } \mathcal{A}[x' : \mathcal{E}] \\
& \forall x, y \cdot \mathcal{A}[x' : \mathcal{E}][x', y' : x, y] \\
= & \text{“Definition of erasure”} \\
& \forall x, y \cdot \widetilde{\mathcal{A}[x' : \mathcal{E}]} \\
= & \text{“Definition of universally true”} \\
& \widetilde{\mathcal{A}[x' : \mathcal{E}]} \text{ is universally true}
\end{aligned}
$$

which leads us to

Theorem 49 (The erasure law for assignment) $\langle \mathcal{A} \rangle \sqsubseteq \mathcal{V} := \mathcal{E}$ exactly if $\widetilde{\mathcal{A}[\mathcal{V}' : \mathcal{E}]}$ is universally true.

For example, if k is of type $\mathbb{Z}$, then $\langle k' > k \rangle \sqsubseteq k := k+42$, as $(k' > k)\,[k' : k + 42]$ is $k + 42 > k$ which is universally true.

This law also extends to parallel assignment.

Theorem 50 (The erasure law for parallel assignment) $\langle \mathcal{A} \rangle \sqsubseteq \mathcal{V}_0, \mathcal{V}_1, ..., \mathcal{V}_n := \mathcal{E}_0, \mathcal{E}_1, ..., \mathcal{E}_n$ exactly if $\widetilde{\mathcal{A}[\mathcal{V}'_0, \mathcal{V}'_1, ..., \mathcal{V}'_n : \mathcal{E}_0, \mathcal{E}_1, ..., \mathcal{E}_n]}$ is universally true.

For example, let's work out whether the following refinement holds.

$$\langle x' = y \wedge y' = x \wedge z' \geq z \rangle \sqsubseteq x, y := y, x$$

After making the multiple variable substitution

$$(x' = y \wedge y' = x \wedge z' \geq z)\,[x', y' : y, x]$$

and erasing the remaining prime, we have $y = y \wedge x = x \wedge z \geq z$, which is universally true, so the refinement holds.

2.2 The substitution laws

For introducing sequential composition, there is a very useful law

Theorem 51 (The forward substitution law) $\langle \mathcal{A}[\mathcal{V} : \mathcal{E}] \rangle = (\mathcal{V} := \mathcal{E}; \langle \mathcal{A} \rangle)$

I won't prove this in general, but again, consider the case of a two variable state space

$$
\begin{array}{ll}
 & (x := \mathcal{E}; \langle \mathcal{A} \rangle) \\
= & \text{“Definitions of assignment and sequential composition”} \\
 & \langle \exists \dot{x}, \dot{y} \cdot \dot{x} = \mathcal{E} \wedge \dot{y} = y \wedge \mathcal{A}[x, y : \dot{x}, \dot{y}] \rangle \\
= & \text{“One-point”} \\
 & \langle \mathcal{A}[x, y : \dot{x}, \dot{y}][\dot{x}, \dot{y} : \mathcal{E}, y] \rangle \\
= & \text{“No dotted variables in } \mathcal{A}\text{”} \\
 & \langle \mathcal{A}[x, y : \mathcal{E}, y] \rangle \\
= & \text{“No substitution for } y\text{”} \\
 & \langle \mathcal{A}[x : \mathcal{E}] \rangle
\end{array}
$$

The substitution law is very useful for introducing sequential composition into programs.

Example 52 Consider the following specification to implement

$$\langle x' = y \wedge y' = x \rangle$$

We will assume that multiple assignments are not allowed. We'll also assume that there is a variable t of appropriate type. Can we derive a sequential

composition of single assignments that does the job?

$$
\begin{array}{ll}
& \underline{\langle x' = y \wedge y' = x \rangle} \\
= & \text{Forward substitution law} \\
& t := x \,;\, \underline{\langle x' = y \wedge y' = t \rangle} \\
= & \text{Forward substitution law} \\
& t := x \,;\, x := y \,;\, \underline{\langle x' = x \wedge y' = t \rangle} \\
\sqsubseteq & \text{Erasure law for assignment} \\
& t := x \,;\, x := y \,;\, y := t
\end{array}
$$

Note how the last step also uses a monotonicity law. We generally won't call attention to uses of monotonicity laws. They are used implicit.

There is also a law for introducing an assignment statement at the end of a sequential composition.

Notation 53 Suppose $\mathcal{E}$ is an expression with all free variables unprimed. We'll write $\mathcal{E}'$ for the same expression, except with primes added to all free variables, and $\dot{\mathcal{E}}$ for the same expression except with a dot added to each variable.

Theorem 54 (The backward substitution law) $\langle \mathcal{A} \rangle \sqsubseteq (\langle \mathcal{A}[\mathcal{V}' : \mathcal{E}'] \rangle \,;\, \mathcal{V} := \mathcal{E})$

Example 55 Consider swapping again. Again, we'll assume there is a variable t that we can use.

$$
\begin{array}{ll}
& \underline{\langle x' = y \wedge y' = x \rangle} \\
\sqsubseteq & \text{“Backward substitution”} \\
& \underline{\langle x' = y \wedge t' = x \rangle}; y := t \\
\sqsubseteq & \text{“Backward substitution”} \\
& \underline{\langle y' = y \wedge t' = x \rangle}; x := y; y := t \\
\sqsubseteq & \text{“Erasure law”} \\
& t := x; x := y; y := t
\end{array}
$$

I won't prove this law in detail, but looking at the case of two variables should be convincing

$$
\begin{array}{ll}
& \langle \mathcal{A}[x' : \mathcal{E}'] \rangle \,;\, x := \mathcal{E} \\
= & \text{“Definition of assignment”} \\
& \langle \mathcal{A}[x' : \mathcal{E}'] \rangle \,;\, \langle x' = \mathcal{E} \wedge y' = y \rangle \\
= & \text{“Definition of sequential composition”} \\
& \left\langle \exists \dot{x}, \dot{y} \cdot \mathcal{A}[x' : \mathcal{E}'][x', y' : \dot{x}, \dot{y}] \wedge x' = \dot{\mathcal{E}} \wedge y' = \dot{y} \right\rangle \\
= & \text{“Since the primes in } \mathcal{E}' \text{ will be replaced with dots”} \\
& \left\langle \exists \dot{x}, \dot{y} \cdot \mathcal{A}[x', y' : \dot{\mathcal{E}}, \dot{y}] \wedge x' = \dot{\mathcal{E}} \wedge y' = \dot{y} \right\rangle \\
= & \text{“One point”} \\
& \left\langle \exists \dot{x}, \dot{y} \cdot \mathcal{A}[x', y' : \dot{\mathcal{E}}, y'] \wedge x' = \dot{\mathcal{E}} \right\rangle \\
= & \text{“It's safe to assume that } x' = \dot{\mathcal{E}}\text{”} \\
& \left\langle \exists \dot{x}, \dot{y} \cdot \mathcal{A}[x', y' : x', y'] \wedge x' = \dot{\mathcal{E}} \right\rangle \\
= & \text{“Null substitution”} \\
& \left\langle \exists \dot{x}, \dot{y} \cdot \mathcal{A} \wedge x' = \dot{\mathcal{E}} \right\rangle \\
= & \text{“Distributivity; no dots in } \mathcal{A}\text{”} \\
& \langle \mathcal{A} \rangle \wedge \left\langle \exists \dot{x}, \dot{y} \cdot x' = \dot{\mathcal{E}} \right\rangle
\end{array}
$$

Now since $\left(\langle \mathcal{A} \rangle \wedge \left\langle \exists \dot{x}, \dot{y} \cdot x' = \dot{\mathcal{E}} \right\rangle \right) = \left(\langle \mathcal{A}[x' : \mathcal{E}'] \rangle \,;\, x := \mathcal{E} \right)$, we have, by strengthening, the backward substitution law:

$$\langle \mathcal{A} \rangle \sqsubseteq \langle \mathcal{A}[x' : \mathcal{E}'] \rangle \,;\, x := \mathcal{E}$$

Note that $\langle \mathcal{A}[x' : \mathcal{E}'] \rangle \,;\, x := \mathcal{E}$ may be a stronger requirement than $\langle A \rangle$; the 'extra' bit $\left\langle \exists \dot{x}, \dot{y} \cdot x' = \dot{\mathcal{E}} \right\rangle$ says that the value of x' is in the range of the expression. Consider refining

$$\langle x' = 4 \vee x' = 5 \rangle$$

using backward substitution and $x := 2 \times x$, where x is an *integer* variable. We have

$$
\begin{array}{ll}
 & \langle x' = 4 \lor x' = 5 \rangle \\
\sqsubseteq & \text{“Backward substitution”} \\
 & \langle 2 \times x' = 4 \lor 2 \times x' = 5 \rangle \, ; x := 2 \times x \\
= & \text{“}x \text{ is an integer variable”} \\
 & \langle x' = 2 \rangle \, ; x := 2 \times x \\
= & \text{“Definitions of sequential composition and assignment”} \\
 & \langle x' = 4 \rangle
\end{array}
$$

You can see that there is a reduction in nondeterminism, the output of $x = 5$ is accepted by $\langle x' = 4 \lor x' = 5 \rangle$, but is not compatible with the chosen assignment.

This reduction in nondeterminism may mean that the law takes us from an implementable $\langle \mathcal{A} \rangle$ to an unimplementable $\langle \mathcal{A}[x' : \mathcal{E}'] \rangle$.

Consider the specification $\langle x' = 5 \rangle$. If we apply backward substitution with $x := 2 \times x$, we get

$$
\begin{array}{ll}
 & \langle x' = 5 \rangle \\
\sqsubseteq & \text{“backward substitution”} \\
 & \langle 2 \times x' = 5 \rangle \, ; x := 2 \times x
\end{array}
$$

At this point, the remaining problem, $\langle 2 \times x' = 5 \rangle$, is unimplementable.

Thus, for practical application, we should be careful how we use backward substitution, so as not to venture into the territory of unimplementable specifications.

The forward substitution law does not suffer from this problem: If $\langle \mathcal{A}[\mathcal{V} : \mathcal{E}] \rangle$ is implementable, then so is $\langle \mathcal{A} \rangle$.

2.3 Weakest prespecification and weakest postspecification

This section is optional reading.

[[And not yet written!]]

2.4 Alternation

Once we have checked a condition, it can become a precondition. This idea is captured in the alternation law

Theorem 56 (alternation law)

$$f = \textbf{if } \mathcal{A} \textbf{ then } (\langle \mathcal{A} \rangle \Rightarrow f) \textbf{ else } (\neg \langle \mathcal{A} \rangle \Rightarrow f)$$

The alternation problem allows us to replace a problem f with two new problems. The trick, of course is to pick an expression $\mathcal{A}$ so that both the new problems are easier to solve than the original.

Example 57 (Finding the minimum) We know that

$$\min(a, b) = a, \text{ if } a \leq b \tag{0}$$
$$\min(a, b) = b, \text{ if } b \leq a \tag{1}$$

Suppose we wish to implement

$$f = \langle a' = \min(a, b) \rangle$$

$$\begin{array}{ll} & f \\ = & \text{Alternation law} \\ & \textbf{if } a \leq b \textbf{ then } (\langle a \leq b \rangle \Rightarrow f) \textbf{ else } (\langle a > b \rangle \Rightarrow f) \end{array}$$

We can implement the first case as follows

$$\begin{array}{ll} & \langle a \leq b \rangle \Rightarrow f \\ = & \text{Defn of } f \\ & \langle a \leq b \Rightarrow a' = \min(a, b) \rangle \\ = & \text{By (0)} \\ & \langle a \leq b \Rightarrow a' = a \rangle \\ \sqsubseteq & \text{Erasure law} \\ & \textbf{skip} \end{array}$$

The second case is implemented by

$$
\begin{array}{ll}
 & \langle a > b \rangle \Rightarrow f \\
= & \text{Defn of } f \\
 & \langle \underline{a > b} \Rightarrow a' = \min(a, b) \rangle \\
\sqsubseteq & \text{Strengthening} \\
 & \left\langle a \geq b \Rightarrow a' = \underline{\min(a, b)} \right\rangle \\
= & (1) \\
 & \langle a \geq b \Rightarrow a' = b \rangle \\
\sqsubseteq & \text{Erasure law} \\
 & a := b
\end{array}
$$

Now we have

$$
\begin{array}{ll}
 & f \\
= & \text{Alternation law} \\
 & \textbf{if } a \leq b \textbf{ then } \underline{(\langle a \leq b \rangle \Rightarrow f)} \textbf{ else } \underline{(\langle a > b \rangle \Rightarrow f)} \\
\sqsubseteq & \text{Above results} \\
 & \textbf{if } a \leq b \textbf{ then skip else } a := b
\end{array}
$$

2.4.0 Nondeterministic alternation

Theorem 58 (alternation law)

$$
\begin{array}{lllll}
f \sqsubseteq & \textbf{if} & \mathcal{A}_0 & \textbf{then} & \langle \mathcal{A}_0 \rangle \Rightarrow f \\
 & \square & \mathcal{A}_1 & \textbf{then} & \langle \mathcal{A}_1 \rangle \Rightarrow f \\
 & \vdots & \vdots & \vdots & \vdots \\
 & \square & \mathcal{A}_{n-1} & \textbf{then} & \langle \mathcal{A}_{n-1} \rangle \Rightarrow f \\
 & \textbf{else} & & & \neg \langle \mathcal{A}_0 \vee \mathcal{A}_1 \vee \cdots \vee \mathcal{A}_{n-1} \rangle \Rightarrow f
\end{array}
$$

Chapter 3

Derivation of Loops

3.0 The recursive refinement approach

3.0.0 While law (incomplete version)

For specifications h and $\langle \mathcal{A} \rangle$, let $t_{\mathcal{A},h}$ be a function

$$t_{\mathcal{A},h}(g) = \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$$

Let

$$w = \textbf{while } \mathcal{A} \textbf{ do } h$$

Now w is a fixed point of $t_{\mathcal{A},h}$, that is

$$w = \textbf{if } \mathcal{A} \textbf{ then } (h; w) \textbf{ else skip}$$

A specification g, such that $g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$, will, in many cases, be implemented by w

While law (incomplete version): For any g, h, and $\mathcal{A}$, such that ... if

$$g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$$

then

$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

Later we will complete this law (fill in the "...") with extra conditions that ensure it is valid for particular g, h, and $\mathcal{A}$.

This gives us an approach to refining a specification g with a while loop. We derive an if command such that its else branch is refined by **skip**. In

refining the then-part, if we come to a point where all that remains to be done is described by g, we simply stop. Now we have

$$g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$$

Then (subject to some conditions we will see later), we have $g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$.

3.0.1 Summation of an array

For this problem, we calculate the sum of all the elements in an array of integers a of size n (a fixed natural number)

$$f = \left\langle s' = \left(\sum_{k \in \{0,..n\}} a(k) \right) \right\rangle$$

It turns out that f is not suitable for directly applying the while law.

The strategy is to *find a generalization* of the problem g that can serve as the specification of a loop:

$$\begin{array}{l} f \\ \sqsubseteq \text{Substitution law} \\ i, s := 0, 0 \,;\, g \end{array}$$

where

$$g = \left\langle s' = s + \left(\sum_{k \in \{i,..n\}} a(k) \right) \right\rangle$$

Now the problem remaining is to derive a program for g.

In the case where $i \geq n$ the problem is easy to solve

$$\begin{array}{ll} & g \\ \sqsubseteq & \\ & \textbf{if } i < n \\ & \textbf{then } \langle i < n \rangle \Rightarrow g \\ & \textbf{else } \langle i \geq n \rangle \Rightarrow g \end{array}$$

Tackling the second problem first we have

$$\left\langle i \geq n \Rightarrow s' = s + \left(\sum_{k \in \underline{\{i,..n\}}} a(k) \right) \right\rangle$$

$=$"If $i \geq n$, then $\{i, ..n\} = \emptyset$"

$$\left\langle i \geq n \Rightarrow s' = s + \underline{\left(\sum_{k \in \emptyset} a(k) \right)} \right\rangle$$

$=$The sum over an empty set is 0

$\langle i \geq n \Rightarrow s' = s + 0 \rangle$

$\sqsubseteq$"Erasure"

skip

In the second case

$$\left\langle i < n \Rightarrow s' = s + \left(\sum_{k \in \underline{\{i,..n\}}} a(k) \right) \right\rangle$$

$=$"If $i < n$ we can rewrite $\{i, ..n\}$ as $\{i\} \cup \{i+1, ..n\}$"

$$\left\langle i < n \Rightarrow s' = s + \underline{\left(\sum_{k \in \{i\} \cup \{i+1,..n\}} a(k) \right)} \right\rangle$$

$=$"Split the summation"

$$\left\langle \underline{i < n} \Rightarrow s' = s + a(i) + \left(\sum_{k \in \{i+1,..n\}} a(k) \right) \right\rangle$$

$\sqsubseteq$"Strengthen"

$$\left\langle s' = \underline{s + a(i)} + \left(\sum_{k \in \underline{\{i+1},..n\}} a(k) \right) \right\rangle$$

$=$"Substitution law"

$i, s := i + 1, s + a(i)\ ; g$

Putting these results together (with monotonicity) we get that

$$
\begin{array}{ll}
 & g \\
\sqsubseteq & \\
 & \quad \textbf{if } i < n \\
 & \quad \textbf{then } \langle i < n \rangle \Rightarrow g \\
 & \quad \textbf{else } \langle i \geq n \rangle \Rightarrow g \\
\sqsubseteq & \text{Above calculations} \\
 & \quad \textbf{if } i < n \\
 & \quad \textbf{then } (i, s := i + 1, s + a(i); g) \\
 & \quad \textbf{else skip}
\end{array}
$$

Now we apply the while law

$$g \sqsubseteq \textbf{while } i < n \textbf{ do } i, s := i + 1, s + a(i)$$

and thus (by monotonicity)

$$
\begin{array}{ll}
f \sqsubseteq & i, s := 0, 0; \\
 & \quad \textbf{while } i < n \textbf{ do} \\
 & \quad\quad i, s := i + 1, s + a(i)
\end{array}
$$

3.0.2 Greatest Common Denominator

For any natural numbers a and b, we write $a \mid b$ iff a divides b, i.e. iff there exists a $q \in \mathbb{N}$ such that $aq = b$. Note that

$$a \mid 0, \text{ for all natural numbers } a \tag{0}$$

The *greatest common denominator* of two natural numbers a and b is a natural number $\gcd(a, b)$ with the following properties.

$$
\begin{array}{l}
\gcd(a, b) \mid a, \text{ for all natural numbers } a, b \\
\gcd(a, b) \mid b, \text{ for all natural numbers } a, b \\
\text{if } c \mid a \text{ and } c \mid b \text{ then } c \mid \gcd(a, b), \\
\quad \text{for all natural numbers } a, b, c
\end{array}
$$

From these properties we can derive the following facts (proof left as exercise)

$$\gcd(a, 0) = a, \tag{1}$$
$$\text{for all natural numbers } a, \text{ where } a \neq 0$$
$$\gcd(a, b) = \gcd(b, a \bmod b), \tag{2}$$
$$\text{for all natural numbers } a, b \text{ where } b \neq 0$$

From (0) it is clear that $\gcd(0, 0)$ is not defined, since there are an infinite number of common divisors. Therefore, it makes sense to use as a precondition that at least one argument is not zero. Let

$$g = \langle a \neq 0 \vee b \neq 0 \Rightarrow a' = \gcd(a, b)\rangle$$

This is ready for implementation as a loop.

$$\begin{array}{l} \quad g \\ =\text{Alternation} \\ \quad \textbf{if } b \neq 0 \\ \quad \textbf{then } \langle b \neq 0\rangle \Rightarrow g \\ \quad \textbf{else } \langle b = 0\rangle \Rightarrow g \end{array}$$

In the else part we have (after shunting)

$$\begin{array}{l} \quad \langle b = 0 \wedge (a \neq 0 \vee b \neq 0) \Rightarrow a' = \gcd(a, b)\rangle \\ = \text{ One point and identity law for } \vee \\ \quad \langle b = 0 \wedge a \neq 0 \Rightarrow a' = \gcd(a, 0)\rangle \\ =\text{Fact (1)} \\ \quad \langle b = 0 \wedge a \neq 0 \Rightarrow a' = a\rangle \\ \sqsubseteq\text{Erasure} \\ \quad \textbf{skip} \end{array}$$

In the then part we have (after shunting)

$$
\begin{array}{ll}
& \langle b \neq 0 \wedge (a \neq 0 \vee b \neq 0) \Rightarrow a' = \gcd(a,b) \rangle \\
= & \text{Domination law for } \vee \\
& \langle b \neq 0 \wedge \mathit{true} \Rightarrow a' = \gcd(a,b) \rangle \\
= & \text{Identity law for } \wedge \\
& \langle b \neq 0 \Rightarrow a' = \gcd(a,b) \rangle \\
= & \text{Fact (2)} \\
& \langle b \neq 0 \Rightarrow a' = \gcd(b, a \bmod b) \rangle \\
\sqsubseteq & \text{Strengthening} \\
& \langle b \neq 0 \vee a \bmod b \neq 0 \Rightarrow a' = \gcd(b, a \bmod b) \rangle \\
= & \text{Forward substitution} \\
& a, b := b, a \bmod b \,;\, g
\end{array}
$$

Now putting the two cases together we get

$$
\begin{array}{ll}
& g \\
\sqsubseteq & \\
& \textbf{if } b \neq 0 \\
& \textbf{then } \; a, b := b, a \bmod b \,;\, g \\
& \textbf{else skip}
\end{array}
$$

So by the while loop law we have

$$g \sqsubseteq \textbf{while } b \neq 0 \textbf{ do } a, b := b, a \bmod b$$

3.1 Loop Termination

In order to use while loops in programming, we need to be able to prove statements such as

$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

So we need to know under what conditions this refinement holds. Earlier we saw an incomplete law

While law (incomplete version): For any g, h, and $\mathcal{A}$, such that ... if

$$g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$$

then

$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

But we still need to fill in the "..."

It turns out that what is missing is that we must require the loop to terminate. Consider two loops where x is a natural number program variable

$$\textbf{while } x > 0 \textbf{ do } x := x - 1$$

and

$$\textbf{while } x \neq n \textbf{ do } x := x + 1$$

The first definitely terminates, while the second may not. How can we show that a loop terminates?

In the case of **while** $x > 0$ **do** $x := x - 1$, we can find an expression, for example x, such that the number of remaining iterations, never exceeds that expression. Such an expression is called a *bound* on the number of remaining iterations, that is an expression $\mathcal{E}$ such that the number of iterations remaining is no more than $\mathcal{E}$.

In the case of **while** $x \neq n$ **do** $x := x + 1$, we can not find any such expression. The expression $n - x$ will act as bound in some states (those where $n \geq x$), but not in all states.

If we can find a bound for a loop, then it always terminates.

3.1..0 A better iteration law

While law (better version): For any g, h, and $\mathcal{A}$, such that **while** $\mathcal{A}$ **do** h always terminates, if

$$g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; g) \textbf{ else skip}$$

then

$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

3.1..1 An even better iteration law

In practice, we do not require that loops terminate regardless of the initial state. We only require that they terminate when started in states that matter. For example, we would expect that, for an integer i,

$$\langle i \leq n \Rightarrow i' = n \rangle \sqsubseteq \textbf{while } i \neq n \textbf{ do } i := i + 1$$

even though, the loop does not terminate when started with an initial value of i that is larger than n.

To show this sort of refinement we need an even better law. We will use a program variable τ of type $\mathbb{Z}$ to count the number of repetitions of loops. You can think of this variable as representing time. You might expect that every operation will take some amount of time and so we should imagine the time variable increasing with each operation. That would be a more realistic approach to time, but also a more complicated one. For many purposes, it is sufficient to consider how time

Let us strengthen the example specification to include information about the maximum number of repetitions allowed

$$g = \langle (i \leq n \Rightarrow i' = n \wedge \tau' \leq \tau + (n - i)) \rangle$$

Now we show

$$g \sqsubseteq \textbf{if } i \neq n \textbf{ then } (i := i + 1; \tau := \tau + 1; g) \textbf{ else skip}$$

While law (final version): For any g, h, and $\mathcal{A}$, where g is of the form $\langle \mathcal{B} \Rightarrow \mathcal{C} \wedge \tau' \leq \tau + \mathcal{E} \rangle$ and $\mathcal{E}$ is a natural number expression, if

$$g \sqsubseteq \textbf{if } \mathcal{A} \textbf{ then } (h; \tau := \tau + 1; g) \textbf{ else skip}$$

then

$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

($\mathcal{E}$ is called the *bound* for the loop.)

Practically, what this means is that when $\mathcal{A}$ and $\mathcal{E} > 0$ are true initially, h needs to decrease the value of $\mathcal{E}$ by at least 1.

3.1..2 Summation revisited

In the summation problem above we had to refine

$$\left\langle i \leq n \Rightarrow s' = s + \left(\sum_{k \in \{i,..n\}} a(k) \right) \right\rangle$$

with a loop. In that case the bound can be $n - i$, so we should refine

$$\left\langle i \leq n \Rightarrow s' = s + \left(\sum_{k \in \{i,..n\}} a(k) \right) \wedge \tau' \leq \tau + n - i \right\rangle$$

by a while loop.

3.1..3 GCD revisited

In the GCD problem we had to refine

$$\langle a \neq 0 \vee b \neq 0 \Rightarrow a' = \gcd(a, b)\rangle$$

in this case we can use b as the bound. We can show that

$$\langle a \neq 0 \vee b \neq 0 \Rightarrow a' = \gcd(a, b) \wedge \tau' \leq \tau + b\rangle$$

is implemented by

$$\textbf{while } b \neq 0 \textbf{ do } a, b := b, a \bmod b$$

since $a \bmod b < b$.

Chapter 4

Loop invariants

4.0 Square root

Suppose x is a natural number. We wish to find the integer part of its square root.

$$\begin{aligned} f &= \left\langle y'^2 \leq x < (y'+1)^2 \right\rangle \\ &= \left\langle y'^2 \leq x \wedge x < (y'+1)^2 \right\rangle \end{aligned}$$

Our aim is to implement f using a loop. If we have already searched the first y natural numbers we know $y^2 \leq x$. Let's call this expression $\mathcal{I}$.

$$\mathcal{I} \text{ is } y^2 \leq x$$

We can generalize f by adding $\mathcal{I}$ as an antecedent.

$$g = (\langle \mathcal{I} \rangle \Rightarrow f)$$

We have

$$f \sqsubseteq y := 0; g$$

$\mathcal{I}$ is called the loop's *invariant*.

Note that $\mathcal{I}$ is one of the conjuncts of f,except with the primes erased. Let's take the other conjunct, negate it, and erase primes; we get $(y+1)^2 \leq x$; we'll use this expression as a guard. Applying the alternation law we have

$$g \sqsubseteq \textbf{if } (y+1)^2 \leq x \textbf{ then } g0 \textbf{ else } g1$$

where the else-clause is

$$g1 = \left\langle y^2 \leq x < (y+1)^2 \Rightarrow y'^2 \leq x < (y'+1)^2 \right\rangle$$

We have a specification of the form $\langle \mathcal{A} \Rightarrow \mathcal{B} \rangle$ such that $\mathcal{A}$ is the same as $\widetilde{\mathcal{B}}$. By the erasure law, such a specification can always be implemented by **skip.**

For the "then" clause we have:

$$g0 = \left\langle \mathcal{I} \wedge (y+1)^2 \leq x \Rightarrow y'^2 \leq x < (y'+1)^2 \right\rangle$$

We need to find a specification h so that

$$g0 \sqsubseteq h; g$$

The following will do

$$h = \left\langle \mathcal{I} \wedge (y+1)^2 \leq x \Rightarrow \mathcal{I}' \wedge x' = x \right\rangle$$

where $\mathcal{I}'$ is the same as $\mathcal{I}$, but with primes:

$$\mathcal{I}' \text{ is } y'^2 \leq x'$$

The key point about h is that its job is to *preserve the invariant.*

$$h = \langle \mathcal{I} \wedge \cdots \Rightarrow \mathcal{I}' \wedge \cdots \rangle$$

That is, if h starts in a state where $\mathcal{I}$ holds, it must end in a state where $\mathcal{I}$ holds.

Exercise 59 Prove that $g0 \sqsubseteq h; g$ using the definition of sequential composition.

Now all that remains is to implement h.

$$\begin{array}{l} h \\ \quad = \text{By definition} \\ \quad \left\langle \mathcal{I} \wedge (y+1)^2 \leq x \Rightarrow \mathcal{I}' \wedge x' = x \right\rangle \\ \quad \sqsubseteq \text{Strengthening and assignment laws} \\ y := y+1 \end{array}$$

To show that the loop terminates, we can use $x - y^2$ as a bound.

4.1 The method of loop invariants

Let's try to separate the *method* used to solve the last problem, from the details of the problem.

The general form of the solution is

$$f \sqsubseteq m; g$$
$$g \sqsubseteq \textbf{while } \mathcal{A} \textbf{ do } h$$

The key to the method is the notion of an invariant $\mathcal{I}$, which is a condition on the initial state.

Suppose $f = \langle \mathcal{B} \rangle$ and $\mathcal{B}$ depends on a list of input variables $\bar{x}$. We choose an invariant $\mathcal{I}$ and generalize $f = \langle \mathcal{B} \rangle$ to $g = \langle \mathcal{I} \Rightarrow \mathcal{B} \rangle$.

The role of the initialization statement m, is to *establish the invariant*, that is to make $\mathcal{I}$ true at the start of the loop. This suggests

$$\langle \mathcal{I}' \rangle$$

However, that is not enough, m must not change any of the variables $\bar{x}$ whose initial value is used in $\mathcal{B}$.

$$m = \langle \mathcal{I}' \wedge \bar{x}' = \bar{x} \rangle$$

Exercise 60 Exercise: Show that $f \sqsubseteq m; g$

Next, we turn our attention to implementing $g = \langle \mathcal{I} \Rightarrow \mathcal{B} \rangle$ with **while** $\mathcal{A}$ **do** h. In the case where $\mathcal{A}$ is false, the while loop will do nothing, so the final state will be the same as the initial state. We can assume that $\mathcal{I}$ is true in the initial state. So we need that $\neg\mathcal{A} \wedge \mathcal{I}$ implies that $\mathcal{B}$ relates the initial state to itself: I.e. we need that

$$\mathcal{I} \wedge \neg\mathcal{A} \Rightarrow \widetilde{\mathcal{B}}, \text{ is universally true}$$

The role of the loop's body h is to *preserve the invariant*; that is

$$\langle \mathcal{I} \Rightarrow \mathcal{I}' \rangle$$

We can assume that $\mathcal{A}$ is true to start with, so h needs to implement

$$\langle \mathcal{A} \wedge \mathcal{I} \Rightarrow \mathcal{I}' \rangle$$

However, h should not change variables whose initial value is used in $\mathcal{B}$. We have

$$h = \langle \mathcal{A} \wedge \mathcal{I} \Rightarrow \mathcal{I}' \wedge \bar{x}' = \bar{x} \rangle$$

To ensure termination, the loop's body should also decrease a bound.

Exercise 61 Show that $g \sqsubseteq \langle A \rangle \Rightarrow (h; g)$

4.2 Examples of using loop invariants

4.2.0 Square Root by Binary Search

The floor $\lfloor a \rfloor$, of a real number a is the largest integer that is not larger than a. We have $\lfloor a \rfloor < i$ iff $a < i$, for all integers i and real numbers a.

The problem is (again) to find the integer part of the square root of a nonnegative real number x: $f = \langle y'^2 \leq x < (y'+1)^2 \rangle$. Rewrite f as

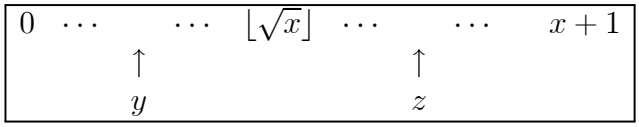
$$\begin{aligned} f &= \left\langle y'^2 \leq x < (y'+1)^2 \right\rangle \\ &= \text{Monotonicity of squaring nonnegative reals} \\ &\langle y' \leq \sqrt{x} < y'+1 \rangle \\ &= \text{Facts about floor} \\ &\langle y' \leq \lfloor \sqrt{x} \rfloor < y'+1 \rangle \\ &= \\ &\langle y' = \lfloor \sqrt{x} \rfloor \rangle \end{aligned}$$

Our plan is to use the method of invariants.

Picking an invariant. The first thing to do is to pick an invariant. We obtain $\mathcal{I}$ by replacing the expression $y'+1$ in f with a new variable, z, and erasing all primes.

$$\begin{aligned} \mathcal{I} &: y \leq \lfloor \sqrt{x} \rfloor < z \\ g &= \langle \mathcal{I} \Rightarrow y' = \lfloor \sqrt{x} \rfloor \rangle \end{aligned}$$

In terms of sets $\mathcal{I} = (\lfloor \sqrt{x} \rfloor \in \{y, ..z\})$. The invariant says that the answer we seek is somewhere between y (inclusive) and z (exclusive)



It is a fact that, for all $x \in \mathbb{N}$

$$0 \leq \lfloor\sqrt{x}\rfloor < x + 1$$

So

$$\begin{array}{l} f \\ = \text{Fact just mentioned} \\ \langle 0 \leq \lfloor\sqrt{x}\rfloor < x+1 \Rightarrow y' = \lfloor\sqrt{x}\rfloor\rangle \\ = \text{Substitution} \\ y, z := 0, x+1; g \end{array}$$

Picking a Guard When $z = y + 1$, according to the invariant, $\lfloor\sqrt{x}\rfloor$ is in the set $\{y, ..y + 1\} = \{y\}$. That is $z = y + 1$ together with the invariant implies that $\lfloor\sqrt{x}\rfloor = y$, and so there is nothing left to do:

$$\langle z = y + 1 \wedge \mathcal{I} \Rightarrow y' = \lfloor\sqrt{x}\rfloor\rangle \sqsubseteq \textbf{skip}$$

So use $z \neq y + 1$ as the loop guard. Given the invariant, this is the same as $z > y + 1$. So we have

$$\begin{array}{l} g \\ \sqsubseteq \\ \textbf{if } z > y+1 \textbf{ then } \langle z > y+1\rangle \Rightarrow g \textbf{ else } \langle z = y+1\rangle \Rightarrow g \\ \sqsubseteq \\ \textbf{if } z > y+1 \textbf{ then } \langle z > y+1\rangle \Rightarrow g \textbf{ else skip} \end{array}$$

Preserving the invariant We need a body that preserves the invariant but doesn't change x.

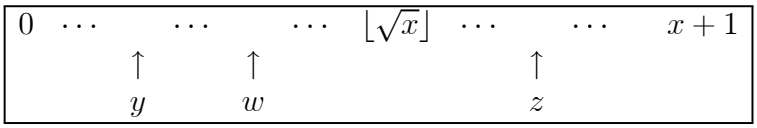
$$h = \left\langle \lfloor\sqrt{x}\rfloor \in \{y, ..z\} \wedge z > y + 1 \Rightarrow \left\lfloor\sqrt{x'}\right\rfloor \in \{y', ..z'\} \wedge x' = x \right\rangle$$

The idea is to consider two parts of the set $\{y, ..z\}$. If $w \in \{y, ..z\}$, then $\{y, ..w\} \cup \{w, ..z\} = \{y, ..z\}$. Since (according to the invariant) $\lfloor\sqrt{x}\rfloor \in$

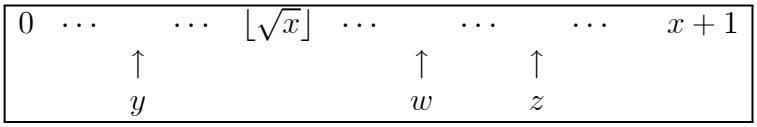
$\{y, ..z\}$, $\lfloor\sqrt{x}\rfloor$ must be in one of the subsets $\{y, ..w\}$ and $\{w, ..z\}$. If we can figure out which one, we can preserve the invariant by setting y and z to the upper and lower bounds of the appropriate subset. If our bound expression is $z - y$, i.e. the size of the set $\{y, ..z\}$, we should pick w so that $y < w < z$.

The fastest way to make progress is to pick w about half way between y and z. Since, at the start of the body, $z > y + 1$, $z - y$ is at least 2, so $\lfloor (z - y)/2 \rfloor$ is at least 1. Let w stand for $y + \lfloor (z - y)/2 \rfloor$. Note that $y < w < z$, as needed. We have $\{y, ..z\} = \{y, ..w\} \cup \{w, ..z\}$. From $\mathcal{I}$ we have $\lfloor\sqrt{x}\rfloor \in \{y, ..w\} \vee \lfloor\sqrt{x}\rfloor \in \{w, ..z\}$

If $w \leq \lfloor\sqrt{x}\rfloor$ then $\lfloor\sqrt{x}\rfloor \in \{w, ..z\}$, and we can set y to w without making the invariant false.



If $w > \lfloor\sqrt{x}\rfloor$ then $\lfloor\sqrt{x}\rfloor \in \{y, ..w\}$, and we can set z to w without making the invariant false.



In both cases the bound $z - y$ is decreased by at least one. We have

$$
\begin{array}{ll}
& h \\
& = \\
& \langle \mathcal{I} \wedge z > y + 1 \Rightarrow \mathcal{I}' \wedge x' = x \rangle \\
\sqsubseteq & \textbf{if } w \leq \lfloor\sqrt{x}\rfloor \\
& \textbf{then } y := w \\
& \textbf{else } z := w
\end{array}
$$

But, we can't use the test $w \leq \lfloor\sqrt{x}\rfloor$, as we can't easily calculate $\lfloor\sqrt{x}\rfloor$. (If we could easily calculate $\lfloor\sqrt{x}\rfloor$ there wouldn't be any point designing this algorithm.) Can we find an equivalent expression? Yes:

$$
\begin{array}{l}
w \leq \lfloor \sqrt{x} \rfloor \\
= \text{ Since } w \text{ is an integer} \\
w \leq \sqrt{x} \\
= \text{ Squaring both sides.} \\
w^2 \leq x
\end{array}
$$

The final algorithm is

$$
\begin{array}{ll}
f & \\
\sqsubseteq & y, z := 0, x+1; \\
& \text{// inv. } y \leq \lfloor \sqrt{x} \rfloor < z \\
& \textbf{while } z > y+1 \\
& \textbf{do } \textbf{var } w := y + \lfloor (z-y)/2 \rfloor \cdot \\
& \quad \textbf{if } w^2 \leq x \\
& \quad \textbf{then } y := w \\
& \quad \textbf{else } z := w
\end{array}
$$

Let's take a moment to look back at how the algorithm works. The invariant says the desired answer is in the set $\{y, y+1, ..., z-1\}$. We call this set the "search space"

In each iteration of the loop, we reduce the size of the search space by roughly 2. This technique is called *binary search*.

Since the size of the search space is roughly halved with each iteration, and the initial size of the search space is x, the number of iterations is roughly $\log_2 x$. For large x, $\log_2 x$ is considerably smaller than $\sqrt{x}$. Consider $x = 10^{12} \cong 2^{40}.\sqrt{x} = 10^6$. $\log_2 x \cong 40$.

4.2.1 Searching for a pattern

In this section we look at a somewhat more difficult problem.

Consider the problem of searching for a pattern sequence p within a target sequence t. For example, if p is "sip" and t is "Mississippi", then the answer is true. But if the pattern were "ips" the answer would be false. We are looking to see if there is a place i such that $p(0) = t(i)$ and $p(1) = t(i+1)$ and so on up to (and including) $p(\|p\|-1) = t(i+\|p\|-1)$. I'll write $t[i, ..i+j]$ to mean the segment of t that starts at i and that has length $\max(j, 0)$; For example if t is "abcd", then $t[1, ..3]$ is "bc".

4.2.1.0 Formalizing the problem

Let us define $\text{match}(i, j)$ to mean that the first j items of p match the j items of t starting at position i, where $0 \leq j \leq \|p\|$ and $0 \leq i$:

$$\text{match}(i, j) = (i + j \leq \|t\| \wedge p[0, ..j] = t[i, ..i + j])$$

Now we can specify our problem as

$$f = \langle (b' = \exists i \in \{0, .. \|t\| - \|p\| + 1\} \cdot \text{match}(i, \|p\|)) \rangle$$

For the rest of this section, I'll regard t and p as mathematical variables rather than state variables, and so I won't explicitly specify that they must not be changed by the initialization code and the loop body.

4.2.1.1 Developing an invariant and guard

Note that there are $\|t\| - \|p\| + 1$ places that the pattern could be found. Suppose we search from left to right using an index k that ranges from 0 up, and we have established that there is no match in the first k places. Then we know $\mathcal{I}0 \wedge \mathcal{I}1$ where

$$\begin{aligned} &\mathcal{I}0 \text{ is } k \geq 0 \\ &\mathcal{I}1 \text{ is } \neg\exists i \in \{0, ..k\} \cdot \text{match}(i, \|p\|) \end{aligned}$$

Suppose we further know that, at position k, the first j items of the pattern match, i.e. we also know $\mathcal{I}2 \wedge \mathcal{I}3$ where

$$\begin{aligned} &\mathcal{I}2 \text{ is } j \in \{0, .., \|p\|\} \\ &\mathcal{I}3 \text{ is } \text{match}(k, j) \end{aligned}$$

All of these invariants are easy to establish by setting k and j to 0.

What about a guard? If $k > \|t\| - \|p\|$, then, by $\mathcal{I}1$, we know that the pattern exists nowhere in the target and so b should be set to false; the loop can stop. On the other hand, if $j = \|p\|$, then, by $\mathcal{I}3$, there is a match at position k and so b should be set to true; again the loop can stop. This suggests that the guard $\mathcal{G}$ be $k \leq \|t\| - \|p\| \wedge j < \|p\|$.

In summary, so far we have

$$f \sqsubseteq k, j := 0, 0; \langle \mathcal{I} \Rightarrow \mathcal{I}' \wedge \mathcal{G}' \rangle ; b := j = \|p\|$$

where $\mathcal{I}$ is $\mathcal{I}0 \wedge \mathcal{I}1 \wedge \mathcal{I}2 \wedge \mathcal{I}3$.

4.2.1.2 Developing a loop body

We can implement $\langle \mathcal{I} \Rightarrow \mathcal{I}' \wedge \mathcal{G}' \rangle$ with a loop

$$\textbf{while } \mathcal{G} \textbf{ do } h$$

where $h = \langle \mathcal{G} \wedge \mathcal{I} \Rightarrow \mathcal{I}' \rangle$, provided we also decrease a bound, so it remains to implement h while decreasing a bound.

If $\mathcal{G} \wedge \mathcal{I}$ holds, we have $0 \leq k \leq \|t\| - \|p\|$ and $0 \leq j < \|p\|$, so it is safe to compare $t(k+j)$ with $p(j)$; neither subscript is out of bounds. If $t(k+j) = p(j)$, incrementing j preserves all invariants. If $t(k+j) \neq p(j)$, incrementing k preserves all invariants if we also set j to 0. Thus we have

$$h \sqsubseteq \textbf{if } t(k+j) = p(j) \textbf{ then } j := j+1 \textbf{ else } k, j := k+1, 0$$

The whole algorithm is

$k, j := 0, 0;$
while $k < \|t\| - \|p\| + 1 \wedge j < \|p\|$ **do**
    if $t(k+j) = p(j)$ **then** $j := j+1$ **else** $k, j := k+1, 0;$
$b := j = \|p\|$

As a bound we can use $(\|t\| - k)\,\|p\| + \|p\| - j$.

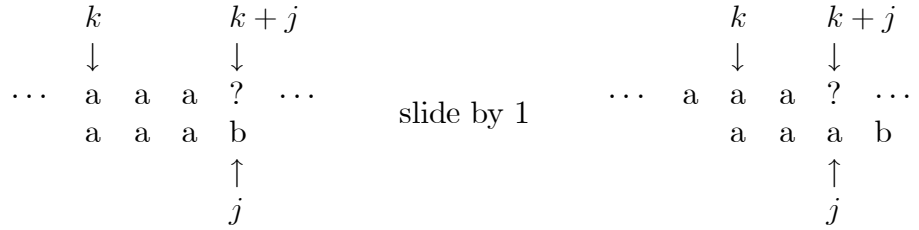
4.2.1.3 Speed

Is this a good algorithm? Consider for example a pattern "aaaaaaaaab" and a target than is "aa$\cdots$a". The algorithm behaves quite badly in this case: it makes 10 comparisons for each value of $k \in \{0, .., \|t\| - 10\}$. In general, the algorithm makes $\|p\| \times (\|t\| - \|p\| + 1)$ item comparisons, in the worst case.

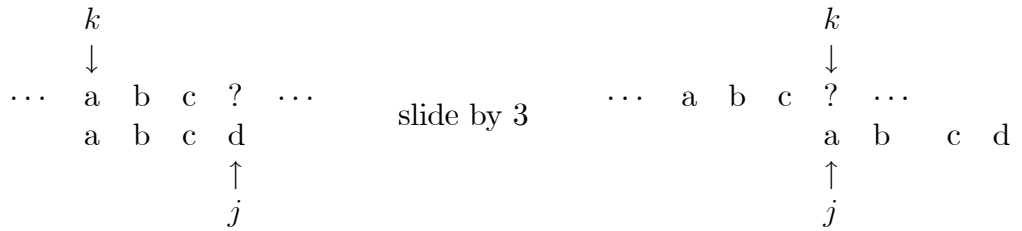
4.2.1.4 Improving the algorithm

Is there a better way? When there is a match, it seems hard to improve on advancing j. When there is a failure and $j = 0$, the best way to make progress is to increment k, while leaving j at 0. What if there is a failure when $j > 0$? The algorithm has already made j successful comparisons and that tells a lot about the target string that is simply forgotten if k is incremented and j is set to 0. We will look at three examples to figure out a way to use that information

Consider the case where the pattern p is "aaab". Suppose that, for some t and k, the match fails on the "b" —i.e. we have $j = 3$ and $t(k+3) \neq p(3)$. The algorithm above sets j back to 0 and k to $k+1$, so that the next comparison will be between $t(k+1)$ and $p(0)$. We can see that $p(0)$ is "a" by looking at p. We can see that $t(k+1)$ is "a" by consider $\mathcal{I}3$ which says that $p[0,..j] = t[k,..k+j]$ and thus that $t(k+1) = p(1)$. Since both items are "a", there is no need to compare them. Similarly, both $p(1)$ and $t(k+2)$ are "a" and so there is no reason to compare them. We can just set j to 2 and k to $k+1$.



Now consider the case where the pattern p is "abcd". Suppose that for some t and k, the match fails on the "d" —i.e., we have $j = 3$ and $t(k+3) \neq p(3)$. From $\mathcal{I}3$ we know that $p[0,..j] = t[k,..k+j]$. And thus $t[k,..k+3]$ is "abc". There is no point trying for a match at position $k+1$; we know that $t(k+1)$ contains a "b", while $p(0)$ contains an "a". Similarly there is no point trying for a match at position $k+2$, as we know that $t(k+2)$ is a "c". In this case we can set k to $k+3$ and j to 0.



Finally, consider the case where the pattern p is "ababc" and the match fails on the "c" when j is 4. This tells us that $t[k+1,..k+4]$ is "bab". So there is no point sliding the pattern 1 place to the right. But if we slide it two places to the right, the first two items of p will match. So we should

increment k by 2 and set j to 2.

$$\begin{array}{cccccccc} & k & & & & k+j & & \\ & \downarrow & & & & \downarrow & & \\ \cdots & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & ? & \cdots & \\ & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} & & \\ & & & & & \uparrow & & \\ & & & & & j & & \end{array} \qquad \text{slide by 2}$$

Note that in all three of these cases, we increment k and decrement j by the same amount so that $j + k$ remains the same

Given the pattern p, we can work out for each $j \in \{1, .. \|p\|\}$, how much can we slide the pattern in case the match process fails at point j without breaking any invariants. This will be the smallest number $m > 0$ such that $p[0, ..j - m] = p[m, ..j]$; this m is in $\{1, .., j\}$. For each $j \in \{1, .. \|p\|\}$, let $m_j = (\min m \in \{1, .., j\} \mid p[0, ..j - m] = p[m, ..j])$. For $j \in j \in \{1, .. \|p\|\}$, we have

$$\begin{array}{ll} & \mathcal{I}3 \\ = & \\ & t[k, ..k + j] = p[0, ..j] \\ \Rightarrow & \\ & t[k + m_j, ..k + j] = p[m_j, ..j] \\ \Rightarrow & \text{Since } p[0, ..j - m_j] = p[m_j, ..j] \\ & t[k + m_j, ..k + j] = p[0, ..j - m_j] \\ = & \\ & \mathcal{I}3\,[k, j : k + m_j, j - m_j] \end{array}$$

This means we will not violate $\mathcal{I}3$ by assigning $k, j := k + m_j, j - m_j$. $\mathcal{I}1$ will not be violated by this assignment, since, if it were, we could show that m is not minimum. The other invariants are trivially not violated.

The new loop body is

$h \sqsubseteq$ **if** $t(k{+}j) = p(j)$ **then** $j := j{+}1$ **else if** $j = 0$ **then** $k := k{+}1$ **else** $k, j := k{+}m_j, j{-}m_j$

We can precompute the value of m_j, based only on the pattern p.The improved algorithm is

var $m : \{1, .. \|p\|\} \xrightarrow{\text{tot}} \{1, .. \|p\|\} \cdot$
for each $j \in \{1, .. \|p\|\}$
do $m(j) := (\min m \in \{1, .., j\} \mid p[0, ..j - m] = p[m, ..j])$
$k, j := 0, 0;$
while $k < \|t\| - \|p\| + 1 \wedge j < \|p\|$ **do**
    if $t(k + j) = p(j)$ **then** $j := j + 1$
    else if $j = 0$ **then** $k := k + 1$
    else $k, j := k + m(j), j - m(j);$
$b := j = \|p\|$

Is this really more efficient? Since $m_j > 0$, we can see that each iteration decreases $2(\|t\| - k) + (\|p\| - j)$. In total the number of item comparisons is no more than $2\|t\| + \|p\|$. There is also some time required to compute the m_j values based on p, but this time depends only on $\|p\|$, not on $\|t\|$. For longer patterns this represents a significant improvement in time.

Let's reflect a bit on what we had to consider and what we didn't have to consider in making this improvement. The invariant for the improved algorithm is the same as the invariant for the earlier version. So, we could confidently make changes to the body of the loop, subject only to the constraint that it preserves the invariant. In making these changes we did not have to think about the guard or the initialization or the code after the loop, nor to come up with a different invariant. The only thing we had to think about was how to get from one state where the invariant and the guard both hold to a state where the invariant holds.

Exercise 62 Show in detail that invariant $\mathcal{I}1$ is preserved by the improved loop body.

Exercise 63 Describe in detail an algorithm to compute the values of m_j into an array.

Exercise 64 There is still information not used in the improved algorithm. When there is a failure, we not only know that $t[k, ..k + j] = p[0, ..j]$, but we also know that $t[k + j] \neq p[j]$. Can we also make use of this additional fact?

Exercise 65 Suppose that the items can be used to index an array. When

a match fails, can we use the value of $t[j + k]$ together with the value of j as the indeces to a 2 dimensional array that gives information about how to increment k and alter j? It should be possible, then, to create an algorithm that only reads each item of t once.

4.3 Finding invariants

[[To do: Add material from slides 6a]]

Chapter 5

Data Transformation

[[TO DO Rewrite this chapter. Currently it is just my slides.]]

5.0 A slightly faster (and smaller) square root

Consider the square root problem

$$f = \left\langle y'^2 \leq x < (y'+1)^2 \right\rangle$$

and its solution

$$\begin{aligned}
&\mathcal{I} \text{is } y^2 \leq x \\
&g = (\langle \mathcal{I} \rangle \Rightarrow f) \\
&f \sqsubseteq y := 0; g \\
&g \sqsubseteq \textbf{while } (y+1)^2 \leq x \textbf{ do } y := y+1
\end{aligned}$$

On many computers, multiplications are slow. In hardware, multipliers are large, slow, or both.

Do we really need to multiply?

Introduce a new variable z and redefine the invariant as

$$\mathcal{I} : y^2 \leq x \wedge z = (y+1)^2$$

As before define

$$g = (\langle \mathcal{I} \rangle \Rightarrow f)$$

Now we must change the two places where the invariant is established. Note that

$$(y+2)^2 = y^2 + 4y + 4 = (y+1)^2 + 2(y+1) + 1$$

We get

$$\begin{array}{ll} f \sqsubseteq & y, z := 0, 1; g \\ g \sqsubseteq & \textbf{while } z \leq x \\ & \textbf{do } y, z := y+1, z + 2(y+1) + 1 \end{array}$$

Although there is still a multiplication by 2, such a multiplication is fast in software and trivial in hardware. (Just shift the binary representation to the left.)

[Exercise: Do this derivation in detail.]

In both the above solutions we *augmented* the state space with a variable z and constrained the relationship of z to the other variables by strengthening the invariant.

This is an example of a **data transformation**. In a data transformation, we replace one set of variables with another while specifying an invariant relationship between the two state spaces.

In both examples we replaced {"x", "y"} with {"x", "y","z"} the added relationships being

$$z = (y+1)^2$$

and

$$y \leq \lfloor \sqrt{x} \rfloor < z$$

respectively.

5.0..5 A challenge:

The Square Root by Binary Search algorithm above still has a multiplication operation in each iteration. For hardware implementation, this will use up considerable time and area, for software implementation, it uses time.

Can you eliminate the multiplication in the Square Root by Binary Search? Hint: Use data transformation.

- Add one or more variables to track quantities that are expensive to calculate.
- Strengthen the invariant to indicate the relationship between these variables and the quantities they track.

5.0.0 An Abstract Binary Search algorithm

We can abstract away from the particulars of the square root problem to obtain a general search problem. Suppose G is a constant, nonempty, "goal set". (In our square root application G is $\{\lfloor\sqrt{x}\rfloor\}$.)

By *constant*, I mean that it does not depend on any variables that are changed.

We wish to find at least one member of the goal set.

$$f = \langle S' \subseteq G \wedge S' \neq \emptyset \rangle$$

We can abstract away from the particulars of the solution to the square root problem to obtain an *Abstract Binary Search* algorithm. The key is the invariant, which says that a set variable S always contains at least one member of the goal set.

$$\begin{array}{l}
\mathcal{I} : S \cap G \neq \emptyset \\
g = \langle \mathcal{I} \Rightarrow f \rangle \\
f \sqsubseteq S := \text{some set such that } S \cap G \neq \emptyset \ ; \ g \\
g \sqsubseteq \ \textbf{while } |S| > 1 \\
\qquad \textbf{do var } S_0,S_1 \mid S_0 \cup S_1 = S \cdot \\
\qquad\quad \textbf{if } S_0 \cap G \neq \emptyset \textbf{ then } S := S_0 \\
\qquad\quad \square\ S_1 \cap G \neq \emptyset \textbf{ then } S := S_1
\end{array}$$

The **var** construct introduces two new variables S_0 and $S_1$and initializes them such that $S_0 \cup S_1 = S$ before executing the body of the var command.

Note that either $S_0 \cap G \neq \emptyset$ or $S_1 \cap G \neq \emptyset$ or both since

$$
\begin{array}{l}
S_0 \cap G \neq \emptyset \vee S_1 \cap G \neq \emptyset \\
= \text{De Morgan} \\
\neg \left((S_0 \cap G = \emptyset) \wedge (S_1 \cap G = \emptyset) \right) \\
= \text{Since } A = \emptyset \wedge B = \emptyset \text{ iff } A \cup B = \emptyset \\
\neg \left((S_0 \cap G) \cup (S_1 \cap G) = \emptyset \right) \\
= \text{Distributivity} \\
\neg \left((S_0 \cup S_1) \cap G = \emptyset \right) \\
= \text{Since } S_0 \cup S_1 = S \\
= \neg \left(S \cap G = \emptyset \right) \\
\text{Definition of } \mathcal{I} \\
\mathcal{I}
\end{array}
$$

The loop bound is the size of S, so we should ensure that both S_0 and S_1 are smaller than S. For efficiency it is best if S_0 and S_1 are disjoint ($S_0 \cap S_1 = \emptyset$) and approximately the same size. In that case the loop will iterate $\Theta(\log |S|)$ times.

5.0.0.0 Applying the general square root algorithm.

Our Square Root by Binary Search algorithm can be obtained from the Abstract Binary Search algorithm by the following data transform

$$
\begin{array}{c}
G = \left\{ \left\lfloor \sqrt{x} \right\rfloor \right\} \\
S = \{y, ..z\} \\
S_0 = \{y, ..w\} \\
S_1 = \{w, ..z\} \\
\text{where } w = y + \left\lfloor (z - y)/2 \right\rfloor
\end{array}
$$

Chapter 6

Placeholder

Typeset January 22, 2018

Part 1

Formal Languages and Models of Computation

Typeset January 22, 2018

Letter conventions for this part of the book

I'll use variables as follows

S an alphabet

$a, b, c, d, e \in S$

$s, t, u, v, w \in S^*$

$M, N \subseteq S^*$

x, y regular expressions over S

Q a set of states

$p, q, r \in Q$

$R, F \subseteq Q$

T a set of transitions

V a set of nonterminal symbols

$A, B, C, D, E \in V$ (A is also used for finite state machines)

$\alpha, \beta, \gamma, \delta, \eta, \kappa \in (V \cup S)^*$

P a set of productions

Chapter 7

Strings, Languages, and Regular Expressions

In this chapter, we will be concerned with what are called "formal languages". Now the word "language" usually means something quite complex with syntax, semantics, and pragmatics. However a "formal" language is a fairly simple thing, simply a set of sequences. We'll worry about questions such as: "How can you describe a language?" "How do these ways of describing languages relate?" "Are some more powerful than others?" "How can you "recognize" whether a particular sequence is in a language?"

Although the idea of a language is simple —"a set of sequences"—, individual languages can be quite complex. That is they can be difficult to describe or difficult to recognize. In fact, formal language theory was one of the first places where ideas about complexity were studied and remains important to understanding complex systems. One particularly intriguing question is whether we can describe a language is so complex that recognizing members of the language is beyond the capability of any imaginable computer.

7.0 Strings

alphabet
symbol

Definition 66 An *alphabet* set S is simply a nonempty set of things we will call *symbols*.

So an alphabet is a set of symbols and a symbol is some member of an

alphabet. Most of the time, we'll make use only of finite alphabets, such as $\{\text{'0'}, \text{'1'}\}$ of $\{\text{'a'}, \text{'b'}, \text{'c'}\}$.

A string is finite sequence of symbols.

string

Definition 67 A *string* of length n over an alphabet S is a function from $\{0, ..n\} \xrightarrow{\text{tot}} S$.

We use the notation S^n for the set of all strings of length n over alphabet S. By definition

$$S^n = \left(\{0, ..n\} \xrightarrow{\text{tot}} S\right)$$

length

If $s \in S^n$ we say that its *length* is n and write $\|s\| = n$.

Note that there is one element of $\{0, ..0\} \xrightarrow{\text{tot}} S$. This is the function that has $\emptyset$ as its domain, S as its co-domain, and $\emptyset$ as its graph. Regardless of S

empty string

we'll write this function as ϵ.[0].

There are $|S|$ elements of S^1. For example, if $S = \{\text{'a'}, \text{'b'}, \text{'c'}\}$, the three elements of S^1 are

$$(\{0\}, S, \{0 \mapsto \text{'a'}\}) = [\text{'a'}]$$
$$(\{0\}, S, \{0 \mapsto \text{'b'}\}) = [\text{'b'}]$$
$$(\{0\}, S, \{0 \mapsto \text{'c'}\}) = [\text{'c'}]$$

We will write these strings as "a", "b", and "c".There are 4 elements of S^2, where $S = \{0, 1\}$.

$$\{[0, 0], [0, 1], [1, 0], [1, 1]\}$$

In general there are $|S|^n$ elements of S^n.

The set of all strings over S is

$$S^* = \bigcup n \in \mathbb{N} \cdot S^n$$

If $S = \{\text{'a'}, \text{'b'}, \text{'c'}\}$ then S^* is an infinite set

$$\begin{aligned} S \quad = \quad & \{\epsilon, \text{"a"}, \text{"b"}, \text{"c"}, \text{"aa"}, \text{"ab"}, \text{"ac"}, \text{"ba"}, \text{"bb"}, \text{"bc"}, \text{"ca"}, \text{"cb"}, \text{"cc"}, \\ & \text{"aaa"}, \text{"aab"}, \text{"aac"}, \text{"aba"}, \text{"abb"}, \text{"abc"}, \text{"baa"}, \text{"bab"}, \text{"bac"}, \cdots\} \end{aligned}$$

Note that, while the size of S^* is infinite, the length of each element of S

concatenate

is finite.

[0]This is the greek letter epsilon. Please don't confuse it with the set membership symbol $\in$. Thus we can write that $\epsilon \in S^0$.

We can *concatenate* two strings s and t to get a string $s\hat{}t$. Such that

$$\begin{aligned} s\hat{}t &\in S^{\|s\|+\|t\|} \text{ and thus} \\ \|s\hat{}t\| &= \|s\| + \|t\| \text{. Furthermore} \\ (s\hat{}t)(i) &= s(i), \text{ if } 0 \leq i < \|s\| \text{, and} \\ (s\hat{}t)(\|s\| + i) &= t(i), \text{ if } 0 \leq i < \|t\| \end{aligned}$$

For example “ab”$\hat{}$“ca” = “abca”.

7.0.0 Languages

A *language* or *formal language* over S is any subset of S^*.

We can overload the concatenation operation to languages; if M and N are languages over S.

$$M\hat{}N = \{s \in M, t \in N \cdot s\hat{}t\}$$

For example if

$$\begin{aligned} M &= \{\text{“a”}, \text{“aa”}, \text{“ab”}\} \text{ and } N = \{\epsilon, \text{“b”}\} \text{ then} \\ M\hat{}N &= \{\text{“a”}, \text{“aa”}, \text{“ab”}, \text{“aab”}, \text{“abb”}\} \end{aligned}$$

For each $i \in \mathbb{N}$ we can consider M^i to be $M\hat{}M\hat{}\cdots\hat{}M$ where there are i Ms. A string is in M^i exactly if it can be cut into i pieces, each of which is in M.

For example if

$$\begin{aligned} M &= \{\text{“a”}, \text{“aa”}, \text{“ab”}\} \text{ then} \\ M^1 &= \{\text{“a”}, \text{“aa”}, \text{“ab”}\} \\ M^2 &= \{\text{“aa”}, \text{“aaa”}, \text{“aab”}, \text{“aaaa”}, \text{“aaab”}, \text{“aba”}, \text{“abaa”}, \text{“abab”}\} \\ M^3 &= \{\text{“aaa”}, \text{“aaaa”}, \text{“aaab”}, \text{“aaaaa”}, \text{“aaaab”}, \text{“aaba”}, \text{“aabaa”}, \text{“aabab”}, \\ &\quad \text{“aaaaaa”}, \text{“aaaaab”}, \text{“aaaba”}, \text{“aaabaa”}, \text{“aaabab”}, \\ &\quad \text{“abaa”}, \text{“abaaa”}, \text{“abaab”}, \text{“abaaaa”}, \text{“abaaab”}, \text{“ababa”}, \text{“ababaa”}, \text{“ababab”}\} \end{aligned}$$

Note that, while the size of M is 3, the size of M^2 is 8, not 9. This is because the string “aaa” is generated in two different ways.

By convention, M^0 is $\{\epsilon\}$. So we can define M^i by

$$\begin{aligned} M^0 &= \{\epsilon\} \\ M^{i+1} &= M\hat{}M^i, \text{ for all } i \in \mathbb{N} \end{aligned}$$

We can define the *Kleene closure*, M^*, of M as the set of all finite strings that can be generated by catenating together strings from M.[1] So if $M = \{\text{“a”}, \text{“bb”}\}$ then

$$M^* = \{\epsilon, \text{“a”}, \text{“bb”}, \text{“aa”}, \text{“abb”}, \text{“bba”}, \text{“bbbb”}, \text{“aaa”}, \text{“aabb”}, \text{“abba”}, \text{“abbbb”}, \text{“bbaa”}, \text{“bbabb”}, \text{“bbbba”}, \text{“bbbbbb”}, \cdots\}$$

A string is in M^* exactly if it can be cut into a finite number of pieces, each of which is in M.

Formally we can define

$$M^* = \bigcup i \in \mathbb{N} \cdot M^i$$

Note that $\epsilon \in M^*$ regardless of M. For example $\emptyset^* = \{\epsilon\}$.

7.1 Regular language

Given that M and N are languages over S, consider three ways to make languages

- Union: $M \cup N$ — the language that contains all strings either in M or in N
- Concatenation: $M \hat{} N$ — the language of strings s that can be split into two parts $s = t \hat{} w$, where $t \in M$ and $w \in N$.
- Kleene closure: M^*— the language of strings s that can be split into some number (including 0) of parts, each of which is in M.

If we start with one of more finite languages we can build more languages by applying one or more of the operations above. Here are some examples useing the alphabet $\{0, 1\}$:

- $\{[0]\} \cup \{[1]\} \cup \{\epsilon\} = \{\epsilon, [0], [1]\}$
- $\{[0]\} \hat{} \{[1]\} = \{[0, 1]\}$

[1] After the logician Stephen Kleene. “Kleene” is pronounced “cleany”.

- $\{[0]\}^* = \{\epsilon, [0], [0,0], [0,0,0], \ldots\}$
- $\{[0]\}^* \cup \{[1]\}^* = \{\epsilon, [0], [1], [0,0], [1,1], \ldots\}$
- $\{[0]\}^* \hat{} \{[1]\}^* = \{\ \epsilon$
  $[0], [1],$
  $[0,0], [0,1], [1,1],$
  $[0,0,0], [0,0,1], [0,1,1], [1,1,1], \ldots\}$

A *regular language* over S is any language that can be formed from the finite languages over S using the three operations of union, concatenation, and Kleene closure. regular language

Next we look at a nice syntax for describing regular languages.

7.2 Regular expressions

A *regular expression* over S is a kind of expression that defines a language over S. Each regular expression itself is simply a string in a language over an alphabet that includes S and the seven additional symbols $\{\underline{\epsilon}, \underline{\emptyset}, \underline{|}, \underline{;}, \underline{*}, \underline{(}, \underline{)}\}$. [2] regular expression

7.2.0 Syntax

Given a set of symbols S, a regular expression over S is a string formed by finite application of the following six rules:

- For each $a \in S$, $\underline{a}$ is a regular expression
- $\underline{\epsilon}$ is a regular expression.
- $\underline{\emptyset}$ is a regular expression.
- If x and y are regular expressions then
  - $\underline{(}\hat{}x\hat{}\ \underline{|}\ \hat{}y\hat{}\underline{)}$ is a regular expression (alternation)
  - $\underline{(}\hat{}x\hat{}\ \underline{;}\hat{}y\hat{}\underline{)}$ is a regular expression (concatenation)

[2] For simplicity we'll assume that S contains none of the seven symbols in $\{\underline{\epsilon}, \underline{\emptyset}, \underline{|}, \underline{;}, \underline{*}, \underline{(}, \underline{)}\}$. If S is set of characters, we'll consider, for example the character '(' to be distinct from the symbol $\underline{(}$.

– $\underline{(\hat{}x\hat{}^*)}$ is a regular expression (repetition)

Some examples:

- $\underline{0}$ is a regular expression over $\{0, 1\}$.
- $\underline{(0; (((1^*) \mid (0^*)); 1))}$ is a regular expression over $\{0, 1\}$.
- $\underline{((\text{`('}^*); (\text{`)'}^*))}$ is a regular expression over $\{\text{`('}, \text{`)'}\}$.

Underlining:

- I've underlined regular expressions to make it clear that that is what they are. For example, $\underline{\epsilon}$ is a regular expression and so is a string of length 1 in the language of regular expressions, whereas ϵ is a string of length 0. I could have written regular expressions in double quotes (e.g. "ϵ") to emphasize that regular expressions are strings, but that leads to some awkwardness.
- Some programming languages use the similar convention of writing regular expressions between slashes. E.g. `/(b|(a*))/` is a regular expression in Perl, whereas I'll write $\underline{(\text{`b'} \mid (\text{`a'}^*))}$.
- When it's clear that we have a regular expression, the underlining isn't needed.

Parentheses:

- Parentheses can be omitted with the understanding that * has higher precedence than ; and that ; has higher precedence than |. Thus.

  $\underline{(a; (b^*))}$ can be abbreviated by $\underline{a; b^*}$,

  whereas

  $$\underline{(a; b)^*}$$

  needs its parentheses. And

  $\underline{((a; b) \mid (c; d))}$ can be abbreviated by $\underline{a; b \mid c; d}$,

  whereas

  $$\underline{a; (b \mid c); d}$$

  needs its parentheses.

- We can write $\underline{x; y; z}$ to mean $\underline{(x; y); z}$
- Similarly, we can write $\underline{x \mid y \mid z}$ to mean $\underline{(x \mid y) \mid z}$.
- Redundant parentheses can be added. E.g. $\underline{(((\text{`b'}) \mid ((\text{`a'})^*)))}$.

(The operators $|$, ; and * are analogous to $+$, $\times$, and exponentiation both in precedence and in some algebraic properties.)

7.2.1 Semantics

As mentioned above, regular expressions are used to define languages.

We can define the "meaning" of regular expressions over an alphabet S by defining a function L from regular expressions to languages.[3]

- $L(\underline{a}) = \{[a]\}$, for each $a \in S$.
- $L(\underline{\epsilon}) = \{\epsilon\}$
- $L(\underline{\emptyset}) = \emptyset$
- If x and y are regular expressions then
  - $L(\underline{(}x\underline{;}y\underline{)}) = L(x)\hat{}L(y)$
  - $L(\underline{(}x \underline{\mid} y\underline{)}) = L(x) \cup L(y)$
  - $L(\underline{(}x^*\underline{)}) = (L(x))^*$

[3] In mathematics we often gloss over the distinction between expressions and their meanings. For example, we might say "$(x+y)(x-y)$ is $x^2 - y^{2}$" and we might say "$x^2 - y^2$ is in sum of products form", but we would not say "$(x+y)(x-y)$ is in sum of products form". In the first case we mean that the meaning of $(x+y)(x-y)$ is the same as the meaning of $x^2 - y^2$. In the second case we mean that the expression $x^2 - y^2$ is in sum of products form.

When we refer to regular expressions, we really mean the expression itself. The variables x and y range over expressions, not (as in ordinary mathematics) meanings. The expressions $\underline{(\text{`a'} \mid \text{`b'})\,;\text{`c'}}$ is not the equal to the exression $\underline{\text{`a'};\text{`c'}\mid\text{`b'};\text{`c'}}$ even though they have the same meaning.

The reason for carefully distinguishing between expressions and meanings is that we are interested in various representations of languages. Regular expressions are just one. In the next chapter we will meet finite recognizers and we will want to be able to clearly express that the meaning of a regular expression is the same as the meaning of a finite recognizer.

If $s \in L(x)$ we say that x *describes* s or that x *recognizes* s. We also say that x describes the language $L(x)$. Two regular expression are *equivalent* if they describe the same language.

7.2.2 Examples and conventions

7.2.2.0 Money

Suppose that $S = \{\text{'\$','0','1','2','3','4','5','6','7','8','9',',','.','-'}\}$

We want to define strings representing amounts of money such as

"$0.05"

"$-12,345.67"

We can define a set of strings representing amounts of money as follows. Let x represent the regular expression

$$\underline{(\text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'})}$$

The regular expression

$$\underline{\text{'\$'}; (\text{'}-\text{'} \mid \epsilon); (x;x;x|x;x;x|x); (\text{','};x;x;x)^*; \text{'.'};x;x} \tag{0}$$

describes one convention for writing amounts of money in dollars and cents.

7.2.2.1 Conventions

To make regular expressions easier to use, we can adopt the following notational conventions

- First, when it is clear from context that we are dealing with a regular expression, we may leave out the underlining.
- Second, by analogy with multiplication, we may leave out the concatenation operator $\hat{}$. Leaving out the symbol does not affect the precedence of catenation. So xy^* means $x\hat{}y^*$, and both mean $x\hat{}\,(y^*)$.
- Third, a concatenation of a regular expression x with itself i times, may be written x^i. For example xxx may be written as x^3. Also x^1 is just a synonym for x and x^0 is just a synonym for ϵ.

- Fourth, an alternation $(x^j \mid x^{j+1} \mid \cdots \mid x^i)$, where $j < i$, may be written as x^i_j. For example, $(x \mid xx \mid xxx)$ may be written as x^3_1.
- Fifth, it follows that x^1_0 is an abbreviation for $x \mid \epsilon$, which we may also write as $x^?$.
- Sixth xx^* may be abbreviated by x^+.
- Seventh, the catenation of a series of symbols can be written as a string. E.g. 'a'^'b'^'c' may be written as "abc".
- Eighth, an alternation of symbols $(a \mid b \mid \cdots \mid c)$, where a, b, $\cdots$, c are successive symbols (e.g. in alphabetic order or numerical order), may be written as $[a - c]$. E.g.

  $$(\text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'})$$

  may be written as ['0'−'9'].
- Ninth, an alternation between all symbols is written as a dot: . .

Summarizing all notations we have

R.E.	**Describes ...**
ϵ	only the empty string.
$\emptyset$	no strings.
a	just the string with just an a.
s	just the string s.
$x\hat{}y$	any string described by x followed by one described by y.
$x\,y$	same as $x\hat{}y$.
$x \mid y$	strings described by x or y or both.
x^*	any catenation of 0 or more strings, each described by x.
x^+	any catenation of 1 or more strings, each described by x.
x^n	any catenation of exactly n strings, each described by x.
x^n_m	any catenation of from m to n strings, each described by x.
$x^?$	a string described by x or ϵ.
.	any string of length one.
$[a, b, c]$	any string of length one using only symbol a, b or c.
$[\hat{}a, b, c]$	any string of length one not using symbols a, b and c.
$[a - b]$	any string s of length one such that $a \leq s(0) \leq b$ in alphabetical order.

With these conventions, we can write (0) as

$$\text{`\$'}\ \text{`-'}^{?}\ [\text{`0'} - \text{`9'}]_1^3\ (\text{`,'}\ [\text{`0'} - \text{`9'}]^3)^*\ \text{`.'}\ [\text{`0'} - \text{`9'}]^2$$

(The dot here is in quotes and so means a single dot character, not an alternation between all characters.)

The conventions listed above are just abbreviations. Any regular expression using these conventions is easily rewritten as one that does not. For theoretical discussions, it is best to stick to the six ways of writing regular expression that were introduced in Section 7.2.0. In practical use, the conventions allow regular expressions that are shorter, easier to write, and often easier to read.

Regular expressions are found in various languages: programming languages such as Perl, JavaScript, Ruby, and awk; editing language such as sed and vi; schema languages such as XML's Document Type Descriptions; parser and lexical analyzer generators such as JavaCC and lex. These various languages have various and conflicting detailed conventions for writing regular expressions. The conventions used here are similar to those used in the JavaCC lexical analyzer generator.[4]

7.2.2.2 Identifiers

Suppose that $S = \{\text{`\_'},\text{`0'},\text{`1'},\ldots,\text{`9'},\text{`a'},\text{`b'},\ldots,\text{`z'},\text{`A'},\text{`B'},\ldots,\text{`Z'}\}$. The regular expression

$$([\text{`a'} - \text{`z'}] \mid [\text{`A'} - \text{`Z'}] \mid \text{\_})([\text{`a'} - \text{`z'}] \mid [\text{`A'} - \text{`Z'}] \mid [\text{`0'} - \text{`9'}] \mid \text{\_})^*$$

describes C++ identifiers.

7.2.2.3 Parity

Let $S = \{\text{`0'}, \text{`1'}\}$. Here is a regular expression that matches only strings with an even number of 1s.

$$(\text{`0'}^*\ \text{`1'}\ \text{`0'}^*\ \text{`1'})^*\ \text{`0'}^*$$

[4] A common abbrevation that we will not use is to allow the quotes around symbols to be omitted from some symbols. For example, in Perl (0) may written

```
\$-?[0-9]{1,3}(,[0-9]){3})*\.[0-9]{2}
```

The \$ and the . are 'quoted' using the backslash \. For this book, to keep things easier to read, we'll quote all character symbols.

7.2.2.4 String search

Let $S = \{\text{`\_'},\text{`0'},\text{`1'},\ldots,\text{`9'},\text{`a'},\text{`b'},\ldots,\text{`z'},\text{`A'},\text{`B'},\ldots,\text{`Z'}\}$. Let us define a regular expression that matches all strings that contain the substring "regular".

Recall that "." represents the choice between all elements of the alphabet. Now the regular expression is

$$.^* \text{"regular"} .^*$$

7.2.2.5 More string searches

Does a document contain any of the strings "woman", "women", "man", "men" ?

$$.^* (\text{"woman"} \mid \text{"women"} \mid \text{"man"} \mid \text{"men"}) .^*$$

or, equivalently,

$$.^* \text{"wo"}^? \text{`m'} (\text{`a'} \mid \text{`e'}) \text{`n'} .^*$$

Does a document contain the string "John" followed, after 0 or more intervening characters, by the string "Smith"?

$$.^* (\text{"John"}) .^* (\text{"Smith"}) .^*$$

7.2.2.6 C comments

In the C programming language, each comment starts with a '/' followed by a '*' and ends with another '*' followed by another '/'. In between there can be any number of '*'s and '/'s, but never a '*' immediately followed by a '/'. There can also be any number of other characters. Let x be [^'*','/'] so that $L(x) = S^1 - \{\text{"*"}, \text{"/"}\}$. Let y be [^'*'], so that $L(y) = S^1 - \{\text{"*"}\}$. Now (I claim) the regular expression

$$\text{"/*"}\ y^*\ \text{`*'}\ (x\ y^*\ \text{`*'} \mid \text{`*'})^*\ \text{`/'}$$

will match any C comment.

7.2.2.7 Sums

For this example we will use $S = \{\text{`0'}, \text{`1'}\}$. We'll represent sums in a way that is best illustrated by an example. Suppose we want to add 6 to 13. We

convert to binary to get $110 + 1101$; we pad out the both numbers with zeros to get $00110 + 01101$. Adding them up we get

$$\begin{array}{r} 00110 \\ +\ 01101 \\ \hline 10011 \end{array}$$

Now reading in column major order, we get the string

$$001\,010\,110\,101\,011$$

Now can we make a regular expression that matches strings such as this, i.e. strings that represent correct binary sums? I propose

$$(\text{“000”} \mid \text{“101”} \mid \text{“011”} \mid \text{“001”}\ (\text{“010”} \mid \text{“100”} \mid \text{“111”})^*\ \text{“110”})^+$$

7.3 Matching

For many regular expressions, it is easy to write a program that determines whether a string s is in the regular expression. For example, if we have a regular expression “ab”*;“bab”, we can do as follows: For any symbol c, let

$$\begin{array}{lll} \text{lookahead}(c) & \text{be} & \|s\| > 0 \wedge s(0) = c, \quad \text{and} \\ \text{consume}(c) & \text{be} & \textbf{if } \text{lookahead}(c) \textbf{ then } s := \text{tail}(s) \textbf{ else } f := \mathfrak{false} \end{array}$$

where $\text{tail}(s)$ is the sequence of length $\|s\| - 1$ such that $\text{tail}(s)(0) = s(1)$, $\text{tail}(s)(1) = s(2)$, and so on. Now the following algorithm matches the regular expression “ab”*^“bab” in the following sense: The algorithm sets flag f to $s \in L\left(\underline{\text{“ab}_\text{J}\text{*; “bab}_\text{J}}\right)$. I.e. the specification is $\left\langle f' = \left(s \in L(\underline{\text{“ab}_\text{J}\text{*; “bab}_\text{J}})\right)\right\rangle$

$f := \mathfrak{true}$;
while lookahead(‘a’) **do** (
    consume(‘a’) ;
    consume(‘b’)) ;
consume(‘b’) ;
consume(‘a’) ;
consume(‘b’) ;
$f := f \wedge \|s\| = 0$

Note how the structure of the algorithm follows the structure of the regular expression.

However this strategy does not always work, as it is sometimes necessary for the algorithm to look many characters ahead in order to determine whether to take one branch or another. Consider "ab"*^"ababa"^"ab"*. If we were to try to write an algorithm analogous to the one we used above, deciding whether to leave the while loop would require looking ahead six characters. In the next chapter we will develop algorithms to determine whether a string is matched by a regular expression; these algorithms will work with any regular expression.

Exercise 68 Write an algorithm to recognize C comments.

Exercise 69 Write an algorithm to recognize

$$(\text{'a'} \mid \text{'b'})^* \mid \text{"ab"}^* \text{'c'} \text{"ab"}^*$$

7.4 Equivalences of regular expressions

[[To be written]]

7.5 Regular languages and looking forward

Regular expressions provide a useful notation for describing languages. The set of languages that can be described using regular expressions over S is set of regular languages over S.

Theorem 70 A language M is a regular language exactly there is a regular expression x such that $L(x) = M$.

Exercise 71 Prove this theorem..

As shown by the examples, regular expressions can be a concise and usually clear notation for describing sets of strings. Importantly, regular expressions give a finite description of (at least some) infinite languages.

However some problems remain which will be explored in the following chapter. First, given a regular expression x, how can we write a program that takes a string as input and determines whether it is in the language described by x? If we can do that, we can no doubt do even better and produce a program that takes as input a regular expression x and a string s and determines whether $s \in L(x)$. Second, can all languages can be described by regular expression? I.e., is there a regular expression for each language, or are are there languages that no regular expression can describe? If the answer is the latter, then are there formalisms that describe the same set of languages and are there formalisms that can describe more languages?

7.6 Reversal

Define $s^\smile$ to be the reverse of string s, i.e. $\left\|s^\smile\right\| = \|s\|$ and $s^\smile(i) = s(\|s\| - 1 - i)$, for all $i \in \{0, .. \|s\|\}$. Now define the reverse of a language to be the language formed by reversing all its strings

$$M^\smile = \left\{s \in M \cdot s^\smile\right\}$$

We will prove that if M is regular then so is $M^\smile$. Now consider a reversal operation mapping regular expressions to regular expressions defined by

$$\begin{aligned}
\underline{\epsilon}^\smile &= \underline{\epsilon} \\
\underline{\emptyset}^\smile &= \underline{\emptyset} \\
\underline{a}^\smile &= \underline{a} \\
\left(\underline{(}\hat{}x\hat{}\underline{;}\hat{}y\hat{}\underline{)}\right)^\smile &= \underline{(}\hat{}y^\smile\hat{}\underline{;}\hat{}x^\smile\hat{}\underline{)} \\
\left(\underline{(}\hat{}x\hat{}\underline{|}\hat{}y\hat{}\underline{)}\right)^\smile &= \underline{(}x^\smile\hat{}\underline{|}\hat{}y^\smile\hat{}\underline{)} \\
\left(\underline{(}\hat{}x\hat{}\underline{*)}\right)^\smile &= \underline{(}\hat{}x^\smile\hat{}\underline{*)}
\end{aligned}$$

By induction we can prove that $L(x^\smile) = (L(x))^\smile$. If M is regular, there must be a regular expression x so that $L(x) = M$. Thus $M^\smile = (L(x))^\smile = L(x^\smile)$ and as $x^\smile$ is a regular expression, $M^\smile$ is a regular language. We say that regular languages are *closed under reversal*.

Chapter 8

Finite Recognizers

In this chapter, we will define a simple model of computation called a *finite recognizer*. At first, the finite recognizers we will describe are (potentially) nondeterministic (NDFR). This means that, in a sense, they rely on luck to work. Later, we will also look at deterministic finite recognizers (DFR), which could be described as not relying on luck. We will see that the set of languages that can be described by nondeterministic finite recognizers is exactly the same as the set of languages that can be described by deterministic finite recognizers. (I.e. luck is not needed.) Furthermore, this set of languages is also exactly the set of languages that can be described by regular expressions. We will call this set of languages, the *regular languages*.

Thus we will answer some of the questions posed at the end of the last chapter. In particular we will see how to develop algorithms that determine whether a string is in the language defined by a regular expression. Furthermore we will see that not every language can be described by a regular expression. And we will see some formalisms that are equivalent to that of regular expressions.

The crucial limitation of finite recognizers is that they have a finite memory. While they can remember the past, their capacity to remember is limited. Any particular automaton can remember only so much and no more. Some languages, it turns out, require machines that can remember an unbounded amount of information to be accurately recognized.

Here is the plan for the chapter.

- We'll start by defining nondeterministic finite recognizers (NDFRs).
- Then we'll see that any regular expression can be translated to an

equivalent NDFR. This will show that regular expressions have no more descriptive power than NDFRs.

- Next we'll see that given an NDFR, we can create an equivalent deterministic finite recognizer (DFR). This will show that NDFRs are no more powerful than DFRs. Since every DFRs is also an NDFR, it is trivial that DFRs are no more powerful than NDFRs. Thus NDFRs and DFRs are equivalent in their descriptive power.

- Finally we will see that, given an NDFR (or DFR), we can construct an equivalent regular expression.

Together these results establish the equivalence of all three formalisms: REs, NDFRs, and DFRs. Any language that can be described by one, can also be described by the other two. As we saw at the end of the last chapter, these languages are known as the regular languages.

Whether a language is regular or not is an important way of measuring its complexity. A language is regular exactly if there is a device with a fixed amount of memory that can process it (or at least recognize it). Computer chips (for example) have only a fixed amount of storage.[0] If a language is not regular, it can not be used at the input language of a computer chip.

8.0 Nondeterministic Finite State Recognizers

We will define a kind of finite state automaton for defining languages.

8.0.0 Syntax

Nondeterministic Finite State Recognizer
NDFR

Definition 72 A *Nondeterministic Finite State Recognizer (NDFR)* is a quintuple $A = (S, Q, q_{\text{start}}, F, T)$ consisting of

[0] The term "finite" is often used instead of "fixed". You can see this in the term "finite recognizer". Since the universe contains a finite amount of mass, it is clear that any digital device will have only a finite amount of memory. What is crucial here is that the amount of memory on a computer chip (and many other digital devices) is finite and fixed. The amount of memory it has at its disposal can not be increased in response to long input strings.

- A finite alphabet set S
- A finite set of states Q
- An *initial state* $q_{\text{start}} \in Q$
- A set of *accepting states* $F \subseteq Q$
- A set of *labelled transitions* $T \subseteq Q \times (S^1 \cup \{\epsilon\}) \times Q$

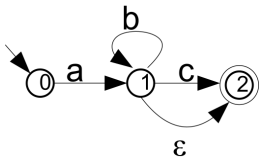
Example 73 As a first example of an NDFR we have $A = (S, Q, q_{\text{start}}, F, T)$ where

- $S = \{\text{'a'}, \text{'b'}, \text{'c'}\}$
- $Q = \{0, 1, 2\}$
- $q_{\text{start}} = 0$
- $F = \{2\}$
- $T = \{(0, \text{“a”}, 1), (1, \text{“b”}, 1), (1, \text{“c”}, 2), (1, \epsilon, 2)\}$

At the heart of an NDFR is an edge-labelled directed multigraph $(Q, T, \leftarrow, \rightarrow, \lambda)$. Each transition (q, s, r) is an edge from q to r labelled by s:

$$\overleftarrow{(q, s, r)} = q \qquad \overrightarrow{(q, s, r)} = r \qquad \lambda(q, s, r) = s$$

We can draw a picture of an NDFR by drawing this graph. We draw an arrow with no source to indicate the start state and the members of F drawn as double circles. Figure **??** shows the NDFR from Example 73.



An NDFR

8.0.1 Semantics

Each state $q \in Q$ of an NDFR $A = (S, Q, q_{\text{start}}, F, T)$ defines a language $L_A(q) \subseteq S^*$.

Two rules define the languages described by states:

- If $q \in F$ then $\epsilon \in L_A(q)$.
- If $(q, s, r) \in T$ then $\{s\}^\frown L_A(r) \subseteq L_A(q)$

Meta-rule: A string is in the language associated with a state iff it can be shown to be by a *finite* number of applications of the above rules.

Example 74 In the NDFR of Example 73, we have (by the first rule) $\epsilon \in L_A(2)$. By the second rule we have "c" $\in L_A(1)$ and $\epsilon \in L_A(1)$. By further application of the second rule using the transition $(1, \text{"b"}, 1)$ we have "bc" $\in L_A(1)$ and "b" $\in L_A(1)$ and then "bbc" $\in L_A(1)$ and "bb" $\in L_A(1)$ and so on. We can see that $L(\underline{\text{`b'}^*; \text{`c'}^?}) \subseteq L_A(1)$. Now by the second rule we have $L(\underline{\text{`a'}; \text{`b'}^*; \text{`c'}^?}) \subseteq L_A(0)$. There are no other facts we can derive from the two rules and so, by the meta-rule, we have $L_A(2) = \{\epsilon\}$, $L_A(1) = L(\underline{\text{`b'}^* \text{`c'}^?})$, and $L_A(0) = L(\underline{\text{`a'}; \text{`b'}^*; \text{`c'}^?})$.

There is another, equivalent, way to define the language associated with each state. We can define that $w \in L_A(q)$ iff there is a path from q to some state in F such that w is the catenation of labels along the path. Formally $w \in L(q)$ exactly if, for some $n \in \mathbb{N}$, there are

- n strings $w_0, w_1, \ldots, w_{n-1}$, such that $w_0{}^\frown w_1{}^\frown \ldots {}^\frown w_{n-1} = w$, and
- $n + 1$ states $q_0, q_1, ..., q_n$, such that $q_0 = q$, $q_n \in F$, and such that, for each $i \in \{0, ..n\}$, $(q_i, w_i, q_{i+1}) \in T$.

Example 75 We can see that "abb" $\in L_A(0)$ for the NDFR of Example 73 by observing the path

$$(0, \text{"a"}, 1)\ (1, \text{"b"}, 1)\ (1, \text{"b"}, 1)\ (1, \epsilon, 2)$$

The language defined by the automaton is the language of its start state: $L(A) = L_A(q_{\text{start}})$.

8.0.2 Systematic state renaming

The states don't really matter.

We can systematically replace any set of states with any other set of the same size.

Let A be an NDFR $A = (S, Q, q_{\text{start}}, F, T)$, $\dot{Q}$ be any set such that $\left|\dot{Q}\right| \geq |Q|$, and f be a one-one total function from Q to $\dot{Q}$. Then $L(A) = L(\dot{A})$, where $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$ is derived from A as follows:

- $\dot{q}_0 = f(q_{\text{start}})$
- $\dot{F} = \{q \in F \cdot f(q)\}$
- $\dot{T} = \{(q, a, r) \in T \cdot (f(q), a, f(r))\}$

8.1 All regular languages are described by NDFRs

8.1.0 Thompson's construction

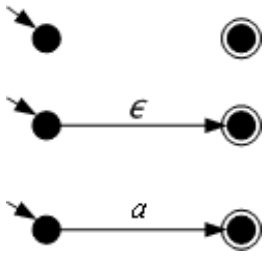
Next we will see that every language described by a regular expression can also be described by an NDFR. To do this we will show how an arbitrary regular expression x can be translated into an NDFR $A(x)$ such that $L(x) = L(A(x))$. This translation is called *Thompson's construction.*[1] Each NDFR constructed by Thompson's construction will have a start state, that has no incoming transitions, and one accepting state, that has no outgoing transitions.

Thompson's construction

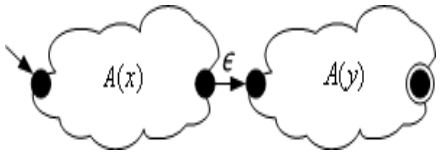
We only need to consider regular expressions made without any of the abbreviations introduced in the last chapter. If a regular expression has abbreviations, we can first rewrite it as an equivalent regular expression without abbreviations.

- For regular expressions $\underline{\emptyset}$, $\underline{\epsilon}$, and $\underline{a}$, we can construct automata:

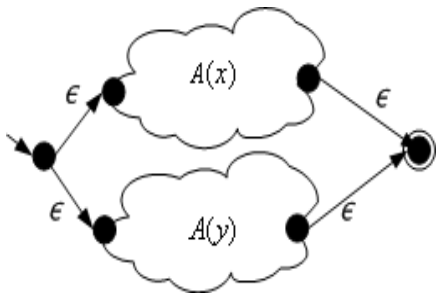
[1] Thompson's construction is named for Ken Thompson, who is perhaps better known as the co-creator of the Unix operating system, the Plan-9 operating system, and the Go programming language.



- For regular expression $x;y$, we first construct $A(x)$ and $A(y)$, using Thompson's construction. If necessary, the states are renamed so that the two automata have disjoint state sets. Finally, we combine them as follows to get $A(x;y)$.

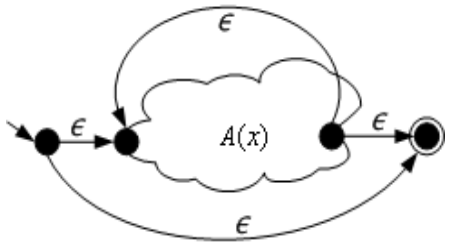


- For regular expression $(x|y)$, we first construct $A(x)$ and $A(y)$, using Thompson's construction. If necessary, the states are renamed so that the two automata have disjoint state sets. Finally, we combine them and add two new states and four new transitions, as follows to get $A((x|y))$.



- For regular expression (x^*), we first construct $A(x)$, using Thompson's construction. Finally, we add two new states and two new transitions

as follows to get $A\left(\underline{(}x^*\underline{)}\right)$.

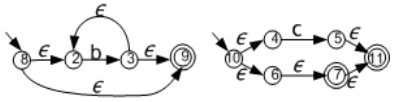


The existence of Thompson's algorithm, shows that every regular language (i.e. every language described by a regular expression) is also described by an NDFR. This shows that the class of all regular languages is a subset of the class of all NDFR languages.

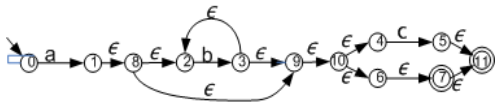
8.1.1 Example of Thompson's construction

Let's consider the regular expression "a"; "b"*; ("c" | ϵ). We start by building NDFRs for all the primitive regular expressions. To simplify things later, I've made sure the state sets are disjoint.

Next we create an NDFR for "b"* and another for ("c" | ϵ)



Finally, we can deal with the catenations by catenating the automata for "a", "b"* , and ("c" | ϵ); we get:



Algorithm 1 The one-finger NDFR recognition algorithm

0: Start by putting your finger on the start state. Whatever state your finger is on is called "the current state".
1: If there are no ϵ-transitions out of the current state, skip to step 4. Otherwise either go to step 4 or to step 2.
2: Pick one ϵ-transition out of the current state and move your finger to the other end of that transition.
3: Go back to step 1.
4: If there are no more input symbols and your finger is on a state in F, you have succeeded. Stop.
5: If there are no more input symbols and your finger is not on an accepting state, you have failed. Stop.
6: If there is no transition out of the current state labelled by the next input symbol, you have failed. Stop.
7: If you get to this step, there must be at least one transition out of the current state that is labelled with the next input symbol. Pick one such transition. Move your finger to the other end of the transition. Cross-off the input symbol.
8: Go back to step 1.

8.2 Recognition algorithms

Recall that it did not appear easy to directly create an algorithm to determine whether a string is or is not in the language described by a regular expression. However, now that we can translate regular expressions to NDFRs, we have a new approach. Given a regular expression, we can translate it to an equivalent NDFR and then determine whether the string is in the language described by the NDFR. So we need an algorithm to determine whether a string is in such a language. The idea is to find a path from the start state to an accepting state with a sequence of labels that, catenated, make the string.

8.2.0 A one-finger algorithm

We could imagine recognizing a string, written on a piece of paper, using a diagram of an NDFR, as described in Algorithm 1.

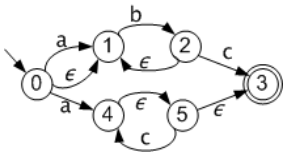


Figure 8.0: An NDFR.

Example 76 Consider the NDFR in Figure 8.0. Suppose we want to recognize a string "abbc". We start with our finger on the start state, state 0. We can run the algorithm as follows successfully recognize the string as follows:

- Move finger to state 1 and cross off the first item of input. Remaining input is now "bbc".
- Move finger to state 2 and cross off the first remaining item of input. Remaining input is now "bc".
- Move finger to state 1, using the ϵ transition.
- Move finger to state 2 and cross off the first remaining item of input. Remaining input is now "c".
- Move finger to state 3 and cross off the final item of input. Remaining input is now ϵ.

At this point we the algorithm finishes successfully.

However, if we take different choices we may finish unsuccessfully. As before we start with our finger on state 0 and remaining input of "abbc"

- Move finger from state 0 to state 4, crossing of the "a". Remaining input is now "bbc".

At this point the algorithm can stop or move on to state 5 and possibly state 3, but there is no way that more symbols can be crossed off the input and so the algorithm can only stop with failure.

The problem with this method is that there are a number of places where you have to make choices with no guidance.

- In step 1, if there are any epsilon transitions out of the current state, you must choose whether to go on to step 4 or to follow an ϵ-transition.
- In step 2, if there is more than one ϵ-transition out of the current state, you must pick which one to follow.
- In step 7, if there is more than one transition out of the current state labelled with the next symbol, you must choose which one to follow.

This is why we say the algorithm is nondeterministic.

If you are sufficiently lucky or prescient, you can always use this algorithm to show that a string is in the language described by the NDFR. However, if luck is not with you, you may find the algorithm fails even if it could have succeeded. Furthermore, the algorithm could loop forever by following a cycle of epsilon transitions in steps 1, 2, and 3.

The best we can say for the algorithm is that if it succeeds then the string is in the language.

Imagine though that you have a magical lucky coin. Whenever you have to make a choice, you flip the magic coin. The magic coin has the following property. Whenever tossed, it comes down heads or tails so as to allow you to succeed. If the answer doesn't matter, the result of the coin is arbitrary. But if the result does matter, that is, one choice leads to possible success and the other to certain failure, the coin comes up heads or tails in a way that keeps the possibility of success alive. The one-finger algorithm used in conjunction with the magical lucky coin will succeed, if the string is in the language, and will either fail or loop forever if the string is not.

8.2.1 Relationship to nondeterminism in programming

This section is optional reading.

In Part 0, we looked at nondeterministic specification.

The one finger algorithm is equivalent to a specification of

$$h = \left\langle a' \Rightarrow w \in L(A) \right\rangle$$

where a indicates success. This is a nondeterministic specification. If a is true at the end, then the string w is in the language of the automaton.

Consider a programming construct

try f **else** g

with the following meaning

$$(\textbf{try } f \textbf{ else } g)\,(i\dagger o) = \textbf{if } (\exists o \cdot f(i\dagger o)) \textbf{ then } f(i\dagger o) \textbf{ else } g(i\dagger o)$$

In other words the output of **try** f **else** g is determined by f if possible. However, if f can not deliver any result, then the result is determined by g. Note that **try** f **else** g is the same as f if f is implementable. It is only an interesting construct if f is not implementable. If g is implementable, then so is **try** f **else** g, regardless of whether f is implementable. We can use **try else** as a mechanism to turn an unimplentable task f into an implementable one **try** f **else** g; if using algorithm f turns out to be infeasible, then algorithm g is used as a back-up plan. This is a bit like exception handling, but it is more like backtracking in languages such as Prolog and Icon.

In contrast to most programming constructs, **try** f **else** g is not monotonic in its first argument.

Another useful construct to use in combination with **try else** is

$$\textbf{force } \mathcal{A}$$

where $\mathcal{A}$ is a boolean expression in the initial state. We define **force** by

$$\textbf{force } \mathcal{A} = \langle A \rangle \wedge \textbf{skip}$$

The **force** command ensures that the expression is true, but it does so in an unimplementable way: it forces the expression to already be true in the input state.

Now we can combine the one-finger algorithm with forcing the result we want

$$h; \textbf{force } a$$

where, as before $h = \left\langle a' \Rightarrow w \in L(A) \right\rangle$. This forces the one-finger algorithm to find a way to succeed if possible

$$(h; \textbf{force } a) = \langle a \wedge s \in L(A) \rangle$$

Of course, this is unimplementable. We can make an implementable specification by

$$\textbf{try } (h; \textbf{force } a) \textbf{ else } a := \mathfrak{false}$$

which refines the specification that we really want

$$\langle a' = (s \in L(A)) \rangle$$

Algorithm 2 The NDFR recognition algorithm

var $r := q_{\text{start}}$.
var $b := \mathfrak{true}$.
while $w \neq \epsilon \wedge b$ **do** (
$\quad$ $(\textbf{var}\ q \mid (r, \epsilon, q) \in T \cdot r := q)$
$\vee$
$\quad$ $(\textbf{var}\ q \mid (r, [w(0)], q) \in T \cdot r := q; w := \text{tail}(w))$
$\vee$
$\quad$ $(b := \mathfrak{false}))$
$a := w \neq \epsilon \wedge r \in F$

Aside: In terms of implementation, we can actually implement an algorithm of the form

$$\textbf{try}\ \ (f; \textbf{force}\ \mathcal{A})\ \ \textbf{else}\ g$$

where f is an a nondeterministic algorithm written in terms of ordinary programming constructs and nondeterministic choices of the form $m0 \vee m1 \vee m2 \vee \cdots$. Every time we come to a nondeterministic choice within f, we record the values of all variables and where we are; then we take one branch of the choice, remembering which. If we get to the force, if $\mathcal{A}$ is true, we are done. However if we get to the force and find that $\mathcal{A}$ is false, we go back to an earlier choice and take the another branch. (This is called backtracking.) If we exhaust all choices, then we must conclude that f can never make $\mathcal{A}$ true, and we then backtrack all the way to the initial program state and execute g. The choice operator $\vee$ provides the coin. The **force** and try constructs provide the magic. End of Aside.

Now what remains is to implement $h = \langle a' \Rightarrow w \in L(A) \rangle$.

Let $A = (S, Q, q_{\text{start}}, F, T)$

The var command $(\textbf{var}\ q \mid E \cdot f)$ introduces a new variable q, chooses an arbitrary value for it satisfying the expression E, and then executes the command f. The tail function computes a sequence that is just like its argument, but without the first item.

This algorithm nondeterministicly sets a to true or false, but it can only set a to true when $w \in L(A)$.

8.2.2 A many-finger algorithm

Next we look at an algorithm that does not rely on luck, magic coins, or backtracking. You can think of it as being similar to the one-finger algorithm above, except that we use many fingers. We place one finger on each state that could be the current state of the one-finger algorithm.

Before getting to the algorithm, it is useful to make a few definitions. Let $A = (S, Q, q_{\text{start}}, F, T)$ be an NDFR.

- For any $r \in Q$, define ϵ-closure(r), to be the set of states reachable from r using zero or more transitions labelled with ϵ. That is

$$\begin{aligned} &r \in \epsilon\text{-closure}(r) \text{ and} \\ &\text{if } p \in \epsilon\text{-closure}(r) \text{ and } (p, \epsilon, q) \in T \text{ then } q \in \epsilon\text{-closure}(r) \end{aligned}$$

- For any $R \subseteq Q$, define ϵ-closure(R) to be the set of states reachable from any $r \in R$ using only zero or more transitions labelled with ϵ.

$$\epsilon\text{-closure}(R) = \bigcup_{r \in R} \epsilon\text{-closure}(r)$$

  Note that $R \subseteq \epsilon$-closure(R).

- For any $r \in Q$ and $a \in S$, define $\delta(r, a)$ to be the set of states reachable from r by a single transition labelled a.

$$\delta(r, a) = \{q \in Q \mid (r, \text{“}a\text{”}, q) \in T\}$$

- For any $R \subseteq Q$ and $a \in S$, define $\delta(R, a)$ to be the set of states reachable from any $r \in R$ using a single transition labelled a.

$$\delta(R, a) = \bigcup_{r \in R} \delta(r, a)$$

Suppose we have a string “abc”. If you follow the one-finger algorithm, what states could be the current state after reading this string? I.e., what states could your finger be on?

- You start with q_{start} as the current state but, even before reading the ‘a’, you can follow ϵ transitions, and so, after reading zero symbols, the current state can change to any of the states in ϵ-closure(q_{start}). Let R_0 be this set.

- After reading the 'a', the current state can become any state connected to an ϵ-closure(q_{start}) by a transition labelled a, thus

  $$\delta(R_0, \text{'a'})$$

  but you can then follow ϵ transitions to reach other states. Thus after reading the 'a' the current state could be any state in

  $$R_1 = \epsilon\text{-closure}(\delta(R_0, \text{'a'}))$$

- Similarly after reading "ab", the current state can be any of the states

  $$R_2 = \epsilon\text{-closure}(\delta(R_1, \text{'b'}))$$

- Finally after reading "abc", the current state can be any of the states in

  $$R_3 = \epsilon\text{-closure}(\delta(R_2, \text{'c'}))$$

  Thus "abc" is in the language iff any of these states are accepting, i.e., if

  $$R_3 \cap F \neq \emptyset$$

This is how the recognition algorithm works.
We can construct an algorithm for recognizing a string w_0 with an NDFR

$$\langle\ f' = (w_0 \in L(A))\rangle$$

We use variables $w \in S^*$ and $R \subseteq Q$. The invariant is that

$$(w_0 \in L_A(q_{\text{start}}))\ \text{iff}\ (\exists r \in R \cdot w \in L_A(r))$$

and that R is closed under ϵ transitions: $R = \epsilon$-closure(R). The resulting algorithm is Algorithm 3.

Example 77 Let's consider the behaviour of the algorithm on the NDFR in Figure 8.0 and with input string "abbc". We start with the start state 0, but even before reading the first input there could be a transition to state 1; the initial value of R is thus $\{0, 1\}$.

Algorithm 3 The NDFR recognition algorithm

var $w := w_0 \cdot$
var $R := \epsilon\text{-closure}(q_{\text{start}}) \cdot$
// inv: $(w_0 \in L_A(q_{\text{start}})) \;=\; (\exists r \in R \cdot w \in L_A(r))$
// inv: $R = \epsilon\text{-closure}(R)$
while $w \neq \epsilon \;\wedge\; R \neq \emptyset$ **do** (
  let $a, s \mid w = ([a];s) \cdot$
  $R := \epsilon\text{-closure}(\delta(R, a))$;
  $w := s$) ;
$f := (R \cap F \neq \emptyset)$

- Now we consider the 'a'. State 0 has and "a" transition to state 1 ($\delta\,(0, \text{`a'}) = \{1, 4\}$). State 1 has no "a" transitions ( $\delta\,(1, \text{`a'}) = \emptyset$). After the first iteration, we have $\delta\,(R, \text{`a'}) = \{1, 4\}$. ϵ closure adds states 5 and 3. So after the first iteration, we have $R = \{1, 3, 4, 5\}$.

- Next we consider the first b. Considering b transitions, we can get to state 2 from state 1; from states 3, 4, and 5, we get nowhere. We have $\delta\,(R, \text{`b'}) = \{2\}$ and then considering ϵ transitions we can also get to state 1. After the second iteration we have $R = \{1, 2\}$.

- Next we consider the second b. Considering b transitions, we can get to state 2 from state 1; from state 2 we get nowhere. Again we have $\delta\,(R, \text{`b'}) = \{1\}$ and so at the end of the third iteration we have $R = \{1, 2\}$.

- Next consider the c. Considering c transitions, we can get to state 3 from state 2. From state 1 we get nowhere. We have $\delta\,(R, \text{`c'}) = \{3\}$. ϵ closure adds no more states, so we end the iteration with $R = \{3\}$.

At this point the loop is done and there is an accepting state in R and so the algorithm ends successfully.

This gives us a way of recognizing regular expressions: Convert the regular expression to an NDFR using Thompson's construction (or some other method) and then run Algorithm 3.

8.2.2.0 Speed

Although the number of iterations equals the length of the string. Each iteration can take time proportional to the size of R with can be roughly as big as Q. So overall the algorithm takes no more than time proportional to $|Q| \times \|w_0\|$.

8.3 Deterministic Finite State Recognizers (DFRs)

Deterministic Finite State Recognizer

DFR

Definition 78 A *Deterministic Finite State Recognizer (DFR)* is an NDFR such that

- There are no transitions labeled by ϵ

  Thus ϵ-closure$(q) = \{q\}$ for all $q \in Q$.

- For each state-symbol pair (q, a), there is exactly one r such that $(q, \text{“}a\text{”}, r) \in T$

  I.e. $|\delta(q, a)| = 1$, for all $q \in Q$ and $a \in S$.

Thus for a DFR, the size of R in the recognition algorithm always 1.
If we have a DFR, we can represent variable $R \subseteq Q$ by variable $r \in Q$:

$$R = \{r\}$$

This is a data refinement.

The NDFR recognition algorithm specialized to a DFR is given as Algorithm 4.

By using numbers for states and symbols, we can represent δ as a 2-dimensional array. Then this algorithm is linear time in the length of w_0.

[[Example needed]]

This gives us another approach to recognizing regular expressions, which is potentially very efficient.

- Convert the RE to an NDFR using Thompson's construction, or some other construction that accomplishes the same thing.

- Convert the NDFR to an equivalent DFR.

Algorithm 4 The DFR recognition algorithm

var $w := w_0$
var $r := q_{\text{start}}$
// inv: $w_0 \in L(q_{\text{start}}) \Leftrightarrow w \in L(r)$
while $w \neq \epsilon$ **do** (
  let $a, s \mid w = ([a];s) \cdot$
  $r :=$ the sole element of $\delta(r, a)$;
  $w := s$)
$f := r \in F$

Algorithm 5 The naive subset construction algorithm

Input: An NDFR $A = (S, Q, q_{\text{start}}, F, T)$
Output: A DFR $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$ that recognizes the same language: $L(A) = L(\dot{A})$

$\dot{Q} := \mathcal{P}(Q)$, the power set of Q.
$\dot{q}_{\text{start}} := \epsilon\text{-closure}(q_{\text{start}})$
$\dot{F} := \{R \subseteq Q \mid F \cap R \neq \emptyset\}$
$\dot{T} := \{a \in S, R \subseteq Q \cdot (R, a, \epsilon\text{-closure}(\delta(R, a)))\}$

- Match using the DFR recognition algorithm (Algorithm 4).

Next, we look at how to accomplish the second step.

8.4 From NDFRs to DFRs

If you try to design a DFR for a complex language, you may find it is far more difficult than designing an NDFR for the same language. It seems that NDFRs may be a more powerful tool for describing languages than are DFRs. However, in at least one sense, this is not true. It turns out that, for any NDFR A, there is a DFR $\dot{A}$ such that $L(A) = L(\dot{A})$.

Algorithm 5 shows one way to do it. The states of $\dot{A}$ are the subsets of the states of A.

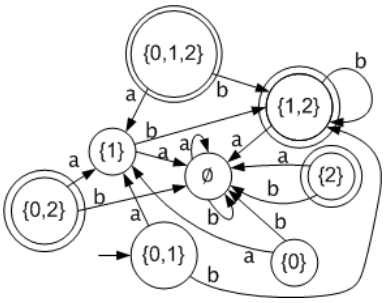
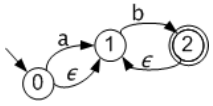


Figure 8.1: The result of a naive subset construction.

Example 79 Consider the following automaton



There are 3 states and so the DFR has $2^3 = 8$ states: $\{\emptyset, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$. The full automaton is shown in Figure 8.1.

The problem with this algorithm is that it always generates a large number of states: $\left|\dot{Q}\right| = 2^{|Q|}$. Many of these states will not be reachable from the start state, and hence contribute nothing. We can obtain some reduction, by including only ϵ-closed sets as states. However this algorithm still potentially generates many states that are not reachable by from the initial state.

The states that are reachable from the start state are exactly the values that the R variable can take on in the NDFR recognition algorithm (Algorithm 3).

A better algorithm is shown as Algorithm 6. It only generates DFR states that are reachable from the start DFR state. Essentially what we do is to compute the set of all sets R that might arise in the recognition algorithm.

- W is a set of DFR states that have been discovered, but not explored.
- $\dot{Q}$ is the set of states that have been discovered and explored.

When we explore a state, we find all its out-going transitions and all the states they go to.

Algorithm 6 The subset construction algorithm

Input: An NDFR $A = (S, Q, q_{\text{start}}, F, T)$
Output: A DFR $\dot{A} = (S, \dot{Q}, \dot{q}_{\text{start}}, \dot{F}, \dot{T})$ that recognizes the same language: $L(A) = L(\dot{A})$

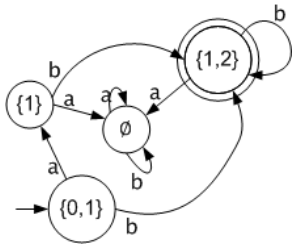
$\dot{q}_{\text{start}} := \epsilon\text{-closure}(q_{\text{start}})$;
var $W := \{\dot{q}_{\text{start}}\}\cdot$
$\dot{Q} := \emptyset$;
$\dot{T} := \emptyset$;
$\dot{F} := \emptyset$;
while $W \neq \emptyset$ **do** (
  // Pick any state that has been discovered, but not explored
  let $\dot{q} \mid \dot{q} \in W\cdot$
  // Explore state $\dot{q}$
  $W := W - \{\dot{q}\}$;
  $\dot{Q} := \dot{Q} \cup \{\dot{q}\}$;
  if $\dot{q} \cap F \neq \emptyset$ **then** $F := F \cup \{\dot{q}\}$ **else skip** ;
  for each $a \in S$ **do** (
    let $\dot{r} = \epsilon\text{-closure}(\delta(\dot{q}, a))\cdot$
    $\dot{T} := \dot{T} \cup \{(\dot{q}, \text{“}a\text{”}, \dot{r})\}$;
    // If $\dot{r}$ has not been discovered already, add it to W.
    if $\dot{r} \notin \dot{Q}$ **then** $W := W \cup \{\dot{r}\}$ **else skip**))

After running the subset construction algorithm, we can systematically rename the states in $\dot{Q}$ with natural numbers so that $\dot{T}$ can be efficiently represented with an array.

Example 80 Looking at the NDFR from Example 79, and applying the subset construction algorithm (Algorithm 6), we start with the state $\dot{q}_{\text{start}} = \epsilon\text{-closure}(0) = \{0, 1\}$. The algorithm proceeds as shown in Table 8.0. The resulting DFR (before renaming) is

W	$\dot{q}$	a	$\dot{r}$	new transition	$\dot{Q}$
$\{\{0,1\}\}$	$\{0,1\}$				$\emptyset$
		'a'	$\{1\}$	$(\{0,1\}, \text{'a'}, \{1\})$	
		'b'	$\{1,2\}$	$(\{0,1\}, \text{'b'}, \{1,2\})$	
$\{\{1\},\{1,2\}\}$	$\{1\}$				$\{\{0,1\}\}$
		'a'	$\emptyset$	$(\{1\}, \text{'a'}, \emptyset)$	
		'b'	$\{1,2\}$	$(\{1\}, \text{'b'}, \{1,2\})$	
$\{\{1,2\},\emptyset\}$	$\{1,2\}$				$\{\{0,1\},\{1\}\}$
		'a'	$\emptyset$	$(\{1,2\}, \text{'a'}, \emptyset)$	
		'b'	$\{1,2\}$	$(\{1,2\}, \text{'b'}, \{1,2\})$	
$\{\emptyset\}$	$\emptyset$				$\{\{0,1\},\{1\},\{1,2\}\}$
		'a'	$\emptyset$	$(\emptyset, \text{'a'}, \emptyset)$	
		'b'	$\emptyset$	$(\emptyset, \text{'b'}, \emptyset)$	
$\emptyset$					$\{\{0,1\},\{1\},\{1,2\},\emptyset\}$

Table 8.0: Table Caption



Note that this automaton includes only ϵ-closed (with respect to the NDFR) sets as states, but it does not include all ϵ-closed sets.

8.4.0 Minimal DFRs

[[Section to be completed]]

8.4.1 Recognizing regular expressions with DFRs

Now we can 'efficiently' recognize regular expressions

- Translate the RE to an NDFR

- Translate the NDFR to a DFR
- Replace subsets with numbers
- Optionally minimize the number of states in the DFR.
- Execute the specialized recognition algorithm

Note that the NDFR will be about the same size as the regular expression, but the number of states in the DFR can be *exponential* in the number of states of the NDFR.

An alternative approach, which may be more space efficient, is

- Translate the RE to an NDFR
- Execute the recognition algorithm

The first approach may be best if the regular expression is not too large and you intend to execute the recognition algorithm many times for the same regular expression, or on a large complex text. The Unix program grep uses this approach, as does the lexical analyzer generator lex.

The second approach may be best if the regular expression is so large that exponential blow-up is worrying or if speed of constructing the machine is more important than speed of executing it. The Unix program fgrep uses the second approach, as does the lexical analyzer generator in JavaCC.

8.5 Equivalence of regular expressions and (N)DFRs

We have already defined that

- A **regular language** is a language described by some regular expression

Let's make the following (temporary) definitions.

- An **NDFR language** is a language described by some NDFR
- A **DFR language** is a language described by some DFR

We have shown that any regular expression can be translated to an equivalent NDFR and any NDFR to an equivalent DFR and so we know

$$\text{the regular languages } \subseteq \text{ the NDFR languages} \subseteq \text{ the DFR languages}$$

Furthermore any DFR is an NDFR and so every DFR language must also be an NDFR language:

$$\text{the regular languages } \subseteq \text{ the NDFR languages} = \text{ the DFR languages}$$

In the next section, we will see that any NDFR can be translated into a regular expression and so

$$\text{the regular languages } = \text{ the NDFR languages} = \text{ the DFR languages}$$

Once we have done that, we no longer need the terms "NDFR language" and "DFR language," we just use the term "regular language."

8.5.0 From NDFRs to Regular Expressions

Definition 81 A *Regular Expression Finite Recognizer (REFR)* is just like an NDFR, except that the transitions are labeled with regular expressions over S.

The language described by each state q of an REFR $A = (S, Q, q_{\text{start}}, F, T)$ is defined by two rules

- If $q \in F$ then $\epsilon \in L_A(q)$
- If $(q, x, r) \in T$ then $L(x)\hat{}L_A(r) \subseteq L_A(q)$

Meta-rule: A string w is in $L(q)$ only if it can be proved so by finite application of the above 2 rules.

Equivalently, we can define that $w \in L_A(q)$ iff there is a finite path from q to some state in F such that w is in the language of the catenation of labels along the path.

I.e., $w \in L_A(q)$ iff, for some $n \geq 0$, there are

- n strings $w_0, w_1, \ldots, w_{n-1}$, such that $w_0\hat{}w_1\hat{}\ldots\hat{}w_n = w$,

- $n+1$ states $q_0, q_1, ..., q_n$, such that, $q_0 = q$ and $q_n \in F$, and

- n regular expressions $x_0, x_1, ..., x_{n-1}$,such that, for each $i \in \{0, ..n\}$, $(q_i, x_i, q_{i+1}) \in T$ and $w_i \in L(x_i)$.

The language defined by the automaton is the language of its start state: $L(A) = L_A(q_{\text{start}})$.

Any NDFR can be trivially translated to an REFR by replacing each label with the corresponding regular expression.

Any regular expression x over S can be trivially represented by an REFR

$$(S, \{0, 1\}, 0, \{1\}, \{(0, x, 1)\})$$

We will look at an algorithm for translating an arbitrary NDFR into a regular expression.

The state removal algorithm

input: An NDFR $A_0 = (S_0, Q_0, q_{\text{start}_0}, F_0, T_0)$

output: An RE x

specification: $\langle L(x') = L(A_0) \rangle$

We start by making an REFR copy, $A = (S, Q, q_{\text{start}}, F, T)$, of A_0

$$A := \text{convertNDFRtoREFR}\,(A_0)$$

As the algorithm modifies A, we maintain, as an invariant, $L(A) = L(A_0)$.

The rest of the algorithm consists of four steps that can calculate a regular expression from any REFR.

- Step 0. Ensure that there is one accepting state and that there are no transitions into the initial state or out of the accepting state.

- Step 1. Make sure each pair of nodes has no more than one transition between them.

- Step 2. Eliminate all nodes that are not initial or accepting.

- Step 3. Output the remaining regular expression.

8.5.0.0 Step 0

In this step we ensure that the REFR has

- no edges into its initial state q_{start},
- exactly one accepting state $F = \{q_{\text{final}}\}$ with $q_{\text{final}} \neq q_{\text{start}}$
- no edges out of its accepting state.

This step is easily accomplished by adding a new initial state, a new accepting state, and ϵ-labeled transitions as needed.

8.5.0.1 Step 1

In this step we ensure that each pair of states has at most one transition between them

for all pairs of states q and r in Q
    coalesceTransitions(q, r)

where coalesceTransitions is defined by

procedure coalesceTransitions(q, r) **is**
    if there is more than one transition from q to r **then**
        let $x_0, x_1, ...x_n$ be the labels on those transitions
        remove all transitions from q to r from T
        add $(q, \underline{(}x_0\underline{)} \mid \underline{(}x_1\underline{)} \mid ... \mid \underline{(}x_n\underline{)}, r)$ to T

8.5.0.2 Step 2

In this step we reduce the number of states in the automaton to 2.
We eliminate states one at a time.

while there are more than two states **do**
    let q be any state that is not initial nor accepting ·
    eliminate(q)

The loop invariant for this repetition is that the REFR has

- a single accepting state, q_{final}, with $q_{\text{final}} \neq q_{\text{start}}$
- no transitions to its initial state nor from its accepting state
- at most one transition between any two states, and
- $L(A) = L(A_0)$

The procedure for eliminating state q is as follows:

procedure eliminate(q) **is**
  for all p and r in Q such that $p \neq q$ and $r \neq q$ and
  there are transitions from p to q and from q to r·
    let x be such that $(p, x, q) \in T$·
    let z be such that $(q, z, r) \in T$·
    if there is a transition from q to q **then** (
      let y be such that $(q, y, q) \in T$·
      add $(p, \underline{(}x\underline{)}\underline{;}\underline{(}y\underline{)}^*\underline{;}\underline{(}z\underline{)}, r)$ to T)
    else
      add $(p, \underline{(}x\underline{)}\underline{;}\underline{(}z\underline{)}, r)$ to T ;
    coalesceTransitions(p, r) ;
  remove from T all transitions either to or from q ;
  remove q from Q

8.5.0.3 Step 3

At this point we have two states, q_{start} and q_{final}, and at most one transition between them

- If there is a transition $(q_{\text{start}}, x, q_{\text{final}})$ output x
- Otherwise output $\underline{\emptyset}$.

8.5.1 Example

NOTE TO SELF A simpler example is to find an RE for the set of all strings in $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^*$ in which a c never immediately follows an a. The NDFA needs only two states.

Comments in C follow a syntax as follows

- They start with a "/*".
- They end with a "*/".
- Between the initial "/*" and the final "*/", there can be any sequence of characters that does not include a "*" immediately followed by a "/".

Examples are "/**/" (the shortest comment), "/*abc/*def*/, and "/*****/".

We will use the algorithm just presented to compute a regular expression for the language of C comments.

For an alphabet, we will use $\{\mathsf{a}, \mathsf{s}, \mathsf{x}\}$ representing respectively asterisks, slashes, and all other characters.

[[Example to be completed.]]

8.5.2 Summary

We've now seen that any DFR or NDFR can be converted to an equivalent regular expression by converting it first to a REFR and then reducing that to a regular expression.

In summary: We can translate any regular expression to an equivalent NDFR (Thompson's construction). We can translate any NDFR to an equivalent DFR (subset construction); and we can translate any DFR to a regular expression (state removal).

This completes the proof that

$$\text{the regular languages} \;=\; \text{the NDFR languages} = \;\text{the DFR languages}$$

8.6 Are all language regular?

At this point, we have three ways to show that a language is regular: we can find a regular expression for it; we can find a DFR for it; we can find an NDFR for it.

To show that a language is not regular, we only need to show that one of the above methods is doomed to failure.

We will answer the question at the head of this section by presenting a language and arguing that there can be no DFR for it. The language is

$$\{\epsilon, \text{“}\mathsf{01}\text{”}, \text{“}\mathsf{0011}\text{”}, \text{“}\mathsf{000111}\text{”}, \cdots\}$$

Suppose A is a DFA for this language. Further, suppose that after reading m '0's the state is q, and after reading n '0's the state is r. If $m \neq n$, then q and r must be different because, if we start the machine in state q and read m '1's, the resulting state must be accepting, whereas, if we start the machine in state r and read m '1's, the resulting state must be non-accepting. The machine must be in a distinct state, for each number of '0s' read. But this means there can not be a finite number of states. A DFR for this language can not exists. Thus the language is not regular.

We can think of the state of a DFR as summarizing the past input. Suppose after reading string s, a DFR A is in state q, then $s\hat{}t \in L(A)$ exactly if $t \in L_A(q)$, so all that the machine 'remembers' about s is summarized in state q. If there are only a finite set of categories of input that need to be remembered, then a DFR can be used and hence the language is regular. If there are an infinite number of categories of input that need to be remember, then there can be no DFR and so the language is not regular.

For example, consider a language that consists of even length strings of '0's and '1's, such that there are never two '0's in a row. We want to know if $s\hat{}t$ is in this language, but we've forgotton what s is. If we can remember something about s we may still be able to determine the answer using only that something and t. In particular, if we can remember whether the length of s was odd or even and whether it ended in a '0' or a '1', then we can still determine whether $s\hat{}t$ is in the language, knowing t and nothing else about s. There are only 4 possibilities (even-0, even-1, odd-0, odd-1), so it should be possible to construct a DFR (with four states) for the language.

Here is another example: Consider the language of strings representing sums in decimal notation in row-major form. E.g. a sum

$$\begin{array}{r} 0456 \\ \underline{0789} \\ 1245 \end{array}$$

is represented by "001472584695". We need to remember whether the number of preceeding characters divided by 3 leaves a remainder of 0, 1, or 2. We need to remember whether a carry is expected from the next place. (E.g., after reading "001", we would expect the next two digits add to more than 9). And may we need to remember the last digit or two, or at least their sum.[2] Finally, we need to remember whether any errors have already been found.

[2] Specifically: when the the length of the input read so far divided by 3 is 0, we don't

If all that is known, we can forget about every other detail of the input read so far. Thus the language is regular. Now let's consider a language formed from sums in column-major form —e.g., "045607891245" is in this langauge— we will have to remember all of the first number before reading the second number, so, unless the size of the input is limited, there can be no DFR for this language and thus it is not regular.

8.7 Regular expressions in practice

[[Section to be done. Discussion of grep, lex, etc.]]

8.8 Chapter summary

- Regular expressions, DFRs, NDFRs, and REFRs are formalisms that all can express the same set of languages: the regular languages.
- All have the same limitation: Fixed, finite memory. Thus this theory is very important for hardware designers as well as software designers.
- DFRs are efficient for recognition.
- For a given language, the smallest NDFR or regular expression can be far smaller than the smallest DFR.
- Thus converting an NDFR or RE or REFR to a DFR may entail an 'explosion' in the number of states. The DFR may be unacceptably big.
- Regular expressions, being textual, are easy to integrate into user-dialogs (e.g., search dialogs on websites, in Eclipse, UltraEdit, and vi) programming languages (e.g. Perl, JavaScript, sed, lex, JavaCC), and libraries (e.g., java.util.regexp, regex.h).
- Regular expressions are very convenient for expressing many languages, while NDFRs occasionally give elegant solutions to problems where regular expressions do not. REFRs give the best of both worlds. (For example, C/Java style comments.)

need to remember any previous digits; when that remainder is 1, we must remember the last digit; when that remainder is 2, we must remember the sum of the last two digits.

- Example: JavaCC's lexical analyzer generator uses a variant of REFRs for input and NDFRs for implementation.

Chapter 9

Reactive Systems

9.0 Reactive Systems

A finite state transducer is similar to a finite state recognizer, but may include output as well as input.

9.0.0 Finite State Transducer

Consider a machine $A = (S, O, Q, q_0, T)$ where

- S is an input alphabet
- O is an output alphabet
- Q is a finite set of states
- q_0 is the initial state
- T is a set of transitions from $Q \times (S \cup \{\epsilon\}) \times (O \cup \{\epsilon\}) \times Q$

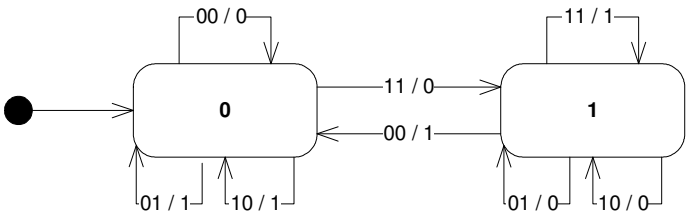
We assume $S \cap O = \emptyset$.
Such a machine defines a set of strings $L(A) \subseteq (S \cup O)^*$:
If there is a path from q_0 to some state in Q and

- s is the catenation of input and output labels on that path,
- then $s \in L(A)$.

9.0.1 Application to digital circuits

If the machine is free of ϵs on both inputs or outputs, then each input is followed by a corresponding output. This gives a good model for synchronous digital systems.

Example: Let $S = \{00, 01, 10, 11\}$ and $O = \{0, 1\}$



Now we have a serial adder. If we feed in two numbers, least-significant bit first, the machine produces the sum. E.g. we have

$$[10/1, 00/0, 11/0, 01/0, 10/0, 00/1, 11/0, 01/0, 10/0, 00/1]$$

corresponding to the sum.

$$\begin{array}{r} 0101010101 \\ 0011001100 \\ \hline 1000100001 \end{array}$$

9.1 System modelling and StateCharts

9.1.0 Reactive systems

Reactive systems are systems that must react to events.

- A Calculator must react to the keypresses
- A Microwave oven must react to keypresses and also to the passage of time.
- An internet router must react to the arrival of packets
- A synchronous hardware circuit must react to the clock ticks.
- Most systems can be viewed as reactive.

We can specify and/or model a reactive system using state machines.

9.1.1 StateCharts

StateCharts is a diagrammatic language for modelling finite state systems.

There are various flavours of StateCharts.

I'll be using UML *StateCharts.*

9.1.2 Transitions

Our terminal alphabet now consist of a set of *events*.

Each transition from state to state is labelled

trigger [condition] / reaction

where

- the *trigger* is an event that triggers the transition
- the *condition* is a boolean expression
- the *reaction* is either
  - an event that is generated,
  - or a change to the system's state variables

9.1.2.0 Example: A Clap-light.

Data Dictionary:

Entity	**Kind**	**Description**
Clap	Event	Occurs when a single clapping sound is detected.
level	System variable	The amount of power sent to the lights.

Here there are 3 states. One event: **Clap** and reactions that change the system's state variable level**.**

If the system is in state **Off** and a **Clap** event occurs, then

- *immediately* and *simultaneously*
- level is set to 60W and the system state changes of **Dim**.

9.1.2.1 Transitions in detail

Transitions are labeled as:

$$(trigger)^? \ ([condition])^? (\ /\ reaction\ )^?$$

- If the *trigger* event is omitted,

  - the transition is taken as soon as the condition is true.

- If the *condition* part is omitted,
  - the condition is *true*
- If the *reaction* is omitted
  - there is no reaction, only a state change.

9.1.3 Conditions

Conditions can be used to inhibit transitions:

Notice the event is parameterized.

Conditions can also be used to choose between transitions:

We could also illustrate this as: (The diamond is not a state.)

9.1.4 Time

It is important to realize:

> *Time passes in the states.*

Thus transitions are essentially instantaneous.

9.1.4.0 What if you want a delay?

For real-time systems the passage of time is an important kind of event.

- after: 1 second is an event that happens 1 second after the source state was entered.
- when(11:59 AM) is an event that takes place at 11:59 AM.

9.1.5 Hierarchy

Time passes within states.

During that time the system need not be idle.

We can divide states into substates

When this system is in state Impulse it will be in exactly one of the substates Half Impulse or Full Impulse.

When it is in state Warp it will be in exactly one of its 3 substates.

At system start the system is in both states Impulse and Half Impulse.

If the system is in state Impulse and a ToWarp event happens, the system will transition to both states Warp and Warp 1.

States like Impulse and Warp are called "or" states, since when the system is in an "or" state, it is also in *exactly one* of the "or" state's substates.

States with no substates are called "basic" states.

The term "state" is being abused here since a system should really be in exactly one state at a time. The set of Statechart states the system is in constitutes its "true" state.

Some people prefer to call "or states" "modes".

9.1.6 Concurrency.

When the system is in an "and" state, it is also in all of the and state's substates.

The substates of an "and" state are always "or" states.

An example (transitions omitted)

Wheels is an "and" state.

Its substates are Left and Right which are both "or" states.

The system could be in states Wheels, Left, Right, LReverse, and RLocked all at the same time.

9.1.7 Communication

You can use events to coordinate actions of concurrent state machines.

If a transition has an event as its reaction,

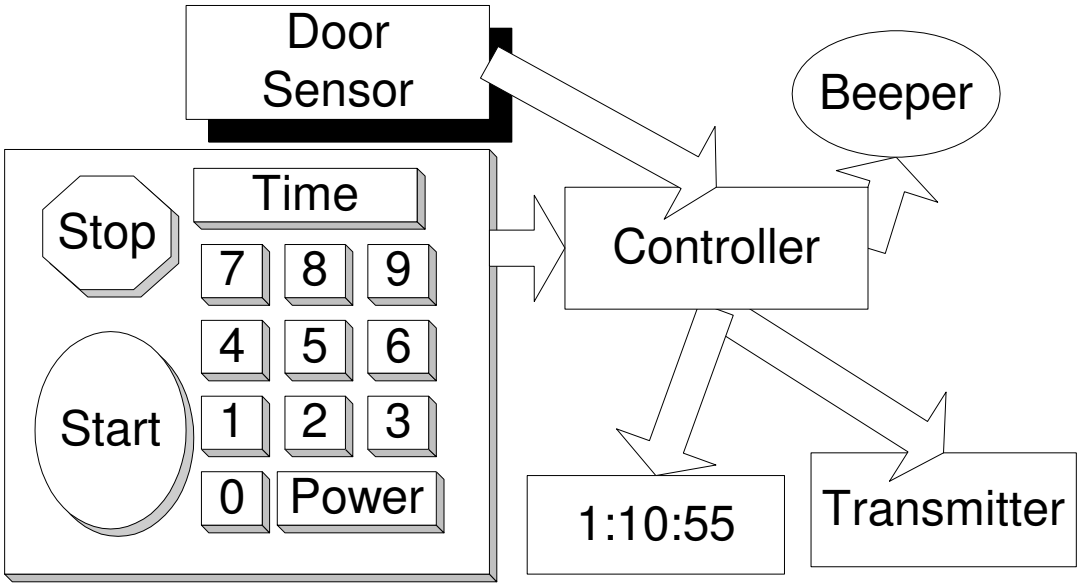
- That event can trigger a transition out of any state the system is in.

Example:

If the system is in both Waiting and Preparing then a codeEntered event causes a disarm event and the new states will include Disarmed and Idle.

9.2 System modelling and StateCharts

9.2.0 A *Microwave oven* Example



9.2.0.0 Inputs

Entity	Type	Description
timeButton	Event	The time button is pressed
powerButton	Event	The power button is pressed
startButton	Event	The start button is pressed
stopButton	Event	The stop button is pressed
digitButton(d:0..9)	Event	A button 0 to 9 is pressed
door	Variable	Can be in states open or closed.

9.2.0.1 Outputs

Entity	Type	Description
transmitter	Variable	The power level of the transmitter {0,1,..,10}
display	Variable	An alpha-numeric display with 6 characters.
beep	Event	Initiate a beeping sound

9.2.0.2 Local entities

Entity	Type	Description
time	Variable	A natural number. In units of seconds
power	Variable	A natural number {1,..,10}

9.2.0.3 Overall behavior

Note

- "entry" actions are performed whenever the state is entered.
- "do" actions are performed continuously when the state is active
- "exit" actions are performed whenever the state is executed.

9.2.0.4 A closer look at the UpdateTime state

We add more detail by adding a transition for digitButton events in the UpdateTime state.

9.2.0.5 The UpdatePower

DigitButton events in the UpdatePower state.

9.2.0.6 The Cook state

Note that ^beep means a beep event occurs as a reaction.

I claim that "the door is open implies transmitter = 0" is a global invariant.

9.3 Using StateCharts to model software classes

Reactive Systems and objects have a lot in common.

- Reactive systems react to events
- Classes react to method invocations.
- Reactive systems have states on which behaviour depends
- Classes have states on which behaviour depends.

Input Events:

- Method calls to this object

Input variables:

- Variables and objects known to the class

Output events

- Calls to objects known to the class

Output variables

- Variables and objects changeable by this object.

9.3.0 Relationship to other UML diagrams:

- Class diagrams describe the "static" relationships between classes.
- Sequence and collaboration diagrams show examples of behaviour.
- StateCharts describe behaviour.

9.3.1 An Example

A savings account.

Account can not be debited once overdrawn:

The behaviour as a state chart:

In C++

```
class Account {
    public: Account() { balance = 0 ; }
    public: withdraw( int amt ) {
        if( balance >= 0 ) balance -= amt ;
        else throw Reject ; }
    public: deposit( int amt ) {
        balance += amt ; }
} ;
```

In this case the state diagram is not simpler than the implementation itself.

This is because the class is very simple.

9.3.2 Another example:

This is class represents a source of network messages.

- It must be initialized before being used and should be shutdown after use.
- It observes a network channel. Thus the network channel is an input variable.
- The client of this class may use it to get lines.

State diagram for Client_network_layer:

Notice that the state can change in response to changes in the observed variable.

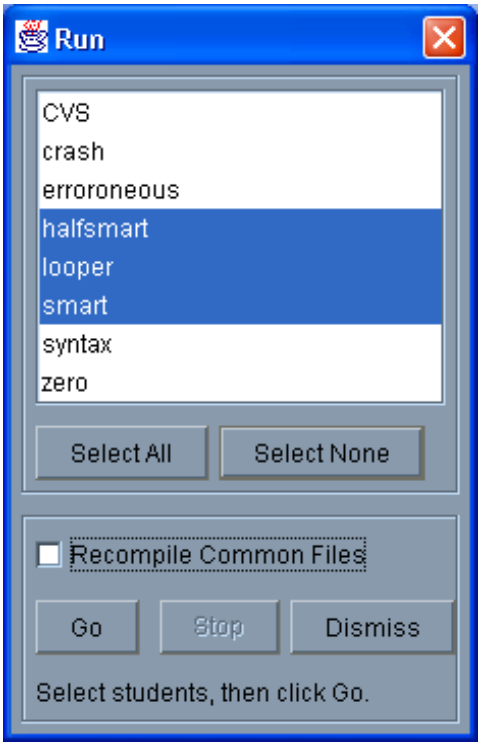
The actual implementation of this class is quite a bit more complex than the state machine, since the implementation must attempt to determine which state the object is in by querying the network connection.

9.4 A Case Study — The RunEditor Dialog

The RunEditor is a GUI class that controls the running of a series of tests on student assignments. The user can pick one or more students and select go. The actual running of the tests is done by a separate worker thread, which takes some time to stop.

The dialog includes 5 buttons, a progress bar, and a list of students.

Buttons are: Go, Dismiss, Stop, Select-All, Select-None



9.4.0 Statechart

The class has 3 major states

- Waiting. The user: can select students from list, initiate run (if students are selected), dismiss the dialog.
- Running. The user: can initiate a stop.
- Stop. The user must wait until the stop is complete.

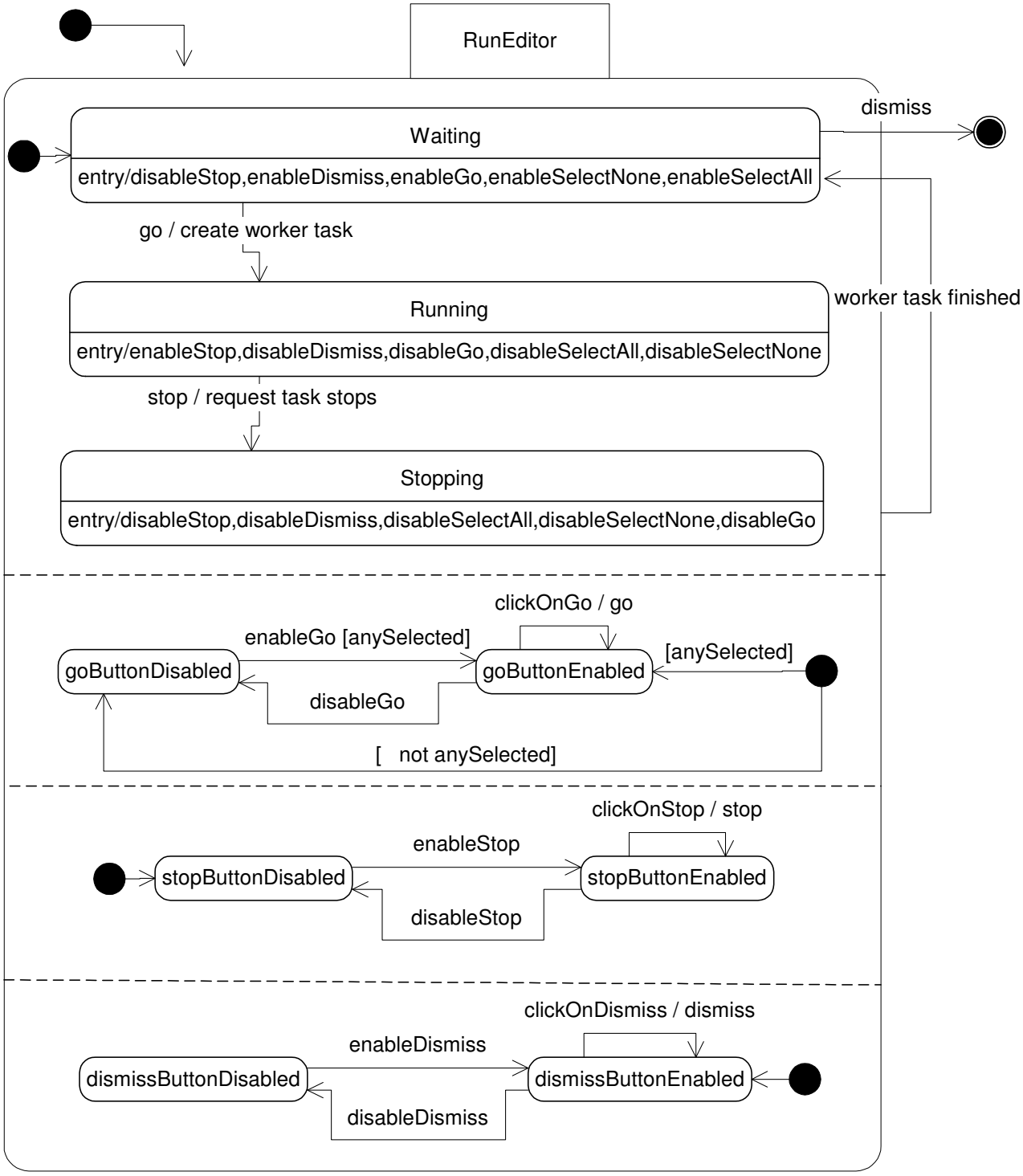
Note the Statechart on the next slide is a simplification as it omits the state of the selection buttons, the list, and the progress bar, as well as state inherited from JDialog.

Note that the state of the buttons is properly a part of the state of the RunEditor. This is because the relationship between these classes is one of Aggregation.

Data Dictionary:

External Entity	Kind	In/Out	Description
clickOnGo	event	in	User clicks on go button
clickOnStop	event	in	User clicks on stop button
clickOnDismiss	event	in	User clicks on dismiss button
create worker task	event	out	A worker thread is created.
request task stops	event	out	Request thread to stop.
worker task finished	event	in	A worker thread completes.

All other signals on the diagram are internal.


RunEditor
Waiting
entry/disableStop,enableDismiss,enableGo,enableSelectNone,enableSelectAll
dismiss
go / create worker task
Running
entry/enableStop,disableDismiss,disableGo,disableSelectAll,disableSelectNone
worker task finished
stop / request task stops
Stopping
entry/disableStop,disableDismiss,disableSelectAll,disableSelectNone,disableGo
clickOnGo / go
enableGo [anySelected]
goButtonDisabled
goButtonEnabled
[anySelected]
disableGo
[not anySelected]
clickOnStop / stop
enableStop
stopButtonDisabled
stopButtonEnabled
disableStop
clickOnDismiss / dismiss
enableDismiss
dismissButtonDisabled
dismissButtonEnabled
disableDismiss

9.4.1 Implementation

JButtons 'know' if the are enabled. We represent the state with fields

```
private final int WAITING = 0, RUNNING = 1, STOPPING = 2 , INIT
  = 3 ;
private int state = INIT ;
JButton goButton = new JButton();
JButton dismissButton = new JButton();
JButton stopButton = new JButton();
... // and more
```

(The INIT 'state' is used only during construction)

The abstraction relation relates the states in the chart (left) to those in the code (right)

```
      (in Waiting                iff  state=WAITING)
and   (in Running                iff  state=RUNNING)
and   (in Stopping               iff  state=STOPPING)
and   (in goButtonEnabled        iff  goButton.isEnabled())
and   (in goButtonDisabled       iff  not goButton.isEnabled())
and   (in stopButtonEnabled      iff  stopButton.isEnabled())
and   (in stopButtonDisabled     iff  not stopButtonIsEnabled())
and   (in dismissButtonEnabled   iff  dismissButton.isEnabled())
and   (in dismissButtonDisabled  iff  not dismissButton.isEnabled())
      ... and more
```

9.4.2 Implementation continued

It is useful to centralize the state switching code in one subroutine

```
private void changeState( int newState ) {
    if( newState == WAITING ) {
        if( studentList.getModel().getSize() != 0 )
            listLabel.setText("Select students...");
        else listLabel.setText("No students ...") ;
        removeProgressBar() ;
        goButton.getModel().setEnabled(anySlctd() );
        dismissButton.getModel().setEnabled( true );
        stopButton.getModel().setEnabled( false );
        allButton.getModel().setEnabled( true );
        noneButton.getModel().setEnabled( true ); }
    else if( newState == RUNNING ) {
        ... code to enter Running state... }
    else if( newState == STOPPING ) {
        ...code to enter Stopping State... }
    state = newState ;
}
```

Chapter 10

Context free grammars and context free parsing.

10.0 Grammars and Parsing

A formal language is simply a set of sequences. Usually we restrict our selves to *possibly infinite* sets of *finite sequences* over a *finite set* S. Formal language theory considers *finite* descriptions of languages. We are particularly interested in description methods that are

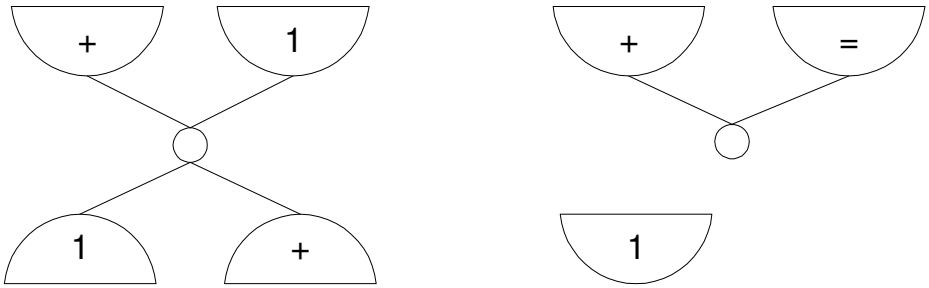
- easy to understand and use
- lead to algorithms for analyzing sequences
- suitable for automated processing

Finite recognizers meet these criteria, but there are many important languages that can not be described by finite recognizers because they (the languages) require more than a fixed amount of memory.

10.0.0 Unrestricted Grammars

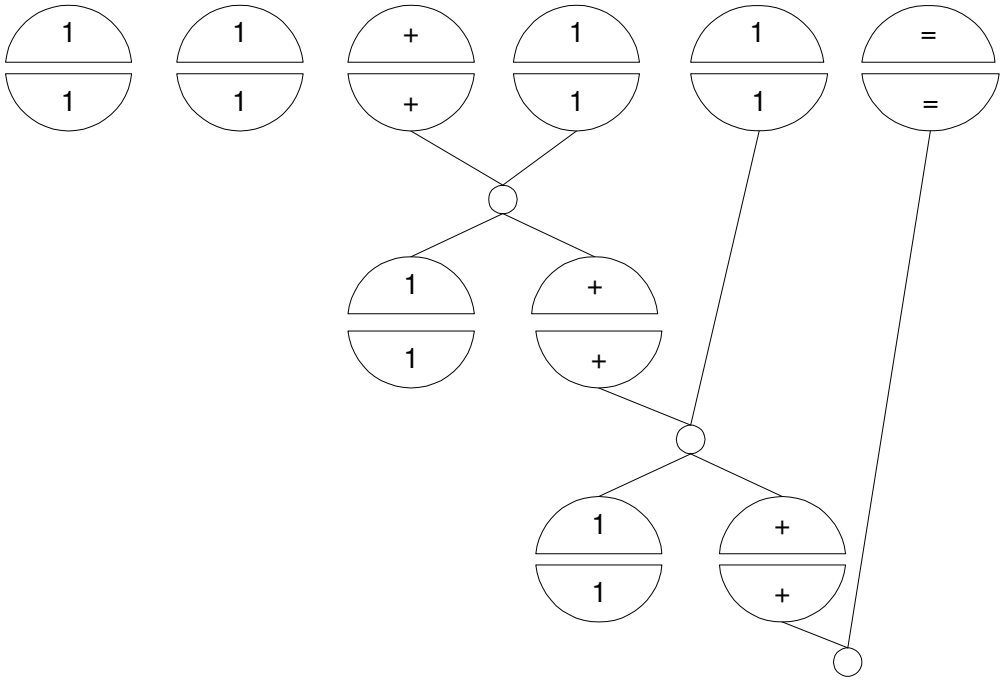
10.0.0.0 Puzzle 0

Suppose we have an unlimited supply of puzzle pieces of each of 3 shapes.



We'll call this supply, the pool of pieces. We can think of the pool as being a program. The puzzle starts with a sequence of pieces we'll call the input.

The goal is to close all the circles. In each step we can add one piece from our infinite pool. We can stretch the pieces, but can not alter the sequence of symbols along their top's or bottom's and can not rotate the pieces. We also can not cross lines. A solution to this puzzle is.



Now looking at the sequence of closed circles (with no lines at the bottom), we get:

1111

Thus our pool describes an algorithm (of sorts) for computing addition in tally notation.

In general, a set of puzzle pieces defines a relation from finite sequences to finite sequences.

Proposition 82 Any function from strings to strings that can be computed by an algorithm can be turned into a puzzle like this.

We call such a function a "computable function".

Exercise 83 Find a pool that will add numbers presented in binary notation. E.g. with input

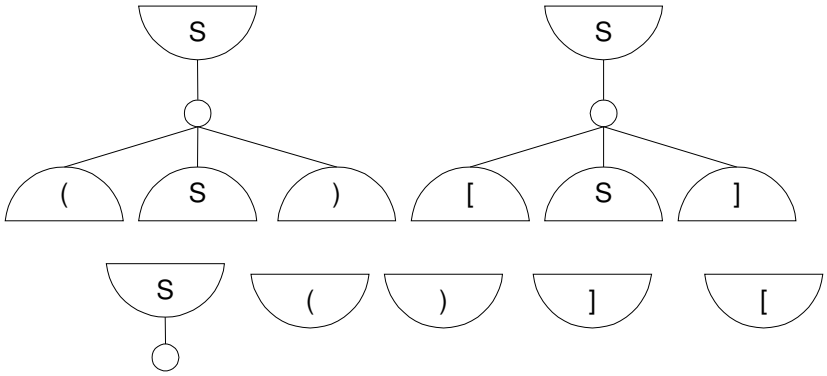
$$1100 + 111 =$$

the output will be 10011.

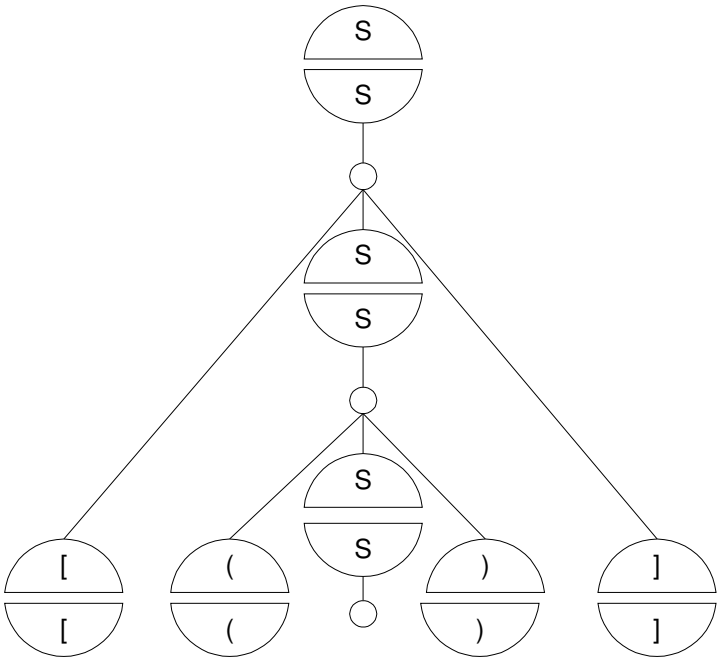
Exercise 84 Find a pool for multiplyng in tally notation: An input of $111 \times 11 =$ should result in an output of 111111.

10.0.0.1 Puzzle 1

For this puzzle we start always start with a symbol S and the 7 piece kinds are



We can make a tree



The final sequence is [()] other sequences we can reach are [[[(([]))]]] and the empty sequence.

This puzzle defines an infinite set of finite sequences over the *alphabet* {'(', ')', '[', ']'}.

Proposition 85 Any language that can be generated by an algorithm can be defined by a pool like this. (I.e., the algorithm produces a possibly infinite list containing each member of the language.)

We call set a language a "recursively enumerable language".

10.0.1 Handier Notation

To save space, we will use a more compact notation.

10.0.1.0 Puzzle 0 again

- We define a finite set of "terminal symbols" $S = \{1\}$.

- We define a finite set of "nonterminal symbols" (a.k.a. "variables") $V = \{\text{`+'},\text{`='}\}$.

- We define a finite set of "production rules"

$$P = \{\ \text{` + ' =} \longrightarrow \epsilon,\\ \text{` + ' `1'} \longrightarrow \text{`1' ` + '}\ \}$$

The above comprise a "grammar".

We solve the puzzle by starting with a sequence, say 11+111=, and replacing any occurrence of the right hand side of a production with its left hand side, stopping when only terminals remain

$$\begin{aligned} &11{+}111{=}\\ &\Longrightarrow 111{+}11{=}\\ &\Longrightarrow 1111{+}1{=}\\ &\Longrightarrow 11111{+}{=}\\ &\Longrightarrow 11111 \end{aligned}$$

10.0.1.1 Puzzle 1 again

- We define a finite set of "terminal symbols" or "alphabet symbols" $S = \{\text{`('},\text{`)'},\text{`['},\text{`]'}\}$.

- We define a finite set of "nonterminal symbols" $V = \{\text{Start}\}$.

- We define a finite set of "productions"

$$P = \{\ \text{Start} \longrightarrow \epsilon,\\ \text{Start} \longrightarrow \text{`(' Start `)'},\\ \text{Start} \longrightarrow \text{`[' Start `]'}\ \}$$

- We define a starting nonterminal Start.

We solve the puzzle by starting with the starting nonterminal and replacing left hand sides with right hand sides until we only have terminals

$$\begin{aligned}
&\text{Start}\\
&\Longrightarrow (\text{Start})\\
&\Longrightarrow ((\text{Start}))\\
&\Longrightarrow (([\text{Start}]))\\
&\Longrightarrow ((\left[\right]))
\end{aligned}$$

Unlike puzzle 0, we are faced with some choices.

10.0.2 Formalizing

10.0.2.0 Grammars

Definition: A "grammar" is a tuple $G = (V, S, P, A_{\text{start}})$ or $G = (V, S, P)$ where

- V is a finite set of nonterminal symbols
- S is a finite set of terminal symbols (disjoint from V)
- P is a finite set of production rules of the form $\alpha \longrightarrow \beta$, where α and β are finite strings over $V \cup S$ with at least one nonterminal in α.
- A_{start} is a member of V called the "start symbol"

10.0.2.1 Productions

If we have a production rule $\alpha \longrightarrow \beta$, we say a string $\gamma\alpha\delta$ "can produce" a string $\gamma\beta\delta$.

Definition: More formally, given a grammar $G = (V, S, P, A_{\text{start}})$ we say that η "can produce" κ exactly if there exist

- a production rule $(\alpha \longrightarrow \beta) \in P$
- and strings γ and δ over $V \cup S$ such that $\eta = \gamma\alpha\delta$ and $\kappa = \gamma\beta\delta$.

We write $\eta \Longrightarrow \kappa$ to mean η "can produce" κ

10.0.2.2 Derivation

Definition: If there exists a finite sequence of strings α_0, α_1, ... , α_n such that

$$\alpha = \alpha_0 \Longrightarrow \alpha_1 \Longrightarrow ... \Longrightarrow \alpha_n = \beta$$

then we say that α "derives" β. In notation:

$$\alpha \overset{*}{\Longrightarrow} \beta$$

And we say that $[\alpha, \alpha_1, ..., \beta]$ is a "derivation".

10.0.2.3 The function defined by a grammar

Each grammar $G = (V, S, P)$ or $G = (V, S, P, A_{\text{start}})$ defines a relation $r_G \in (V \cup S)^* \leftrightarrow S^*$ so that if $\alpha \overset{*}{\Longrightarrow} w \in S^*$ then

$$(\alpha, w) \in \text{graph}(r_G)$$

For some grammars, this relation is a function.

Every computable function is expressible as a grammar.

10.0.2.4 The language generated by a grammar

Definition: For a grammar $G = (V, S, P, A_{\text{start}})$ with a start symbol A_{start}, **the language generated by** the grammar is

$$L(G) = \{w \in S^* \mid A_{\text{start}} \overset{*}{\Longrightarrow} w\}$$

Note that $L(G) \subseteq S^*$. For example for G from puzzle 0 we have

$$L(G) = \{\epsilon, (), [], (()), ([]), [()], [[]], \ldots\}$$

Every language that can be recognized by some sort of digital computer can be expressed by a grammar.

10.0.3 Context Free Grammars

Puzzle 0 and Puzzle 1 have a significant difference. Puzzle 0 only produces trees. This is because each puzzle piece has only one semicircle in its top row. We call such a grammar "context free".

10.0.3.0 Definitions

Definition: A "context free grammar" is a grammar where each production rule is of the form

$$A \longrightarrow \beta$$

for some $A \in V$.

Definition: A "context free language" is a language generated by some context free grammar.

10.0.3.1 Significance

Con: There are (computable) languages that are not context free.

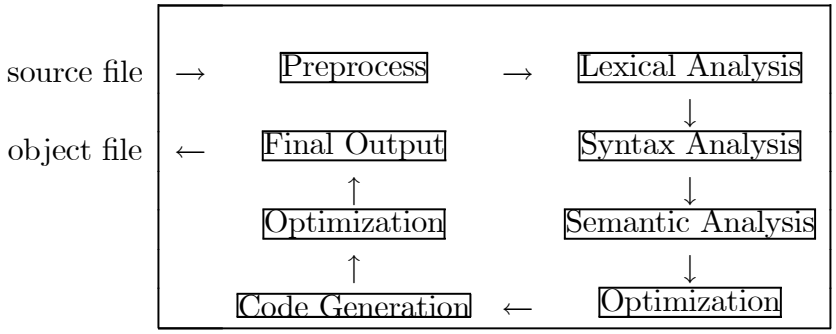
Pro: context free grammars:

- Are easy to use and understand
- Given a grammar, there is always an algorithm to determine whether or not a string is in the language generated by the grammar: $O(N^3)$
  - For common special cases there are fast algorithms: $O(N)$.
- Many useful and important languages are context free.
  - Example: The language of syntactically correct Java classes
  - Example: Well-formed XML documents.
  - Example: Valid XML documents.
  - Example: Correct usages of many communication protocols.
- For languages that are not context free, we can often start by defining a context free language and then restricting that language
  - Example: The language of compile-time error free Java classes.

10.1 Examples Of Context Free Grammars

10.1.0 Programming language examples

Typical compiler phase structure



Phase goals

- Preprocessing: character sequence to character sequence.
- Lexical analysis: character sequence to sequence of tokens
- Syntax analysis (aka parsing): token sequence to "abstract syntax tree"
- Semantic analysis: build symbol table and find errors
- Code Generation: Select instruction sequences
- Optimization: various time and space improvements
- Final output: output machine code (or assembly code).

10.1.0.0 Role of grammars: Grammars are used in

- **Preprocessing** to parse macro definitions & uses, includes, conditional compilation etc.
- **Lexical analysis** uses a grammar to describe how to break a sequence of characters into a sequence of "tokens"
  - spaces, newlines, and comments not output

- Example input to lexical analysis:

```
// Read two numbers
    var i :  float    read i
    var j :  float    read j
// find the average, and print it
    var k    k := (i+j)/2    print k
```

- Example Output:

  var, (**id**, i), **:**, **float**, ..., /, (**num**, 2), **print** (**id** k)

- **Syntax Analysis (parsing)** determines if the sequence of tokens is syntactically in the language and (typically) builds a tree representation.

- **Code generation** Grammars are sometimes used to describe sequences of operations that correspond to machine instructions.

10.1.0.1 A handy abbreviation:

We abbreviate multiple productions with the same left-hand side by writing

$$A \longrightarrow \alpha \mid \beta$$

to mean that both $A \longrightarrow \alpha$ and $A \longrightarrow \beta$ are productions.

10.1.0.2 Floating point numbers in C/C++ (lexical phase).

Terminals are characters written in `typewriter` font.

$$
\begin{aligned}
\textit{floatNum} &\longrightarrow \textit{fract optExp optFloatSufix} \\
&\mid \textit{digits exp optFloatSufix} \\
\textit{fract} &\longrightarrow \textit{optDigits} \texttt{.} \textit{digits} \mid \textit{digits} \texttt{.} \\
\textit{digits} &\longrightarrow \textit{digit} \mid \textit{digit digits} \\
\textit{exp} &\longrightarrow \texttt{E}\ \textit{sign digits} \\
\textit{exp} &\longrightarrow \texttt{e}\ \textit{sign digits} \\
\textit{optExp} &\longrightarrow \epsilon \mid \textit{exp} \\
\textit{optDigits} &\longrightarrow \epsilon \mid \textit{digits} \\
\textit{optFloatSufix} &\longrightarrow \epsilon \mid \texttt{f} \mid \texttt{l} \mid \texttt{F} \mid \texttt{L} \\
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \\
&\mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{sign} &\longrightarrow \epsilon \mid + \mid -
\end{aligned}
$$

Example strings in the language:

```
123.456     .456E+789     123E798
```

Not in the language:

```
123$456
.456.789
123
123E0.1
```

An example derivation:

$$
\begin{aligned}
\textit{floatNum} &\Longrightarrow \textit{fract optExp } \underline{\textit{optFloatSufix}} \\
&\Longrightarrow \textit{fract } \underline{\textit{optExp}} \\
&\Longrightarrow \underline{\textit{fract}} \\
&\Longrightarrow \underline{\textit{optDigits}} \textit{ . digits} \\
&\Longrightarrow \underline{\textit{digits}} \textit{ . digits} \\
&\Longrightarrow \textit{digit . } \underline{\textit{digits}} \\
&\Longrightarrow \textit{digit . digit } \underline{\textit{digits}} \\
&\Longrightarrow \underline{\textit{digit}} \textit{ . digit digit} \\
&\Longrightarrow \texttt{1} \textit{ . } \underline{\textit{digit}} \textit{ digit} \\
&\Longrightarrow \texttt{1 . 2} \underline{\textit{digit}} \\
&\Longrightarrow \texttt{1 . 2 3}
\end{aligned}
$$

10.1.0.3 A simple programming language (lexical level, partial grammar)

Terminals are all ASCII characters

$$
\begin{aligned}
\mathit{token} &\longrightarrow \mathit{spaces}\ \mathit{tk} \\
\mathit{tk} &\longrightarrow \mathit{keyword} \mid \mathit{id} \mid \mathit{num} \mid \mathit{punc} \mid \mathit{op} \mid \mathit{EOF} \\
\mathit{spaces} &\longrightarrow \epsilon \mid \mathit{space}\ \mathit{spaces} \\
\mathit{space} &\longrightarrow \mathit{spacechar} \mid \mathit{newlinechar} \\
&\mid \mathit{tabchar} \mid \mathit{comment} \\
\mathit{comment} &\longrightarrow /\ /\ \mathit{nonnewlines}\ \mathit{newlinechar} \\
\mathit{nonnewlines} &\longrightarrow \epsilon \mid \mathit{nonnewline}\ \mathit{nonnewlines} \\
\mathit{nonnewline} &\longrightarrow \mathit{alpha} \mid \mathit{digit} \mid \texttt{(} \mid \texttt{)} \mid + \mid - \mid / \mid * \mid \cdots \\
\mathit{keyword} &\longrightarrow \texttt{p r i n t} \\
&\mid \texttt{r e a d} \\
&\mid \ldots \\
\mathit{id} &\longrightarrow \mathit{alpha}\ \mathit{alphasOrDigits} \\
\mathit{num} &\longrightarrow \mathit{digit} \mid \mathit{digit}\ \mathit{num} \\
\mathit{alphasOrDigits} &\longrightarrow \epsilon \mid \mathit{alphaOrDigit}\ \mathit{alphasOrDigits} \\
\mathit{alphaOrDigit} &\longrightarrow \mathit{alpha} \mid \mathit{digit} \\
\mathit{alpha} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \\
\mathit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{punc} &\longrightarrow (\mid) \\
\mathit{op} &\longrightarrow + \mid - \mid / \mid * \mid = \mid !=
\end{aligned}
$$

10.1.0.4 Expressions for a programming language (syntactic level)

Terminals are **(**, **)**, **num**, **id**, +, -, /, ***, =,** and **!=**. Starting nonterminal is *exp*

$$
\begin{aligned}
exp &\longrightarrow \mathbf{num} \\
&\quad | \ \mathbf{id} \\
&\quad | \ exp \ bop \ exp \\
&\quad | \ \ uop \ exp \\
&\quad | \ \mathbf{(} \ exp \ \mathbf{)} \\
bop &\longrightarrow + \ | \ \mathbf{-} \ | \ \mathbf{*} \ | \ / \ | = | \ \mathbf{!=} \\
uop &\longrightarrow + \ | \ \mathbf{-}
\end{aligned}
$$

Example string in the language:

(num + num) / - id = id / num / id

Not in the language:

(num + (id * id)))
- num (num * * id)

10.1.0.5 A simple programming language (syntactic level)

Terminals are as in the previous example plus **print**, **read**, **var**, **if**, **else**, **end**, **while**, **int**, **float**, **bool**, **eof**

$$
\begin{aligned}
\mathit{prog} &\longrightarrow \mathit{stat}\ \textbf{eof} \\
\mathit{stat} &\longrightarrow \textbf{print}\ \mathit{exp} \\
&\quad \mid \textbf{read}\ v \\
&\quad \mid \textbf{var}\ v : \mathit{type} \\
&\quad \mid v := \mathit{exp} \\
&\quad \mid \textbf{if}\ \mathit{exp}\ \mathit{stat}\ \textbf{else}\ \mathit{stat}\ \textbf{end} \\
&\quad \mid \textbf{while}\ \mathit{exp}\ \mathit{stat}\ \textbf{end} \\
&\quad \mid \mathit{stat}\ \mathit{stat} \\
v &\longrightarrow \textbf{id} \\
\mathit{type} &\longrightarrow \textbf{int} \mid \textbf{float} \mid \textbf{bool} \\
\mathit{exp} &\longrightarrow \textit{as in previous example}
\end{aligned}
$$

Strings in the language:

var id : int read id print num + id eof
if num id := num else id := id end eof

Note that there may still be "semantic" errors.

Strings not in the language:

var id read num print num + eof
if num id := num else id := id eof

10.1.1 Internet applications

10.1.1.0 HTML tags (as Netscape and IE recognize them)

$$
\begin{aligned}
\mathit{startTag} &\longrightarrow < \mathit{elementName}\ \mathit{attributes} > \\
\mathit{elementName} &\longrightarrow \mathit{letter}\ \mathit{moreElementName} \\
\mathit{letter} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\mathit{moreElementName} &\longrightarrow \mathit{nonSpace}\ \mathit{moreElementName} \mid \epsilon \\
\mathit{nonSpace} &\longrightarrow \mathit{letter} \mid \ldots \\
\mathit{attributes} &\longrightarrow \mathit{etc.}
\end{aligned}
$$

Attributes is a bit complex, so let's leave it for now

10.1.1.1 The http URI

$$
\begin{aligned}
\mathit{http\_URL} &\longrightarrow \texttt{h t t p :\ \ / /}\ \mathit{host}\ \mathit{optPort}\ \mathit{optAbsPath} \\
\mathit{optPort} &\longrightarrow \epsilon \mid \texttt{:}\ \mathit{port} \\
\mathit{optAbsPath} &\longrightarrow \epsilon \mid \mathit{absPath} \mid \mathit{absPath}\ ?\ \mathit{query}
\end{aligned}
$$

Hosts and ports are defined by

$$
\begin{aligned}
\mathit{host} &\longrightarrow \mathit{hostName} \mid \mathit{iPv4address} \\
\mathit{hostName} &\longrightarrow \mathit{labels}\ \mathit{optDot} \\
\mathit{labels} &\longrightarrow \mathit{dlabel}\ .\ \mathit{labels} \mid \mathit{tlabel} \\
\mathit{dlabel} &\longrightarrow \mathit{alphaNum} \mid \mathit{alphaNum}\ \mathit{labelChars}\ \mathit{alphaNum} \\
\mathit{tlabel} &\longrightarrow \mathit{alpha} \mid \mathit{alpha}\ \mathit{labelChars}\ \mathit{alphaNum} \\
\mathit{labelChars} &\longrightarrow \mathit{alphaNum} \mid \texttt{-} \\
\mathit{optDot} &\longrightarrow \epsilon \mid \texttt{.} \\
\mathit{iPv4address} &\longrightarrow \mathit{digits}\ \texttt{.}\ \mathit{digits}\ \texttt{.}\ \mathit{digits}\ \texttt{.}\ \mathit{digits} \\
\mathit{digits} &\longrightarrow \mathit{num} \mid \mathit{num}\ \mathit{digits} \\
\mathit{alphaNum} &\longrightarrow \mathit{alpha} \mid \mathit{num} \\
\mathit{alpha} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\mathit{num} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{port} &\longrightarrow \epsilon \mid \mathit{num}\ \mathit{port}
\end{aligned}
$$

Paths

$$\begin{aligned} \textit{absPath} &\longrightarrow \textbf{/}\ \textit{segments} \\ \textit{segments} &\longrightarrow \textit{segment} \mid \textit{segment}\ \textbf{/}\ \textit{segments} \\ \textit{segment} &\longrightarrow \text{etc} \\ \textit{query} &\longrightarrow \text{etc} \end{aligned}$$

I won't go into all the details, but just comment that a segment is a sequence of almost any characters, as is a query.

10.1.1.2 Protocols

In this case the tokens are requests and replies. Requests go from client to server and replies from server to client.

This is a greatly simplified FTP (File Transfer Protocol).

$$\begin{aligned} \textit{session} &\longrightarrow \textbf{greetingRequest greetingReply}\ \textit{moreSesn} \\ \textit{moreSesn} &\longrightarrow \textbf{quitRequest quitReply} \\ &\mid \textbf{sendFileRequest sendFileReply}\ \textit{moreSesn} \\ &\mid \textbf{sendFileRequest errorReply}\ \textit{moreSesn} \\ &\mid \textbf{getFileRequest getFileReply}\ \textit{moreSesn} \\ &\mid \textbf{getFileRequest errorReply}\ \textit{moreSesn} \end{aligned}$$

The syntax of the various requests and replies can also be specified in terms of sequences of bytes, just as tokens are specified in compilers.

10.2 Recognition and Parsing

Given a grammar G, the recognition problem is this

Input: A string w of terminal symbols.

Output: Whether or not w is in $L(G)$

Parsing problems are similar, but the output also includes a useful data structure when $w \in L(G)$.

For example:

- In a compiler: we might output an abstract syntax tree.

- In a calculator: we might output the numerical value of an expression.
- We might output machine code or a reverse polish notation (RPN) representation of the input.

10.3 Derivation Trees and Left-most Derivations

Definition 86 A left-most derivation is one where, at each step, the left-most nonterminal is replaced. We write $\alpha \Longrightarrow_{\text{lm}} \beta$

More formally, have $\alpha \Longrightarrow_{\text{lm}} \beta$ iff there are A, s, γ, and δ such that $\alpha = sA\gamma$ and $A \longrightarrow \delta$ and $\beta = s\delta\gamma$. (Recall that $s \in S^*$).

We write $\alpha \overset{*}{\Longrightarrow}_{\text{lm}} \beta$ to indicate that there is a derivation of β from α in 0 or more left-most production steps.

Given a grammar G and a string s in $L(G)$ we can consider a tree that illustrates the proof that the tree is in the language. Any derivation corresponds to a "derivation tree".

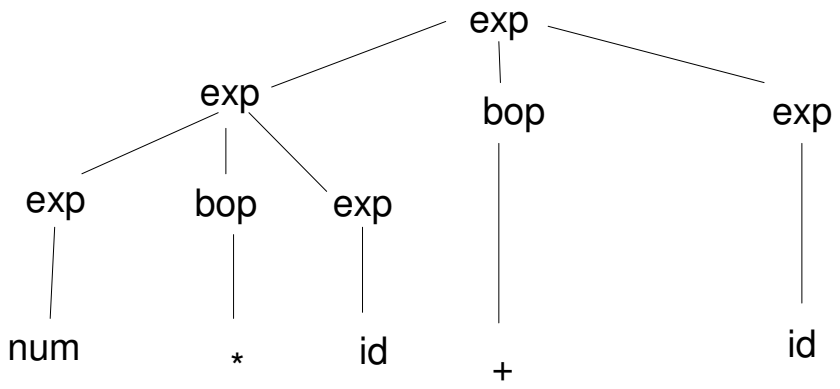
Example

$$
\begin{aligned}
&\textit{exp} \longrightarrow \textbf{num} \mid \textbf{id} \mid \textit{exp}\ \textit{bop}\ \textit{exp} \mid \ \textit{uop}\ \textit{exp} \mid (\ \textit{exp}\ )\\
&\textit{bop} \longrightarrow + \mid \texttt{-} \mid / \mid \texttt{*} \mid = \mid \texttt{!=}\\
&\textit{uop} \longrightarrow + \mid \texttt{-}
\end{aligned}
$$

Consider the input 2*i+j, which as a string of terminals is: **num * id + id**. One derivation is

$$
\begin{aligned}
\textit{exp} &\Longrightarrow \textit{exp}\ \textit{bop}\ \underline{\textit{exp}}\\
&\Longrightarrow \underline{\textit{exp}}\ \textit{bop}\ \textbf{id}\\
&\Longrightarrow \textit{exp}\ \textit{bop}\ \underline{\textit{exp}}\ \textit{bop}\ \textbf{id}\\
&\Longrightarrow \textit{exp}\ \underline{\textit{bop}}\ \textbf{id}\ \textit{bop}\ \textbf{id}\\
&\Longrightarrow \underline{\textit{exp}} * \textbf{id}\ \textit{bop}\ \textbf{id}\\
&\Longrightarrow \textbf{num} * \textbf{id}\ \underline{\textit{bop}}\ \textbf{id}\\
&\Longrightarrow \textbf{num} * \textbf{id} + \textbf{id}
\end{aligned}
$$

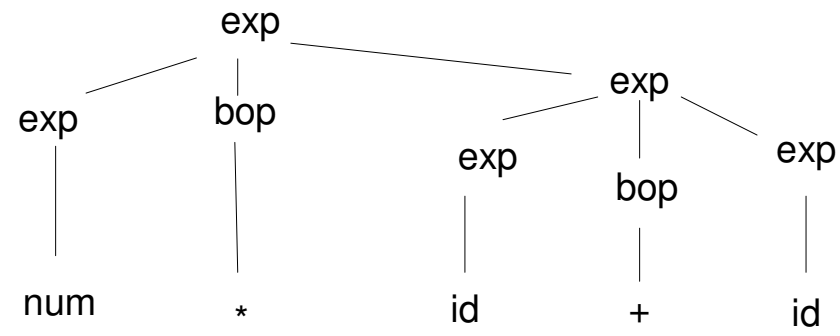
from which we can build the following "derivation tree".



We can build a "left-most derivation" by traversing the tree depth-first and left to right, expanding the nonterminal that we encounter

$$
\begin{aligned}
\underline{exp} &\Longrightarrow_{\text{lm}} \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \underline{exp}\ bop\ exp\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \underline{bop}\ exp\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ \underline{bop}\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ +\ \underline{exp} \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ +\ \underline{exp}
\end{aligned}
$$

But with the same grammar and the same string, we can build a *different* tree.



The corresponding left-most derivation

$$\begin{aligned}
\underline{exp} &\Longrightarrow_{\text{lm}} \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \underline{bop}\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \underline{exp} \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ \underline{bop}\ exp \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \underline{exp} \\
&\Longrightarrow_{\text{lm}} \mathbf{num}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id}
\end{aligned}$$

10.3..3 Ambiguity

Definition: We say that a grammar is *ambiguous* iff for some string there exist two or more derivation trees.

Equivalently: a grammar is ambiguous iff some string has two or more left-most derivations.

For many applications, we should avoid ambiguous grammars since

- other matters (semantics) are generally described in terms of the grammar and we don't want ambiguous semantics.
- it is hard to build an efficient parser for ambiguous grammars.

For other applications: e.g. natural language understanding, ambiguity is useful.

Consider

- "I saw a bird with a telescope"
- "I saw a man with a hat"
- "I saw a man with a telescope"

All 3 sentences fit the pattern

pronoun verb det noun prep det noun

but, in the first, the prepositional phrase attaches to the verb, whereas in the second the prepositional phrase attaches to the object. This means we want multiple derivation trees for the same sequence of word forms.

10.3.0 Ambiguity and expression grammars

Here is a grammar for expressions Exp0

$$E \longrightarrow \mathbf{n} \mid (\ E\ ) \mid E + E \mid E \text{ - } E \mid E * E \mid E \ / \ E$$

This grammar is highly ambiguous.

How many derivation trees are there for **n - n / n / n - n** ?

14? In a sense only 1 reflects the correct precedence and associativity of the operators.

To 'enforce' 'correct' parsing of expressions we can write a new grammar, Exp1

$$\begin{aligned} E &\longrightarrow T \mid E + T \mid E \text{ - } T \\ T &\longrightarrow F \mid T * F \mid T \ / \ F \\ F &\longrightarrow \mathbf{n} \mid (\ E\ ) \end{aligned}$$

This grammar is unambiguous.

10.3.0.0 Left-recursion

A nonterminal A is said to be *left-recursive* if there is a derivation

$$A \stackrel{*}{\Longrightarrow} A\beta$$

with at least one step (for some β).

A grammar is *left-recursive* iff it has at least one left-recursive nonterminal.

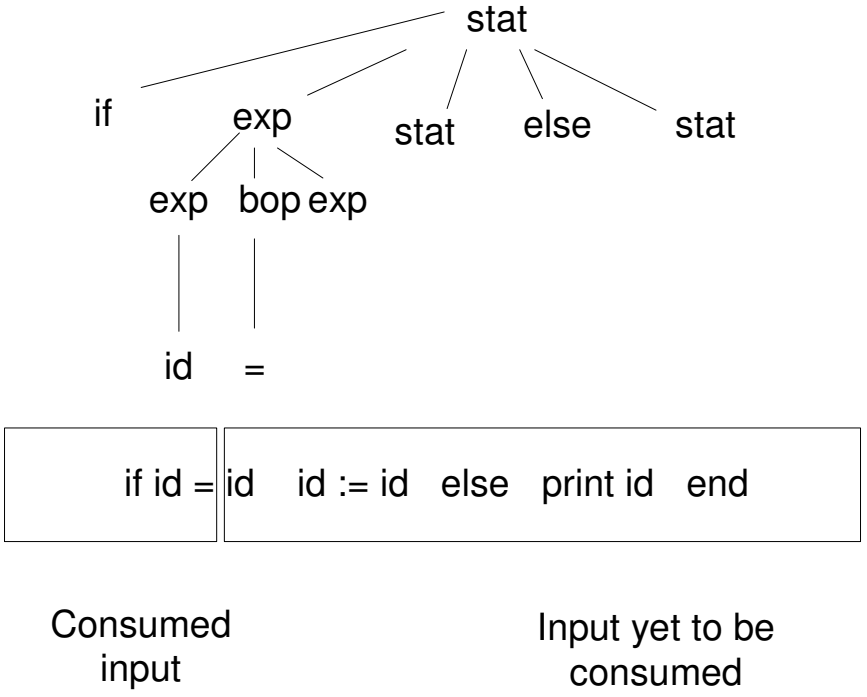
Clearly both Exp0 and Exp1 are left recursive. Here is an unambiguous grammar for the same language as Exp0 and Exp1 that is not left-recursive. Exp2:

$$\begin{aligned} E &\longrightarrow T\ E1 \\ E1 &\longrightarrow +\ T\ E1 \mid \text{-}\ T\ E1 \mid \epsilon \\ T &\longrightarrow F\ T1 \\ T1 &\longrightarrow *\ F\ T1 \mid /\ F\ T1 \mid \epsilon \\ F &\longrightarrow \mathbf{n} \mid (\ E\ ) \end{aligned}$$

10.4 Top-Down predictive parsing and recognition.

A top-down predictive parser works by trying to build the derivation tree from the top down.

For example here is a derivation tree that is partially built.



The leaves of the tree are **if id =** *exp stat* **else** *stat*
This tree is a proof that

$$stat \stackrel{*}{\Longrightarrow} \textbf{if id } = exp\ stat\ \textbf{else}\ stat$$

10.4.0 Conceptual view

The idea is to walk the derivation tree in a depth-first manner, while (conceptually) building the tree and consuming the input. Circled nodes are not yet visited. The left to right sequence of circled nodes is called *the prediction*.

10.4.1 Augmenting the grammar

It will simplify things later if we augment the grammar with

- a new terminal: \$
- a new starting nonterminal A'_{start} and

- a new production rule $A'_{\text{start}} \longrightarrow A_{\text{start}}$ \$, where A_{start} was the original starting nonterminal.

We will also add a \$ to the end of each input string.
(Typically \$ represents the end of the file.)

10.4.2 States, steps, and stops

We will process one terminal at a time.

10.4.2.0 States

A top-down predictive parser's state consists of

- the input processed so far s
- and the current prediction α
- and the remaining input sequence, t

Let's write this as

$$s_{\blacktriangle}\alpha,\ t$$

(Note: we don't represent the partially built tree, but only the sequence of unvisited nodes α).

As a loop invariant we'll have that

$$A'_{\text{start}} \stackrel{*}{\Longrightarrow}_{\text{lm}} s\alpha$$

and that $st = w\$$ where w is the original input.

As an initial state we'll start with

$$\epsilon_{\blacktriangle}A'_{\text{start}},\ w\$$$

where w is the input string.

10.4.2.1 Steps

We use two rules to step from one state to another

- Shift: (Read one terminal from the input)

$$s_{\blacktriangle} a\beta,\ au \quad \vdash \quad sa_{\blacktriangle}\beta,\ u$$

- Produce: (Expand the left most nonterminal)

$$s_{\blacktriangle} A\beta,\ t \quad \vdash \quad s_{\blacktriangle}\gamma\beta,\ t$$

  where $A \longrightarrow \gamma$ is a production rule.

Theorem: $A_{\text{start}} \stackrel{*}{\Longrightarrow} w$ iff there is a sequence of steps that goes

$$\epsilon_{\blacktriangle} A'_{\text{start}},\ w\$ \quad \vdash \quad \ldots \quad \vdash \quad w\$_{\blacktriangle}\epsilon,\ \epsilon$$

10.4.2.2 Stopping

- Successful stop: We stop when the state is $s_{\blacktriangle}\epsilon,\ \epsilon$, in which case (from the invariant) the input w is in the original grammar.
- Error stop 0: We also stop when the predicted input does not match the actual input, i.e. the state is $s_{\blacktriangle} a\beta,\ bu$ where $a \neq b$.
- Error stop 1: We also stop when the state is $s_{\blacktriangle} A\beta,\ t$ and there is no appropriate production rule.

If we come to a "Successful stop" then

- the input was in the language. $A_{\text{start}} \stackrel{*}{\Longrightarrow} w$

If we come to an "Error stop" then either

- the input was not in the language,
- or we made a bad choice in a 'produce' step.

Later we'll see how to avoid bad choices.

10.4.3 Example

Here is a trace of the algorithm for grammar Exp2 (augmented) and an input of **n * n**

	$s_{\blacktriangle}\alpha$,	t	**Action**
	$\epsilon_{\blacktriangle}E'$,	**n * n $**	Produce $E' \longrightarrow E\$$
$\vdash$	$\epsilon_{\blacktriangle}E$ \$,	**n * n $**	Produce $E \longrightarrow T\ E1$
$\vdash$	$\epsilon_{\blacktriangle}T\ E1$ \$,	**n * n $**	Produce $T \longrightarrow F\ T1$
$\vdash$	$\epsilon_{\blacktriangle}F\ T1\ E1$ \$,	**n * n $**	Produce $F \longrightarrow$ **n**
$\vdash$	$\epsilon_{\blacktriangle}$**n** $T1\ E1$ \$,	**n * n $**	Shift
$\vdash$	**n**$_{\blacktriangle}T1\ E1$ \$,	*** n $**	Produce $T1 \longrightarrow$ ***** $F\ T1$
$\vdash$	**n**$_{\blacktriangle}$***** $F\ T1\ E1$ \$,	*** n $**	Shift
$\vdash$	**n *** $_{\blacktriangle}\ F\ T1\ E1$ \$,	**n $**	Produce $F \longrightarrow$ **n**
$\vdash$	**n *** $_{\blacktriangle}$**n** $T1\ E1$ \$,	**n $**	Shift
$\vdash$	**n * n** $_{\blacktriangle}T1\ E1$ \$,	**$**	Produce $T1 \longrightarrow \epsilon$
$\vdash$	**n * n** $_{\blacktriangle}E1$ \$,	**$**	Produce $E1 \longrightarrow \epsilon$
$\vdash$	**n * n** $_{\blacktriangle}$\$,	**$**	Shift
$\vdash$	**n * n $**$_{\blacktriangle}\epsilon$,	ϵ	Success

So the string is in the language.

10.4.4 In a more algorithmic form

Note that s is not really needed in the state unless we use it to help pick the production rule, which we will not do.

Algorithm: Top down predictive parsing

Input: A string w

Output: 'success' or 'error'. If the output is 'success' then $w \in L(G)$. If the output is 'error', then either $w \notin L(G)$ or a bad choice was made.

```
var t := w$ // where w is the input string
var α := A'_start // Note: α behaves as a stack.
while α ≠ ε do
    if α(0) ∈ S then
        if α(0) = t(0) then (
            // Shift step
            t := tail(t)
```

```
            α := tail(α) )
        else  // α(0) ≠ t(0)
            error
    else /* α(0) ∈ V  */ (
        try to pick a suitable production rule α(0) → γ
        if a suitable production rule exists
            // Produce step
            α := (γ^tail(α))
        else  // no suitable production rule exists
            error ) )
if t = $ then success else error
```

(tail(s) is the string $[s(1), s(2), ..., s(\|s\| - 1)]$. error means stop with output 'error'. success means stop with output 'success')

Assuming the grammar is not left-recursive, the top-down predictive parsing algorithm must terminate and is $O(N)$ time, where N is the length of the input.

To be done: We still haven't said how to pick a suitable production rule when a nonterminal comes to the top of the α stack.

10.4.5 LL(1) Grammars

Definition: An augmented grammar is called an 'LL(1) grammar' when the suitable production rule can always be chosen on the basis of the next input item $t(0)$ and the left-most nonterminal $\alpha(0)$.

Left-recursive grammars can never be LL(1). (Why?)

For each production rule $A \longrightarrow \gamma$ we compute the set of terminals which $t(0)$ might equal when $A \longrightarrow \gamma$ is chosen as the production rule in a successful run of the TDPP algorithm.

This set is called the *selector set* of the algorithm.

So let's consider Exp2 augmented

$E' \longrightarrow E\$$	$\{\mathbf{n}, (\}$	$T1 \longrightarrow \mathbf{*}\ F\ T1$	$\{\mathbf{*}\}$
$E \longrightarrow T\ E1$	$\{\mathbf{n}, (\}$	$T1 \longrightarrow /\ F\ T1$	$\{/\}$
$E1 \longrightarrow +\ T\ E1$	$\{+\}$	$T1 \longrightarrow \epsilon$	$\{\$,), +, \mathbf{-}\}$
$E1 \longrightarrow \mathbf{-}\ T\ E1$	$\{\mathbf{-}\}$	$F \longrightarrow \mathbf{n}$	$\{\mathbf{n}\}$
$E1 \longrightarrow \epsilon$	$\{\$,)\}$	$F \longrightarrow (\ E\ )$	$\{(\}$
$T \longrightarrow F\ T1$	$\{\mathbf{n}, (\}$		

On the right we have a selector set for each production rule.

We implement the picking part of the algorithm

/*do pick a suitable production rule $\alpha(0) \rightarrow \gamma$ by*/

if there is a production rule $\alpha(0) \rightarrow \gamma$
with $t(0)$ in its selector set **then**

pick that production rule

else

there is no suitable production rule

A grammar is LL(1) iff for each nonterminal A and for each pair of productions for A, the selectors sets of the two productions are disjoint.

I.e. a grammar is LL(1) iff, for all A, α, β, such that $A \longrightarrow \alpha$ and $A \longrightarrow \beta$ are distinct productions,

$$\left(\begin{array}{ll} & \text{SelectorSet}\,(A \longrightarrow \alpha) \\ \cap & \text{SelectorSet}\,(A \longrightarrow \beta) \end{array} \right) = \emptyset$$

10.4.5.0 Computing the selector sets.

Consider the selector set for a production rule $A \longrightarrow \alpha$

Look at $E1 \longrightarrow + \ E$. It is clear that this production rule should only be picked if the next terminal is a + sign.

From the example it is clear that if $\alpha \stackrel{*}{\Longrightarrow} b\beta$ then b should be in the selector set of $A \longrightarrow \alpha$.

But there is more to it then that when $\alpha \stackrel{*}{\Longrightarrow} \epsilon$.

Consider $E1 \rightarrow \epsilon$ the next item in the input should be one that could legitimately follow an $E1$ in a successful derivation. Only items that could follow E qualify and these are \$ and).

Suppose $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A b \gamma \Longrightarrow \beta \alpha b \gamma \stackrel{*}{\Longrightarrow} \beta b \gamma$ then b should be in the selector set of $A \longrightarrow \alpha$.

(Note that $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A \Longrightarrow \beta\alpha \stackrel{*}{\Longrightarrow} \beta$ is not possible!)

Define functions First and Follow for an augmented grammar by

- $b \in \text{First}(\alpha)$ iff $\alpha \stackrel{*}{\Longrightarrow} b\beta$, for some β and
- $b \in \text{Follow}(A)$ iff $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A b \gamma$, for some β and γ.

The selector set for $A \longrightarrow \alpha$ is

$$\text{First}(\alpha), \text{ when } \alpha \stackrel{*}{\not\Longrightarrow} \epsilon$$

and is

$$\text{First}(\alpha) \cup \text{Follow}(A), \text{ when } \alpha \stackrel{*}{\Longrightarrow} \epsilon$$

10.5 Recursive Descent Parsing

Recursive descent parsing works on the same principle as our state based parser, but uses the call-return stack rather than an explicit stack.

To write a recursive descent parser, we create a subroutine for each nonterminal.

Let t be the input stream (a global variable)

The subroutine for nonterminal A has as its specification.

if $\left(\exists u, v \cdot t = uv \wedge A \stackrel{*}{\Longrightarrow} u\right)$
then $t :=$ some v such that $\left(\exists u \cdot t = uv \wedge A \stackrel{*}{\Longrightarrow} u\right)$
else error

I.e., each subroutine is responsible for recognizing a string u produced by its nonterminal in the input and removing that string from the remaining input.

Furthermore, each subroutine is responsible for reporting an error if the remaining input does not start with a string that can be derived from A.

10.5.0 Recursive Descent parsing of LL(1) grammars

If the grammar is LL(1) then creating an R.D. parser for it is a mechanical process.

Let's look at an example. We can write a recursive descent recognizer for Exp2 as follows

global var t ·

procedure main **is**
$t := w$ '**$**' ; // Where w is the input
SPrime

```
procedure consume is
    t := tail(t)

procedure expect( a ) is
    if t(0) = a then consume
    else error

procedure SPrime is
    E ;
    expect( '$' )

procedure E is
    if t(0) ∈ {'n','('} then (
        T ;
        E1 )
    else
        error

procedure E1 is
    if t(0) = '+' then (
        consume ;
        T ;
        E1 )
    else if t(0) = '-' then (
        consume ;
        T ;
        E1 )
    else if t(0) ∉ {'$',')'} then
        error
```

... T and T1 are similar to E and E1 ...

```
procedure F is
```

if $t(0) =$ '**n**' **then**
    consume
else if $t(0) =$ '**(**' **then** (
    consume ;
    E ;
    expect('**)**'))
else error

10.5.1 Getting results

Often we not only want to recognize the input but also process the input to create an output in the case where the input is recognized.

We can do this by augmenting the recursive decent parser with extra code.

As an example, we will produce numbers.

global var t ·

procedure main **is**
    $t := w$ '**$**' ; // Where w is the input
    output SPrime

procedure SPrime **is**
    var $p :=$ E ·
    expect('**$**')
    return p

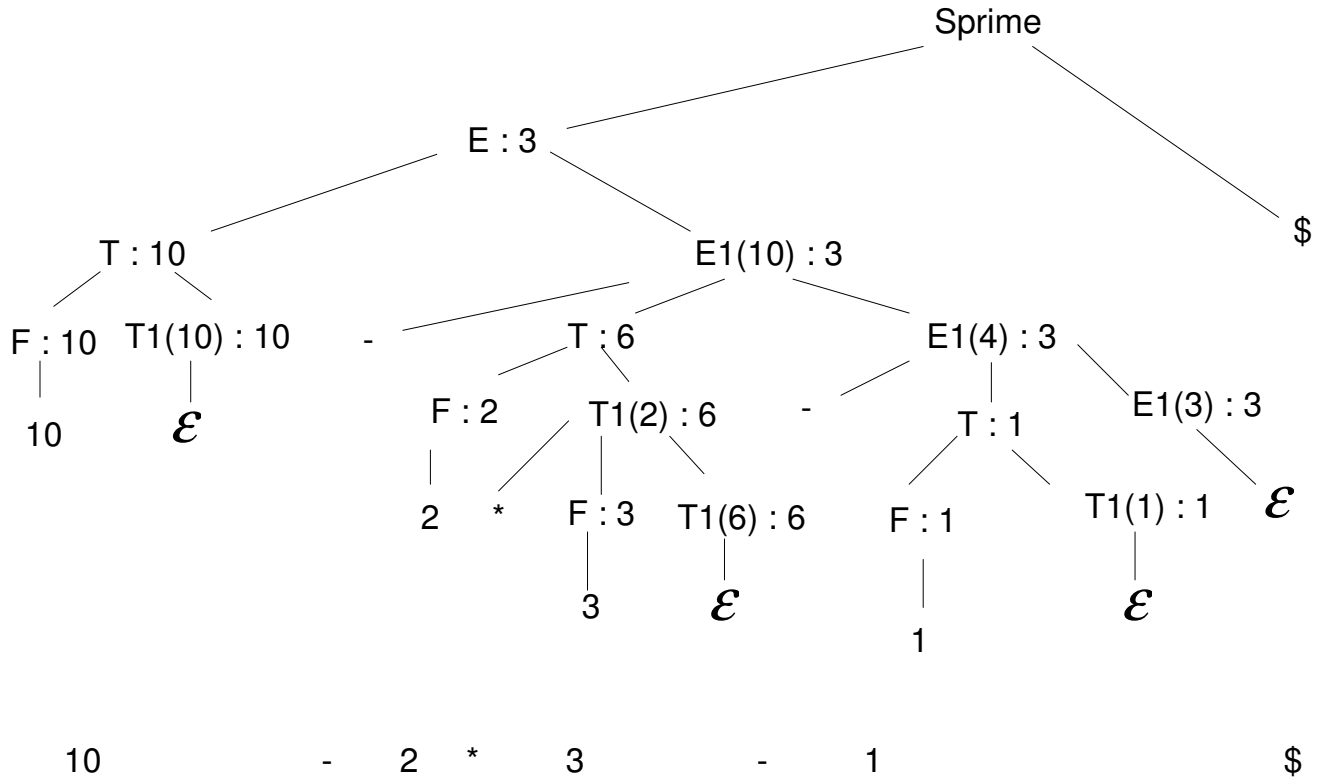
```
procedure E is
    if t(0) ∈ {'n','-'} then (
        var p := T ·
        return E1(p))
    else
        error

procedure E1( p ) is
    if t(0) = '+' then (
        t := tail(t) ;
        var q := T ·
        return E1( p + q ) )
    else if t(0) = '-' then (
        t := tail(t) ;
        var q := T ·
        return E1( p − q ) )
    else if t(0) ∉ {'$',')'} then
        error
    else
        return p
```

... T and T1 are similar to E and E1 ...

```
procedure F is
    if t(0) = 'n' then (
        var p:= the value associated with t(0)·
        t := tail(t) ;
        return p )
    else if t(0) = '(' then (
        t := tail(t)
        var p:= E then (
        expect( ')' ) )
        return p
    else error
```

Consider parsing the sequence: **10-2*3-1** As a string of terminals we have **n-n*n-n**.

We have the following call tree which reflects the parse tree. Return values are shown after the colon.



Parsing 10-2*3-1 with the parameter values and the return values shown.

Questions to consider:

- Can you modify the recursive descent parser above to produce an abstract syntax tree for an expression?

- Can you modify the recursive descent parser above to produce RPN?
- How could you alter the top-down predictive recognizer to compute results rather than to just recognize? *Hint: consider adding "commands" to the productions and executing the commands when they come to the top of the stack. See the 'Command' pattern in Gamma et al.*
- Can you modify the grammar Exp2 with added commands so that it computes the right result (hint use an extra stack to hold intermediate results).
- Add unary operators to the grammar Exp2 so that you get a LL(1) grammar. Can you parse $12/-2$?

10.6 Dealing with non LL(1) grammars

10.6.0 Converting to LL(1)

In many cases we can convert an non-LL(1) grammar to an LL(1) grammar.

10.6.0.0 Factoring

When two productions for a nonterminal start the same way, the nonterminal will not be LL(1).

Example

$$\begin{aligned} \mathit{Stat} &\longrightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \textbf{end} \\ \mathit{Stat} &\longrightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \textbf{else}\ \mathit{Stat}\ \textbf{end} \end{aligned}$$

The terminal **if** will be in the selector sets of the first to productions.

We can factor out the common left parts to get:

$$\begin{aligned} \mathit{Stat} &\longrightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \mathit{MoreIf} \\ \mathit{Stat} &\longrightarrow \textbf{other} \\ \mathit{MoreIf} &\longrightarrow \textbf{else}\ \mathit{Stat}\ \textbf{end} \\ \mathit{MoreIf} &\longrightarrow \textbf{end} \end{aligned}$$

10.6.0.1 Eliminating left recursion

Consider a sequence of one or more items separated by commas

$$\begin{aligned} \mathit{List} &\longrightarrow \mathit{List}\ \text{ , } \textbf{item} \\ \mathit{List} &\longrightarrow \textbf{item} \end{aligned}$$

We can replace these rules with equivalent productions

$$\begin{aligned} \mathit{List} &\longrightarrow \textbf{item}\ \mathit{List}' \\ \mathit{List}' &\longrightarrow \text{ , } \textbf{item}\ \mathit{List}' \mid \epsilon \end{aligned}$$

In general you can eliminate direct left-recursion by replacing rules

$$A \longrightarrow A\alpha_0 \mid A\alpha_1 \mid \beta_0 \mid \beta_1$$

with rules

$$\begin{aligned} A &\longrightarrow \beta_0 A' \mid \beta_1 A' \\ A' &\longrightarrow \alpha_0 A' \mid \alpha_1 A' \mid \epsilon \end{aligned}$$

There exist methods for eliminating indirect left-recursion, e.g.:

$$A \longrightarrow B\alpha \mid \beta \qquad B \longrightarrow A\gamma \mid \delta$$

10.6.0.2 Is that all there is to it?

The above methods will convert many grammars to LL(1) form.

But not all.

In fact there exist languages for which there exists no LL(1) grammar.

Consider if statements in C/C++/Java/Pascal

$$\begin{aligned} \mathit{Stat} &\longrightarrow \textbf{if}\ \text{ (} E \text{)}\ \mathit{Stat}\ \mathit{MoreIf} \mid \ldots \\ \mathit{MoreIf} &\longrightarrow \textbf{else}\ \mathit{Stat} \mid \epsilon \\ E &\longrightarrow \ldots \end{aligned}$$

(This is not LL(1) since **else** is in Follow(*Stat*))

You can still use recursive decent or top-down predictive parsing, in this case, but you have to use a means other than the selector set to pick the production rule.

For example in the "ambiguous else" example, the parser should pick the first production rule for *MoreIf* when the next terminal is **else**.

10.7 Bottom-up, Shift-Reduce Parsing

We don't need (or use) augmented grammars for this.

10.7.0 State

In this parsing method the state is $\alpha_\wedge t$ where

- α is a stack (top is at *right*) representing consumed input and
- t is the remaining input

We initialize the state to $\epsilon_\wedge w\$$, where w is the original input, where w is the original input and **$** is a symbol not in S

Invariant: $\alpha t \stackrel{*}{\Longrightarrow} w\$$ I.e. there is a derivation from αt to $w\$$.

10.7.1 Steps

There are two kinds of steps

- Shift steps: $\alpha_\wedge a u \quad \vdash_{\text{bu}} \quad \alpha a_\wedge u$
- Reduce steps: $\beta\gamma_\wedge t \quad \vdash_{\text{bu}} \quad \beta A_\wedge t$
  where $A \longrightarrow \gamma$ is a production rule

10.7.2 Stops

- State $A_{\text{start}\wedge}\$$ means success
- If neither a shift nor a reduce can lead to a successful parse, then an error is declared.

10.7.3 Example using grammar Exp1

$$
\begin{aligned}
E &\longrightarrow T \mid E + T \mid E \text{ - } T \\
T &\longrightarrow F \mid T \text{ * } F \mid T \text{ / } F \\
F &\longrightarrow \mathbf{n} \mid (\ E\ )
\end{aligned}
$$

	$\boldsymbol{\alpha}_\wedge t\$$	**Action**
	$\epsilon_\wedge$ **n - n * n - n $**	Shift
$\vdash_{\text{bu}}$	**n** $_\wedge$ **- n * n - n $**	Reduce $F \longrightarrow$ **n**
$\vdash_{\text{bu}}$	F $_\wedge$ **- n * n - n $**	Reduce $T \longrightarrow F$
$\vdash_{\text{bu}}$	T $_\wedge$ **- n * n - n $**	Reduce $E \longrightarrow T$
$\vdash_{\text{bu}}$	E $_\wedge$ **- n * n - n $**	Shift
$\vdash_{\text{bu}}$	E **-** $_\wedge$ **n * n - n $**	Shift
$\vdash_{\text{bu}}$	E **-** **n** $_\wedge$ *** n - n $**	Reduce $F \longrightarrow$ **n**
$\vdash_{\text{bu}}$	E **-** F $_\wedge$ *** n - n $**	Reduce $T \longrightarrow F$
$\vdash_{\text{bu}}$	E **-** T $_\wedge$ *** n - n $**	Shift
$\vdash_{\text{bu}}$	E **-** T ***** $_\wedge$ **n - n $**	Shift
$\vdash_{\text{bu}}$	E **-** T ***** **n** $_\wedge$ **- n $**	Reduce $F \longrightarrow$ **n**
$\vdash_{\text{bu}}$	E **-** T ***** F $_\wedge$ **- n $**	Reduce $T \longrightarrow T$ ***** F
$\vdash_{\text{bu}}$	E **-** T $_\wedge$ **- n $**	Reduce $E \longrightarrow E$ **-** T
$\vdash_{\text{bu}}$	E $_\wedge$ **- n $**	Shift
$\vdash_{\text{bu}}$	E **-** $_\wedge$ **n $**	Shift
$\vdash_{\text{bu}}$	E **-** **n** $_\wedge$ **$**	Reduce $F \longrightarrow$ **n**
$\vdash_{\text{bu}}$	E **-** F $_\wedge$ **$**	Reduce $T \longrightarrow F$
$\vdash_{\text{bu}}$	E **-** T $_\wedge$ **$**	Reduce $E \longrightarrow E$ **-** T
$\vdash_{\text{bu}}$	$E_\wedge$ **$**	

Notice how this traces out a (right-most) derivation in reverse.

10.7.4 LR(1) grammars and Parser Generators

Definition: If you can always pick the correct step on the basis of

- the current stack, and
- the first terminal in the remaining input

then the grammar is said to be LR(1)

Theorem (Knuth): This decision can be made by running a deterministic finite state machine on the stack and then basing the decision on the final state of that machine and the next terminal.

Theorem (Knuth): LR(1) grammars can be parsed in O(N) time

Proof idea: Represent the stack of symbols with a stack of finite machine states (this is a data refinement). Then only the top state on this stack and the next input symbol need to be consulted, not the whole stack.

Theorem: All LL(1) grammars are LR(1).

Why: LL(1) parsers must decide the production rule for A on the basis of the first symbol after the *start* of A. An LR(1) parser must decide the production rule for A on the basis of the first symbol after the *end* of A. Thus an LR(1) parser has at least as much information on which to base a decision.

Implementing an LR(1) parser by hand is not easy for nontrivial grammars.

The "*yacc*" and "*bison*" parser generators use shift-reduce parsing and can handle almost all LR(1) grammars.

yacc and bison produce parsers written in C.

10.7.5 Deterministic shift-reduce parsing

Algorithm: Deterministic shift-reduce parsing

Input: a string w

Output: 'error' or 'success'

var $t := w\$\cdot$ // *where* w *is the input string*
var $\alpha := \epsilon\cdot$ // *Note:* α *behaves as a stack.*
while $\alpha \neq A_{\text{start}} \vee t \neq [\$]$ **do** (
    var $q :=$ decide what to do $\cdot$
    if $q =$ shift **then** (
        $\alpha := (\alpha \hat{} t(0))$;
        $t := \text{tail}(t)$)
    else if $q = \text{reduce}(A \rightarrow \gamma)$ **then**
        let β be such that $\alpha = \beta\gamma\cdot$
        $\alpha := \beta A$
    else /* $q =$ error */
        error)
success

The tricky bit is deciding what to do next: There are three possibilities

- shift
- reduce($A \to \gamma$) where $A \to \gamma$ is a production and γ is the top of the stack.
- error

10.8 Extended BNF (EBNF)

Terminological aside: Context Free Grammars were invented by *Noam Chomsky* in 1957 in the study of natural languages.

John Backus invented an equivalent formalism for describing programming languages.

Peter Naur used Backus's notation in the description of Algol-60.

Thus CFG notation is often called BNF for "Backus-Naur Form"

10.8.0 Back to Extended BNF (or extended CFGs)

We extend CFG notation with convenience notations.

These do not extend range of languages we can describe.

Each production rule now has the form

$$A \longrightarrow x$$

where A is a nonterminal and x is a regular expression over $S \cup V$.

For example we can write Exp3 (equivalent to Exp0, Exp1, Exp2) as

$$E \longrightarrow E; ((\text{`}\mathbf{+}\text{'} \mid \text{`}\mathbf{-}\text{'} \mid \text{`}\mathbf{*}\text{'} \mid \text{`}\mathbf{/}\text{'});\ E)^* \mid \text{`}\mathbf{(}\text{'}; E; \text{`}\mathbf{)}\text{'} \mid \text{`}\mathbf{n}\text{'}$$

Revisiting the C++ Floating Number grammar. We can now be more concise.

$$\begin{aligned} \mathit{floatNum} \longrightarrow\ & \mathit{fract}\ \mathit{exp}^?; (\text{`}\mathbf{f}\text{'} \mid \text{`}\mathbf{l}\text{'} \mid \text{`}\mathbf{F}\text{'} \mid \text{`}\mathbf{L}\text{'})^? \\ & \mid [\text{`}\mathbf{0}\text{'} - \text{`}\mathbf{9}\text{'}]^+; \mathit{exp}; (\text{`}\mathbf{f}\text{'} \mid \text{`}\mathbf{l}\text{'} \mid \text{`}\mathbf{F}\text{'} \mid \text{`}\mathbf{L}\text{'})^? \end{aligned}$$

$$\begin{aligned} \mathit{fract} &\longrightarrow [\text{`}\mathbf{0}\text{'} - \text{`}\mathbf{9}\text{'}]^*; \text{`}.\text{'}; [\text{`}\mathbf{0}\text{'} - \text{`}\mathbf{9}\text{'}]^+ \mid [\text{`}\mathbf{0}\text{'} - \text{`}\mathbf{9}\text{'}]^+; \text{`}.\text{'} \\ \mathit{exp} &\longrightarrow (\text{`}\mathbf{e}\text{'} \mid \text{`}\mathbf{E}\text{'}); (\text{`}+\text{'} \mid \text{`}-\text{'} \mid \epsilon); [\text{`}\mathbf{0}\text{'} - \text{`}\mathbf{9}\text{'}]^+ \end{aligned}$$

10.8.1 EBNF is no more powerful than CFGs

Given an EBNF grammar, we can rewrite its productions to obtain a CFG for the same language:

Algorithm: Apply the following replacements until the grammar is a CFG..

$A \to (P)$	replace with	$A \to P$
$A \to P^*$	replace with	$A \to A'$
		$A' \to \epsilon \mid P\,A'$
$A \to P\,;Q$	replace with	$A \to A1\,A2$
		$A1 \to P$
		$A2 \to Q$
$A \to P \mid Q$	replace with	$A \to A1 \mid A2$
		$A1 \to P$
		$A2 \to Q$

where $A1$ and $A2$ are brand-new nonterminals.

10.8.2 Recursive Descent Parsing with EBNF

We can implement repetition with a while loop and choice with and if.

Consider this augmented grammar for expressions Exp4

$$
\begin{aligned}
E' &\longrightarrow E\,\$ \\
E &\longrightarrow T\ ((\text{`}\mathbf{+}\text{'} \mid \text{`}\mathbf{-}\text{'})\ T)^* \\
T &\longrightarrow F\ ((\text{`}\mathbf{*}\text{'} \mid \text{`}\mathbf{/}\text{'})\ F)^* \\
F &\longrightarrow \text{`}\mathbf{n}\text{'} \mid \text{`}\mathbf{(}\text{'}\ E\ \text{`}\mathbf{)}\text{'}
\end{aligned}
$$

The choices implicit in the repetitions can be made on the basis of the next token since:

- neither + nor - are in the Follow set of E
- neither * nor / are in the Follow set of T

We can implement E with a subroutine

procedure E is

var p:= T ·
while $t(0) \in$ {'+','-'} **do**

    if $t(0)$ = '+' **then** (
        t:= tail(t) ;
        var q := T ·
        p:= $p + q$)
    else (
        t := tail(t) ;
        var q:= T ·
        $p := p - q$)

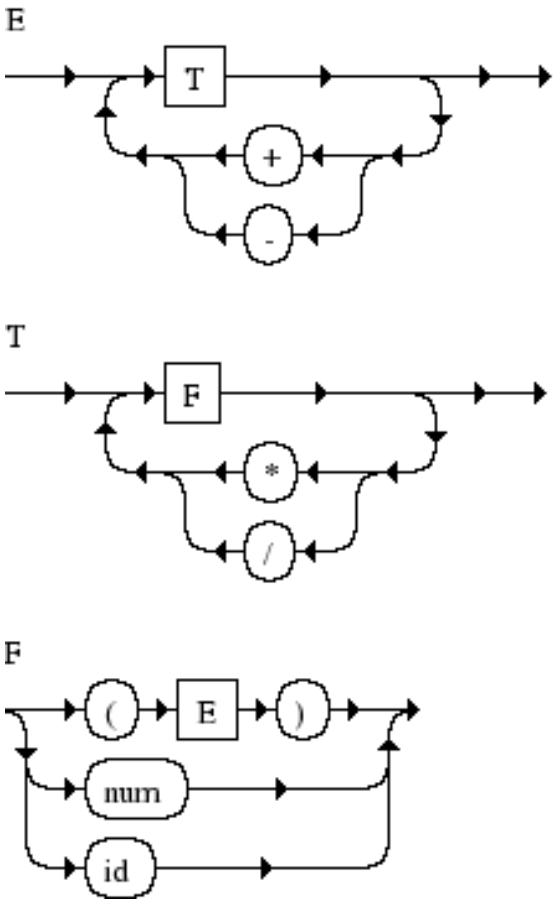
return p

Note that I didn't bother to check that $t(0)$ is '**)**' or **$** prior to returning, since the caller of E will presumably make that check and can provide a better error message.

Parser Generation: The *JavaCC* parser generator accepts Extended BNF grammars and produces parsers written in Java. (https://javacc.dev.java.net/)

10.8.3 Syntax diagrams (or railroad diagrams)

Syntax diagrams are similar to EBNF except, instead of using regular expressions, we use NDFRs. These NDFRs are conventionally drawn with networks of lines representing the states and boxes representing the transitions. For example:



10.9 Regular languages

Parsing with LL(1) and LR(1) grammars takes

- $O(N)$ time and (worst case)
- $O(N)$ space. (N is the length of the input)

Regular languages are those that take $O(1)$ space.

We can define regular languages in terms of grammars one of several equivalent ways

10.9.0 First way

A regular language is one that can be described by an EBNF grammar with no recursion (direct or indirect) between the nonterminals

For example the syntax of floating point numbers in C++.

Example:

$$M \longrightarrow \mathbf{\$}DD^?D^?(\mathbf{,}DDD)^*\mathbf{.}DD$$
$$D \longrightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Counter-example:

$$E \longrightarrow (\mathbf{n}\ ((+ \mid \mathbf{-} \mid \mathbf{*} \mid /)\ \mathbf{n})^*) \mid \text{`}\mathbf{(}\text{'}\ E\ \text{`}\mathbf{)}\text{'}$$

has recursion.

10.9.1 Second way

A production rule is right linear if it is of the form

$$A \longrightarrow sB$$

or of the form

$$A \longrightarrow s$$

(recall that s contains no nonterminal).

A regular language is one that can described by a CFG containing only right-linear productions.

Example

$$A \longrightarrow \mathbf{\$}\ B$$
$$B \longrightarrow \mathbf{d}\,C$$
$$C \longrightarrow D \mid \mathbf{d}\,D$$
$$D \longrightarrow E \mid \mathbf{d}\,E$$
$$E \longrightarrow \mathbf{,d\,d\,d}\,E|\mathbf{.\,d\,d}$$

Counter-example:

$$E \longrightarrow \mathbf{n} \mid \text{`}\mathbf{(}\text{'}\ E\ \text{`}\mathbf{)}\text{'}$$

Recursion of E is not at the very right of the production rule

10.10 Attribute grammars

Attribute grammars augment terminals and nonterminals with attributes.

Each production has an boolean expression that must be satisfied at each node in the derivation tree.

10.11 Recognition by "dynamic programming"

Input: A string w and a grammar $G = (V, S, P, A_{\text{start}})$ such that every production is in one of the following forms

$$A \longrightarrow a$$
$$A \longrightarrow B\ C$$

(Exercise. Show that any grammar such that $\epsilon \notin L(G)$ can be transformed to an equivalent grammar that satisfies this constraint.)

Let n be the length of w.

var m : **array** $\{0, .., n\} \times \{0, ..n\}$ **of** $\mathcal{P}(V)$
// Each element of m is a set of nonterminals
for $(i, j) \in \{0, .., n\} \times \{0, ..n\}$ **do** $m(i, j) := \emptyset$

The idea is to put into each element $m(i, j)$ all nonterminals that describe the substring $w[i, .., j]$ where $i < j$.

for $i \in \{0, ..n\}$ **do** $m(i, i + 1) := \{A \mid A \longrightarrow w(i) \in P\}$

So far we have succeeded for substrings of length 1.

We can 'multiply' two sets of nonterminals U and V as follows: if $B \in U$ and $C \in V$ and $A \longrightarrow B\,C$ is a production then $A \in U \otimes V$. I.e.

$$U \otimes V = \{A, B, C \mid (A \longrightarrow B\,C) \in P \wedge B \in U \wedge C \in V \cdot A\}$$

If s and t are strings and U contains all nonterminals B such that $B \stackrel{*}{\Longrightarrow} s$ and V contains all nonterminals C such that $C \stackrel{*}{\Longrightarrow} t$, then $U \otimes V$ contains all nonterminals A such that $A \stackrel{*}{\Longrightarrow} s\hat{}t$.

The rest of the algogithm fills in the table for segments of increasing length.

for k **from** 2 **to** n **do**

for $i \in \{0, .., n-k\}$ **do**
  let $j := i + k\cdot$
    for $\ell \in \{i+1, .., j-1\}$ **do**
      $m(i,j) := m(i,j) \cup (m(i,\ell) \otimes m(\ell,j))$

Upon completion, the string is in the language iff $A_{\text{start}} \in m(0,n)$.

Part 2

Efficiency

Typeset January 22, 2018

Chapter 11

Computation time

11.0 The RAM model of computation

If we are going to consider how long programs take to run, a natural question is, "On what machine?" We could pick some particular current model of computer. I'm writing this on an HP dv-4040ca, so why not that? But then our results would be very specific. Would they carry over to other current computers or to future computers? In this part of the book, instead of considering the time taken to run programs on particular computers, we will try to draw conclusions about running time that will hold true over all computers — or at least a broad class of computers.

We will consider an idealized computer defined by the following state space.

$$\left\{ \text{“}m\text{”} \mapsto \left(\mathbb{Z} \overset{\text{tot}}{\to} \mathbb{Z} \right), \text{“}c\text{”} \mapsto \mathbb{N}, \text{“}p\text{”} \mapsto \left(\mathbb{N} \overset{\text{tot}}{\to} \mathbb{I} \right), \text{“}A\text{”} \mapsto \mathcal{F}(\mathbb{Z}), \text{“}\mathit{status}\text{”} \mapsto \{\text{running}, \text{stopped}\} \right\}$$

where $\mathbb{I}$ is a set of instructions to be defined shortly. m is the data memory. c is the program counter, p is the program memory, A is the set of allocated memory locations, and s is a status flag used to indicate that the machine is done. The notation $\mathcal{F}(\mathbb{Z})$ means the set of all finite subsets of integers. We will call this computer a Random Access Machine, or RAM for short.[0]

[0] The abbreviation RAM is a bit unfortunate, as the same three letters often stand for "Random Access Memory". In both cases the word "random" doesn't mean random at all, but rather that memory locations can be accessed in an arbitrary order, rather than sequentially, as with a tape.

Compared to the state of a real computer, this is very simple. Real computers use many levels of storage, registers, cache, main memory. Multiple levels exist for reasons of efficiency, we will abstract away from such details and so we don't need to consider multiple levels of memory. Another simplification is that we keep the program separate from the main memory. This is called a Harvard architecture, as opposed to the Princeton architecture, which puts the program in the main memory. A Princeton architecture allows self-modifying code, which we won't need, and is very useful if you are writing an operating system, which we won't be doing.

You should note that m consists of an infinite number of cells. This is one reason that we call this an idealized computer; it never runs out of memory. Designers of real computers attempt to approximate this idealization by having very large physical memories and by using virtual memory. You should also note that each cell of m can hold any integer. This is another idealization. On typical real computers, each cell is limited to some small number of bits, e.g. 8, 16, 32, 48, 64. It is then up to the software designers (and/or compiler writers) to lump together enough adjacent cells to hold whatever number is desired. (And to deal with the consequences when they haven't used enough cells to hold the result of a computation, i.e. to deal with overflow.)

There are a few reasons for considering a computer with an infinite number of cells. The first is that we want to consider the time required by a program to execute as a function of input size; and we want to consider what happens as the input size goes to infinity. If we put any limit on the size of the memory, that will impose a limit on the size of the input and we'll be stuck. The second reason is that the infinite memory model is close to what programming languages try to provide. For example the **new** operator in Java or C++ provides new cells without any *a priori* limit. As programmers, we work with the abstraction of infinite memory, although if we are good programmers, we also plan for the case where the abstraction fails, e.g. when the **new** operator throws an exception because memory has run out. It's a bit like the physicist who works with an infinite and flat 3D space; we know the real universe is finite, curved and expanding, but, most of the time, the abstraction is "good enough" and considerably simpler.

Now given that we have an infinite number of cells, is there a good reason that each cell should be able to hold any integer? Perhaps for the sake of realism, we should make each cell some fixed width w bits. E.g. $w = 8$ is quite common in the real world. This simplification is sometimes made.

Theoretical computer scientists often pick $w = 1$, as any other number is larger than absolutely necessary. However, fixed sized cells make programs more complicated, as we have to deal with grouping cells together to obtain enough space to hold data. Even worse, as the size of the input heads toward infinity, we might have to change the number of cells we lump together. For example, suppose we represent a graph by a list of edges; if $w = 8$ and the number of nodes is less than 257, we can represent each edge with two memory cells. However for graphs with 257 through $65,536$ nodes, we need four cells per edge, and then for larger graphs, more cells. A second reason for not using fixed sized cells is that sometimes we want to store the address of one cell in another cell, since addresses are integers, so are cell contents.

Now we need an instruction set $\mathbb{I}$ for the machine. I propose the following set.

bpos(i, j)	**if** $m(j) > 0$ **then** $c := i$ **else** $c := c + 1$
call(i, j)	$m(j) := c + 1; c := i$
return(i)	$c := m(i)$
halt()	*status* := stopped
add(i, j, k)	$m(i) := m(j) + m(k); c := c + 1$
mult(i, j, k)	$m(i) := m(j) \times m(k); c := c + 1$
negate(i, j)	$m(i) := -m(j); c := c + 1$
odd(i, j)	**if** $m(j)$ is odd **then** $m(i) := 1$ **else** $m(i) := 0$
shift(i, j)	$m(i) := \lfloor m(j)/2 \rfloor ; c := c + 1$
fetch(i, j)	$m(i) := m(m(j)); c := c + 1$
store(i, j)	$m(m(i)) := m(j); c := c + 1$
load(i, j)	$m(i) := j; c := c + 1$

Why this set? Well, I admit that some of the choices are arbitrary, but I am guided by the following aims:

- I want to make sure that the machine is capable of any computing any function that any other computer can compute. This property is called *universality*.

- On the other hand, I want to keep the set of instructions small and simple.

- A program that runs on the RAM should be easily translatable to one that runs on a physical processor.

- Finally I want to ensure that the number of steps taken by the machine to compute a function is similar to the number of steps taken by a typical physical central processor.

The first requirement, universality, turns out to be impossible to prove, at least in a mathematical sense. We can certainly prove that one model of computation is as expressive as another, but it is hard to show that no one will ever devise a more expressive model. Neverthelesss we can say that our RAM is as expressive as any other "realistic" model of computation that has been devised yet.

Thw final requirement involves a few subtleties. For example our RAM can add any two integers in a single step, whereas a typical physical processor, having fixed width registers, would take several steps to, for example, add two 100 digit numbers. On the other hand, most (but not all) physical processors have an instruction that will divide two (fixed width) integers. On our RAM, division requires several instructions. Whether we have satisfied the final requirement depends on what we mean by "similar".

Here is an example that shows that computation times on a RAM can be misleading, if we are not careful. If $m(0)$ is initially n, with $n > 0$, this program computes $2^{(2^n)}$ into $m(1)$ in only $3n + 3$ steps.

$$
\begin{aligned}
&0 : \text{load}(1, 2)\\
&1 : \text{load}(-1, -1)\\
&2 : \text{mult}(1, 1, 1)\\
&3 : \text{add}(0, 0, -1)\\
&4 : \text{bpos}(2, 0)\\
&5 : \text{halt}()
\end{aligned}
$$

For very very small n this is a realistic result. As soon as $n = 5$, we will find that the program will not run on a 32 bit machine without using multiple memory locations to represent $m(1)$. Furthermore, every time n inreases by one the memory requirements of a real computer double and the time requirement will surely at least double. The point of this example is to show that we need to be careful about when to use the RAM model and about interpreting results obtained with the RAM model.

11.1 Parameterized RAM

A $\text{RAM}_{w,s}$ is a model of computation much like a RAM, but whose memory consists of s words that are each w bits wide; here s and w are integers greater than 1 or are infinite. Define

$$\text{words}(w) = \begin{cases} \{-2^{w-1}, ..2^{w-1}\} & \text{if } w \in \mathbb{N} \\ \mathbb{Z} & \text{if } w = \infty \end{cases}$$

$$\text{addresses}(s) = \begin{cases} \{\lceil \frac{-s}{2} \rceil, .. \lceil \frac{s}{2} \rceil\} & \text{if } s \in \mathbb{N} \\ \mathbb{Z} & \text{if } s = \infty \end{cases}$$

The type of the memory for a $\text{RAM}_{w,s}$ is $\text{addresses}(s) \xrightarrow{\text{tot}} \text{words}(w)$. Thus m has type

- $\mathbb{Z} \xrightarrow{\text{tot}} \mathbb{Z}$ if $w = \infty$ and $s = \infty$
- $\mathbb{Z} \xrightarrow{\text{tot}} \{-2^{w-1}, ..2^{w-1}\}$ if $w \in \mathbb{N}$ and $s = \infty$
- $\{\lceil \frac{-s}{2} \rceil, .. \lceil \frac{s}{2} \rceil\} \xrightarrow{\text{tot}} \mathbb{Z}$ if $w = \infty$ and $s \in \mathbb{N}$
- $\{\lceil \frac{-s}{2} \rceil, .. \lceil \frac{s}{2} \rceil\} \xrightarrow{\text{tot}} \{-2^{w-1}, ..2^{w-1}\}$ if $s, w \in \mathbb{N}$.

For example, in a $\text{RAM}_{8,99}$, the address space is $\{-49, ..50\}$ and each memory word holds a value in the range $\{-128, ..128\}$.

We also change the type of *status* to $\{\text{running}, \text{stopped}, \text{crashed}\}$.

We redefine the instructions for the RAM of section 11.0 so that the machine crashes rather than storing into a nonexistent location or storing a value that is too big or too small. For example:

$\text{call}(i, j)$	**if** $j \in \text{addresses}(s) \wedge c + 1 \in \text{words}(w)$ **then** $(m(j) := c + 1; c := i)$ **else** *status* := cras
$\text{add}(i, j, k)$	**if** $i, j, k \in \text{addresses}(s) \wedge m(j) + m(k) \in \text{words}(w)$ **then** $(m(i) := m(j) + m(k); c := c$

Since a $\text{RAM}_{\infty,\infty}$ never crashes, it is essentially the same as the RAM of section 11.0.

Our program for computing $2^{(2^n)}$ still takes linear time, but requires at least a $\text{RAM}_{2^n+1,3}$.

Chapter 12

A Dialogue concerning $P = NP$ and NP-Completeness

Dramatis personae (with apologies to Galileo Galilei)

- Salviatus: A scholar who is well informed and perhaps a bit of a know-it-all.
- Sagredus: A willing and apt pupil.
- Simplicius: Another willing pupil, but one who is perhaps not as quick on the uptake as Sagredus

12.0 First day

SAGREDUS: Salviati, good friend, we have heard that the question of whether $P = NP$ or not is considered by many to be the most important open question in computing science, but I have to confess that I've never really understood it. Perhaps, like much of mathematics, it is simply beyond what I can fathom.

SALVIATUS: I seriously doubt that. The concepts aren't hard. Like the 4-colour theorem, or Fermat's last theorem, the proof that $P = NP$ or that $P \neq NP$, if we ever have one, will no doubt be hard to follow, but the statement of the question is easy to understand.

SIMPLICIUS: Isn't it just that NP means hard problems and P means easy problems, so the $P \stackrel{?}{=} NP$ question just asks whether all hard problems are easy. It seems a rather dumb question.

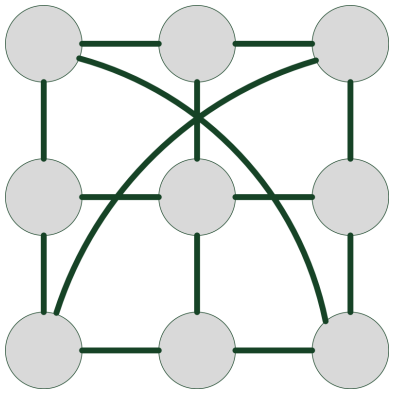


Figure 12.0: Can you colour the nodes of this graph with three colours so that nodes connected by an edges are coloured differently?

SALVI.: No, NP is a set of problems whose affirmative answers can be quickly checked, given some evidence. NP includes both problems that we can solve quickly (easy problems) and problems that we don't know how to solve quickly (apparently hard problems).

SAGR.: Can you give us an example?

SALVI.: If I claim that a number n is composite, this claim can be quickly checked if I back it up with *evidence*. In this case the evidence would be two factors p and q. You can quickly check the evidence by multiplying p by q and comparing the result to n. So the question of whether a number is composite or not is in NP.

SAGR.: Can you give us another example?

SALVI.: Consider the question of whether a graph can be 3-coloured.

SIMPL.: Meaning?

SALVI.: A graph is 3-coloured iff each node of the graph is assigned one of 3 colours such that each edge connects nodes of different colours. [See Figure 12.0] If a graph can be 3-coloured, there is a way to quickly convince someone that it can; here's how: you simply write out a list of nodes and write a colour beside each, red, green, or blue. This list is the evidence. Anyone can quickly check that each edge connects nodes of different colours. [See Figure 12.1]

SIMPL.: You keep using the word 'quickly', but what do you mean by that?

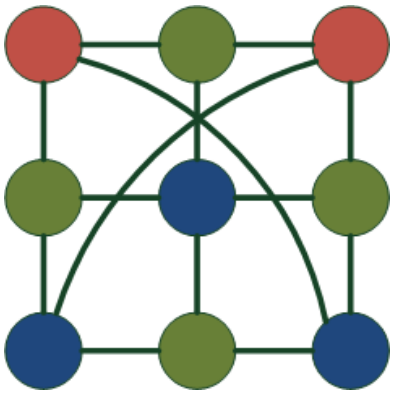


Figure 12.1: Evidence that the graph of Figure 12.0 can be 3-coloured.

SALVI.: That is a good question. I mean *in polynomial time*. That is, the checking problem can be solved in polynomial time with respect to the size of the original input.

SAGR.: So in the example of determining whether a number n is composite, would the size be n?

SALVI.: When an input is an integer, it is usual to use the number of bits required to represent the number; so for this problem, the input size is $\lceil \log_2 n \rceil$.

SAGR.: And for the graph colouring example.

SALVI.: In that case, the number of nodes plus the number of edges will do fine.

SAGR.: What about 'no' answers? Should there also be a quick way of demonstrating that the answer is 'no'?

SALVI.: That is not needed for a problem to be in NP. If there is a quick way of showing that an answer is 'no', we say that the problem is in co-NP. ("co" stands for "complementary".) It is quite possible for a problem to be in both NP and co-NP.

SAGR.: This all makes sense for problems that have 'yes' or 'no' as their answers, that is problems that ask one to determine whether its input is in a particular set or not.

SALVI.: That is true. The set NP contains only *decision problems*, which simply means problems with either 'yes' or 'no' as answer. We do that for two reasons. First it makes it simpler to translate between problems, we only

need to translate the inputs, the outputs need no translation. The second reason is that it allows us to treat the 'yes' outputs differently from the 'no' outputs, which is what we do in the definition of NP.

SAGR.: So NP is the set of decision problems such that evidence for affirmative answers can be checked in time that is polynomial with respect to the size of the original input.

SALVI.: Exactly.

SIMPL.: I had thought that NP problems were all hard, but this definition seems to put no restriction on how easy a problem is, but rather on how hard. So no problem is too easy to be in NP.

SALVI.: That's right. Consider determining whether the shortest path between two points in a graph is less than a given number. This problem can be solved in polynomial time —use Dijkstra's algorithm to calculate the actual size of the shortest path and then simply compare with the given number. We can use the actual shortest path as the evidence. It fits the definition of NP.

SAGR.: You could also use as evidence, the input to the problem, that is the graph, the two nodes, and the length. Then the checker can use Dijkstra's algorithm.

SALVI.: In fact any decision problem that can be solved in polynomial time is in NP since all the evidence that is needed is the input to the problem itself.

SIMPL.: But I thought NP stood for 'nonpolynomial.'

SALVI.: Actually it stands for *nondeterministic polynomial time*. This means that the problem is solvable in polynomial time on a kind of nondeterministic computer. This special kind of computer has all the conventional instructions plus one special instruction called a "nondeterministic branch". Whenever the nondeterministic branch is encountered, the machine takes the branch or doesn't as follows:

- If both choices inevitably lead to a 'no' output, then the choice between branching and not branching is arbitrary.
- If both choices can lead to a 'yes' output, then the choice between branching and not branching is again arbitrary.
- But, if only one choice can possibly lead to a 'yes' output, then that choice is taken.

Here is a nondeterministic algorithm for determining whether a graph can be 3-coloured.

```
read a graph (V, E)
for each node v ∈ V
    paint v red
    nondeterministic branch to A
    repaint v green
    nondeterministic branch to A
    repaint v blue
    A:
for each edge e ∈ E
    let u and v be the endpoints of e
    if u and v are painted the same colour, output 'no' and stop
output 'yes' and stop
```

Please understand that this kind of machine is simply used for thought experiments. It's not something we can really implement.

SIMPL.: You could implement the nondeterministic branch by bracktracking. I read a paper on this by Floyd [[citation needed]].

SALVI.: Yes, but that would take too long. We assume each nondeterministic branch takes constant time, just like a conventional instruction.

SIMPL.: Could you implement a nondeterministic branch with parallelism? Every time a nondeterministic branch is encountered, create a new process. The new process takes the branch and the old one doesn't. Once all processes are done, 'or' the results computed by all the processes.

SALVI.: Yes. Unfortunately this is not efficient, as it takes a lot of hardware or a lot of time. In the 3-colouring example the number of processes needed is 3^N where N is the number of nodes in the input graph. If each process runs on its own processor, this takes too much hardware — that is, more than a polynomial amount of hardware. If we time-share a polynomial number of processors, then it takes too long — that is, more than a polynomial amount of time. In general a nondeterministic polynomial time algorithm can take an exponential amount of time, if we execute it on a conventional processor, and can take either an exponential amount of time or an exponential number of processors if executed on a multiprocessor made from conventional processors.

SIMPL.: So when we talk about 'time' complexity, we really mean work, i.e. the time it takes to do something multiplied by the number of processors we have.

SAGR.: So now we have two definitions of NP. The first is based on quickly checking evidence. The second is that NP problems are decision problems that can be solved with a nondeterministic machine in polynomial time. Let's call these two sets NP_0 and NP_1 respectively. I think I see why these sets are the same.

Suppose a problem is in NP_0. This means there is some evidence for 'yes' answers that can be checked in polynomial time. Then we can write a nondeterministic polynomial time algorithm for it as follows. The first part of the algorithm uses the nondeterministic branch instruction to construct the evidence. Since the size of the evidence must be polynomial in the size of the input, this part will take only polynomial time. The second part uses only conventional instructions to check the evidence in polynomial time. (This is the nature of the program you showed as an example.) The existence of this algorithm pattern shows that the problem is also in NP_1. In conclusion any problem in NP_0 is in NP_1.

Now suppose that a problem is in NP_1. We have a nondeterministic algorithm that runs in polynomial time. For 'yes' answers, a trace of this algorithm will serve as evidence that the answer is 'yes'. In fact the only evidence we need is list of bits telling us whether or not each nondeterministic branch is taken or not. (Say 0 for not taken, and 1 for taken.) In the graph colouring example, 10100... would mean that the first three nodes are to be painted red, green, and blue respectively. We can check the evidence by simulating the execution of the algorithm, consulting the evidence each time we need to know whether a branch is to be taken or not. Since the nondeterministic algorithm takes polynomial time, so does the checking procedure. In conclusion any problem in NP_1 is also in NP_0. We now know that both definitions for NP define the same set.

SALVI.: Well done my friend.

SAGR.: I'd read that NP is often defined using something called a Turing Machine.

SALVI.: Yes. Often the definition uses a mathematical model for computers called Turing Machines. Turing Machines are rather different from conventional computers, but for our purposes they amount to the same thing. The reason is that, while Turing Machines generally take more time to do the same thing, compared to a conventional computer, they are only slower by a

factor that is polynomial in the size of the input. So when we ask whether an algorithm takes polynomial time, it doesn't matter whether we plan to execute it on a conventional computer or a Turing Machine.

SAGR.: Getting back to $P \stackrel{?}{=} NP$, we now know what NP is, so what is P?

SALVI.: P is thankfully much simpler. P is the set of decision problems that can be solved in polynomial time on a regular computer.

SIMPL.: And since the same algorithm can be run on a nondeterministic computer, we have that $P \subseteq NP$.

SALVI.: Exactly.

SIMPL.: So $P \stackrel{?}{=} NP$ asks whether these fictitious nondeterministic branch instructions would really be that helpful. If $P \subset NP$, there is at least one problem that can be solved quickly on the nondeterministic machine, but not on a conventional machine. On the other hand if $P = NP$, every problem in NP can be solved quickly on a conventional computer.

SALVI.: Quite right.

SAGR.: Well it's a nice theory, but why is it important.

SALVI.: A lot of practical problems are known to be in NP but are not known to be in P — we don't have fast algorithms for them. If $P = NP$,then all these problems will have polynomial time algorithms. Furthermore, if $P = NP$, that would imply that a number of problems that are not decision problems also have polynomial time solutions. Not so obvious is the dual to that. If $P \subset NP$, then this implies there are many problems (both decision and optimization) that can not be solved quickly. Knowing whether $P = NP$ or not would decide the difficulty of a large number of really practical problems that come up every day.

SIMPL.: How can that be? I can see that if $P \subset NP$, there is at least one problem in NP that is difficult, but not that there are many. Perhaps $P \subset NP$ but there is only one problem in NP that is difficult and that that is a problem that no one happens to care about. Isn't that possible?

SALVI.: As I said, it is not so obvious that this is not the case. Let us leave this question for another day.

12.1 Second day

SIMPL.: Yesterday you told us that if $P \subset NP$, there is more than one problem in NP that can not be solved quickly.

SALVI.: Yes, that's right. The key to this is to show that certain problems are among the most difficult in NP.

SAGR.: Most difficult in what sense, since it seems we don't know how difficult the most difficult problems in NP are?

SALVI.: Suppose we have a problem Q in NP and we know that if any problem in NP is difficult then Q is difficult. ...

SIMPL.: And by difficult, you mean that there is no polynomial time algorithm for it.

SALVI.: Yes, exactly. So in that case, we can say that Q is one of the *most difficult* problems in NP. Any problem that is among the most difficult in NP in this sense is called NP-complete.

SIMPL.: So how can you show that a problem is NP-complete?

SALVI.: Well suppose that Q and R are two problems in NP and that we have a polynomial-time algorithm f that implements a function F such that $Q(F(x)) = R(x)$, for all x. That is, f translates each input to R into an input to Q. We say f is a polynomial-time reduction from R to Q. Now if Q is in P, then so is R since we can use a polynomial time algorithm for Q, in combination with our polynomial time algorithm for F, to get a polynomial time algorithm for R. If Q is easy, so is R. Equivalently: if R is difficult, so is Q.

SIMPL.: Can you give an example?

SALVI.: Sure. Let's take PSAT and 3-Colouring and find a reduction from 3-Colouing to PSAT, which is the problem of determining whether a propositional formula is satisfiable. ...

SIMPL.: Refresh my memory ...

SALVI.: A propositional formula is just a formula that uses ands, ors, nots, xors, equivalences, parentheses, and variables. A formula ϕ is satisfiable if there is a way to set each variable to true or false so that the whole formula is true.

SAGR.: So that there is a line in its truth table where it's true.

SALVI.: Exactly.

SAGR.: I can see that this could be useful in checking whether an acyclic digital circuit meets its specification. Suppose ψ is the formula for a circuit with inputs u and outputs v and ϕ is the specification for that circuit with inputs x and outputs y, then we could check a formula

$$(x_0 = u_0) \wedge (x_1 = u_1) \wedge \cdots \wedge \psi \wedge \phi \wedge ((v_0 \neq y_0) \vee (v_1 \neq y_1) \vee \cdots)$$

If it's not satisfiable then, the circuit and its specification are equivalent.

SALVI.: Just so. And that's just one application, there are plenty of others. You can see that PSAT is a useful problem on its own. Now we can see that if 3-Colouring is hard, then so is PSAT ...

SIMPL.: So we need to transform each graph that might be input into 3-Colouring into a formula so that the graph has a 3-Colouring exactly if the formula can be satisfied.

SAGR.: And the formula must be produced in polynomial time with respect to the size of the graph.

SALVI.: Exactly. Given any graph $G = (V, E)$ we can produce a formula with $3 \times |V|$ variables. For each node $u \in V$ we have variables R_u, G_u, and B_u. The idea is that $R_u = \mathtt{true}$ represents vertex v being coloured red. We need to say that each vertex is given one and only one colour, so for each vertex u we need

$$(R_u \vee G_u \vee B_u) \wedge \neg (R_u \wedge G_u) \wedge \neg (G_u \wedge B_u) \wedge \neg (B_u \wedge R_u)$$

and we conjoin (and-together) the formulas for all the vertices. Then for each edge $\{u, v\} \in E$ we need a conjunct

$$\neg(R_u \wedge R_v) \wedge \neg(G_u \wedge G_v) \wedge \neg(B_u \wedge B_v).$$

For example for a completely connected graph with three nodes $(\{0, 1, 2\}, \{\{0, 1\}, \{1, 2\}, \{0, 2\}\})$ we would have

$$\begin{aligned}
&(R_0 \vee G_0 \vee B_0) \wedge \neg (R_0 \wedge G_0) \wedge \neg (G_0 \wedge B_0) \wedge \neg (B_0 \wedge R_0) \\
\wedge\, &(R_1 \vee G_1 \vee B_1) \wedge \neg (R_1 \wedge G_1) \wedge \neg (G_1 \wedge B_1) \wedge \neg (B_1 \wedge R_1) \\
\wedge\, &(R_2 \vee G_2 \vee B_2) \wedge \neg (R_2 \wedge G_2) \wedge \neg (G_2 \wedge B_2) \wedge \neg (B_2 \wedge R_2) \\
\wedge &\neg(R_0 \wedge R_1) \wedge \neg(G_0 \wedge G_1) \wedge \neg(B_0 \wedge B_1) \\
\wedge &\neg(R_1 \wedge R_2) \wedge \neg(G_1 \wedge G_2) \wedge \neg(B_1 \wedge B_2) \\
\wedge &\neg(R_2 \wedge R_0) \wedge \neg(G_2 \wedge G_0) \wedge \neg(B_2 \wedge B_0)
\end{aligned}$$

This algorithm for producing formulas from graphs is a polynomial-time reduction from 3-Colouring to PSAT.

$$\text{3COL} \longrightarrow \text{PSAT}$$

Now if we have a quick (polynomial time) algorithm for PSAT, we have a quick algorithm for 3-Colouring. So that shows that if 3-coloring is hard, so is PSAT.

SAGR.: So back to NP-completeness, to show that a problem Q is NP-complete, we need to show that for every problem R in NP, there is a polynomial-time reduction from R to Q.

SIMPL.: This seems a difficult thing to prove. Are there any problems known to be NP-complete?

SALVI.: Yes there are. In fact there are many. It turns out that once you know one problem is NP-complete, you can build on that knowledge to show that other problems are NP-complete without having to worry about reductions from every problem in NP.

SAGR.: I think I see why. Suppose we know that S is NP-complete and we are wondering whether a problem Q, that is known to be in NP, is also NP-complete. Since S is NP-complete, we know that , for any problem R in NP, there is a polynomial-time reduction f from R to S. Suppose we can find a polynomial time reduction g from S to Q; then, for every problem R in NP, there is also a polynomial-time reduction from R to Q formed by composing f with g; thus Q is NP-complete.

$$R \xrightarrow{f} S \xrightarrow{g} Q \text{ and so } R \xrightarrow{f;g} S$$

SIMPL.: I see, because executing one polynomial-time algorithm and then another is still polynomial-time.

SALVI.: Exactly. And the more problems we learn to be NP-complete, the easier it becomes to show that yet others are as well, because we then have more choices of problems to reduce from, that is a wider choice for Sagredus's S.

SAGR.: So if you think of NP-completeness as a disease, then a polynomial time reduction from S to Q allows the disease to spread. If S is NP-complete, then Q must be too.

SIMPL.: OK. But don't you need a patient zero? You need to start by showing that some particular problem is NP-Complete, and that means you have to find an infinite number of reductions, one for each of the other problems in NP. That seems an impossible task.

SALVI.: Nevertheless, that's exactly what was done by Stephen Cook in 1971. He found that any problem in NP can be reduced to PSAT.

SIMPL.: and how did he do that?

SALVI.: Suppose you have a problem R in NP. We need to be able to transform any input x to R represented as a sequence of bits $[x_0, x_1, \cdots, x_{n-1}]$, to a formula ϕ_x that is satisfiable exactly if $R(x)$.

Since R is in NP, there is a nondeterministic program r for R that runs in polynomial time, say no more than $p(n)$ steps on a particular computer, for each input of size n, where p is some polynomial function.

SAGR.: A computer with a nondeterministic branch instruction.

SALVI.: Right, a computer with a nondeterministic branch instruction. A deterministic program defines a function that maps the state of the computer in one time cycle to the state of the computer in the next time cycle. (You can arrange that when the algorithm is done, the function acts as the identity function.) A nondeterministic program defines a relation that maps each state of the computer to one or more states in the next time cycle. Since r implements a decision algorithm, we can assume that one particular bit of memory is used to represent the output, say bit o. For each input size n we can workout the maximum number of additional bits of memory the program needs, say $q(n)$. Of course $q(n)$ must be polynomial, since the algorithm only has a polynomial amount of time to read or write the bits. We'll call these bits b_0, b_1, ... , $b_{q(n)-1}$. We can also assume that some of these bits initially hold the input at the start of the computation, say bits b_0 to b_{n-1}. Now for each input x of size n, you can create a formula ϕ as follows: You use a bunch of boolean variables to represent the state of the memory and registers at time 0 (say o_0, $b_{0,0}$ $b_{1,0}$, and so on up to $b_{q(n)-1}$) and a bunch of other variables to represent the state of the memory at time 1 (say o_1, $b_{0,1}$ $b_{1,1}$, and so on up to $b_{q(n)-1,1}$); and you can use a formula ψ_0 to represent the relation between these two states defined by r. Then you use a bunch more boolean variables to represent the state at cycle 2 and make a formula ψ_1 to represent the relation between the state in cycle 1 and the state in cycle 2. And so on for all $p(n)$ steps. Finally you conjoin ('and' together) all these one-step formulas

$$\psi = \psi_0 \wedge \psi_1 \wedge \cdots \wedge \psi_{p(n)-1}$$

We can throw in a formula that forces the variable representing the output bit at cycle $p(n)$ to be true and formulas that force the states of the bits representing the input to represent x in cycle 0. Now we have a big formula

$$\phi_x = \psi \wedge o_{p(n)} \wedge (b_{0,0} = x_0) \wedge \cdots \wedge (b_{n-1,0} = x_{n-1})$$

Now if $R(x)$ is true, then there is a computation of r that starts with the input bits set to $[x_0, x_1, \cdots, x_{n-1}]$ and ends with the output bit being true after $p(n)$ cycles, thus there is a way to set all the propositional variables of ϕ that satisfies ϕ. On the other hand, if $R(x)$ is false, there can be no

computation of the program r with input x and the output being 'yes'; and so, in this case, ϕ is not satisfiable.

SIMPL.: That's a big formula.

SALVI.: Yes, but since the algorithm takes a polynomial amount of time, it can't use more than a polynomial number of bits, so the whole computation can be represented by a polynomial number of boolean variables. The size of the whole formula is still only a polynomial function of n and the time taken to produce it is also polynomial.

SAGR.: So for any problem R in NP, we can make a transformation, based on a nondeterministic program r for R and a polynomial time bound p, that turns any input x to R into a formula that is satisfiable exactly if $R(x)$. Furthermore, it only takes polynomial time to do so. Let's call this transformation $\text{compile}_{r,p}$. Now if the satisfiability problem PSAT can be solved in polynomial time, by a regular algorithm sat, we can solve any problem R in NP in polynomial time by first running $\text{compile}_{r,p}$ on the input and then running sat on the resulting formula.

SALVI.: Exactly.

SAGR.: So if PSAT can be solved in polynomial time, all problems in NP can be solved in polynomial time, meaning $P = NP$.

SIMPL.: And if PSAT can't be solved in polynomial time, it is not in P, but still in NP and so $P \neq NP$.

So the question of whether $P = NP$ or not is the same as the question of whether PSAT can be solved in polynomial time or not.

SALVI.: Exactly.

SIMPL.: But that's still just one problem.

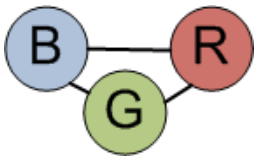
SALVI.: Lots of other problems are in the same boat. There are lots of NP-complete problems. For example the 3-colouring problem that we talked about yesterday. It turns out that any propositional formula ϕ can be translated into a graph so that the formula is satisfiable exactly if the the graph can be 3-coloured. Furthermore the graph is polynomial in size with respect to the size of ϕ and takes only a polynomial amount of time to compute. So for all problems R in NP we have:

$$R \longrightarrow \text{PSAT} \longrightarrow \text{3COL}$$

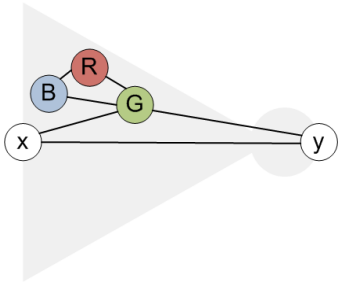
so a quick solution for 3COL implies a quick solution for all problems in NP, i.e., 3COL is NP-complete.

SIMPL.: Can you show us how.

SALVI.: Sure. Start by rewriting ϕ as an equivalent formula ψ, that uses only conjunction (and) and negation (not). Now draw a digital circuit equivalent to ψ. The circuit is a tree consisting of and gates, not gates, and wires. Now we make a graph as follows. Start with three nodes connected with three edges like this.

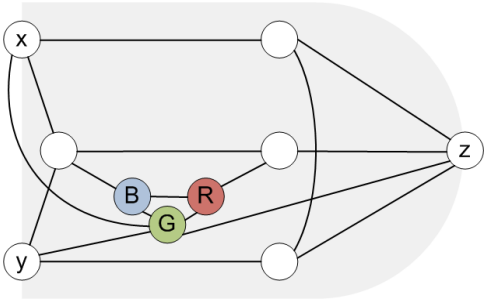


We'll call these nodes R, G, and B. Obviously in any colouring they have to be coloured differently from each other. We can assume that in any colouring B is coloured blue, R is coloured red and G is coloured green. Now for each 'not' gate in the circuit, add two more nodes and three more edges like this.



SIMPL.: So, Y is coloured blue if X is coloured red, and Y is coloured red if X is coloured blue, and neither can be coloured green.

SALVI.: Exactly. So, if we represent "true" with colour blue and "false" with colour red, it represents a 'not' gate. Next, for each 'and' gate, add 7 nodes and 14 edges as follows.



SAGR.: Clearly X, Y, and Z can only be coloured red or blue; Since you are using it to represent and 'and'-gate, and said that red represents false while blue represents true, I'd guess

- that, when X and Y are both coloured blue, Z can and must be coloured blue; and
- that, when one or both of X and Y are coloured red, Z can and must be coloured blue.

SALVI.: Try it your self.

[At this point Sagredus and Simplicius start making copies of the graph and colouring the graph, and you, dear Reader, should do the same; try colouring X and Y with all 4 combinations of colours red and blue and see what colour options are left for Z. Do they agree with Sagredus's guess?]

SIMPL.: And to represent wires from the output of one gate to an input to another, we just need to merge the two nodes in the sense that all edges to one are redirected to the other and then the first node can be deleted.

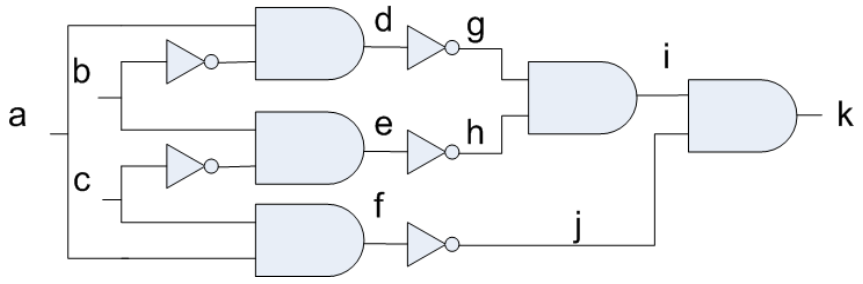
SALVI.: Right. And if two ports represent the same input variable we merge them as well. Finally we merge the output of the root gate with the B node. Here is an example. Consider a ϕ which is

$$(a \Rightarrow b) \wedge (b \Rightarrow c) \wedge (c \Rightarrow \neg a)$$

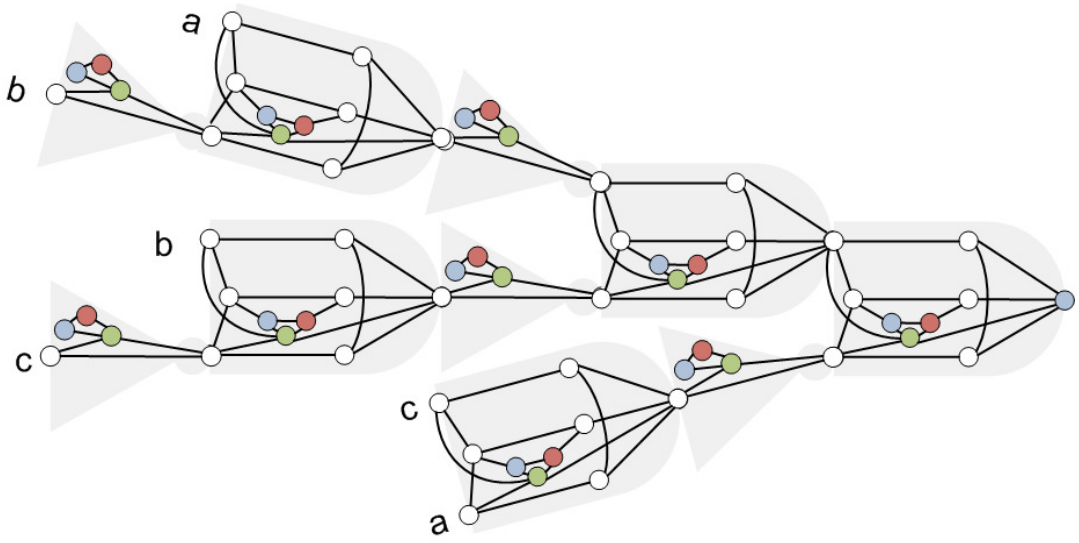
we get for ψ

$$\neg(a \wedge \neg b) \wedge \neg(b \wedge \neg c) \wedge \neg(c \wedge a)$$

The circuit is this



and a drawing of the final graph is this



In this drawing of the graph, the 'two nodes' labelled a are in fact two drawings of the same node and likewise for the nodes labeled b and c; furthermore nodes R, G, and B have been drawn repeatedly as have several edges —e.g., the edge between a and G is drawn twice and the edge between B and G is drawn 11 times.

Now for any ϕ, the corresponding graph can be three coloured if and only if ϕ can be satisfied. (i) Suppose that ϕ can be satisfied. Then there is an assignment to its variables that make it true and thus a way to set the inputs to the circuit to make it result in true. Start by colouring nodes R, G, and B red, green, and blue; For each node representing a variable, colour it blue if the corresponding variable is true in the assignment and red if the corresponding variable is false in the assignment. Now you can colour each 'gate' in the graph and of course the final root output port will be have to be coloured blue, which it is, so every thing works out. (ii) Now suppose the graph can be three coloured; then there must be a three colouring in which R is red, G is green, and B is blue; starting with such a colouring, we can construct an assignment that satisfies ϕ: For each variable node, if it is coloured blue, then the corresponding variable is true in the assignment, and if it is coloured red then the corresponding variable is false in the assignment.

SAGR.: And that means that any problem Q in NP can be reduced to 3-colouring. Suppose we have an input x to Q and want to know whether $Q(x)$ is true. Start with nondeterministic polynomial time algorithm q for Q. Using Cook's method, turn q and x into a formula ϕ and then turn ϕ

into a graph. Note all this takes polynomial time and produces a graph that is polynomial in size with respect to the size of x. Now $Q(x)$ is true iff the graph can be three coloured. If we had a fast way to three-colour graphs, we would have a fast way to calculate $Q(x)$.

SIMPL.: And that means that the question of whether or not $P = NP$ is also equivalent to the question of whether 3-colouring can be done in polynomial time or not.

SALVI.: Now you're getting it; and there are many other problems that can also be seen to have efficient solutions if and only if $P = NP$.

Part 3

Reference

Typeset January 22, 2018

Appendix A

Mathematical Background

This appendix summarizes the mathematical notations and term used in the rest of the book. Most of the notations and terms presented here are standard. Where there is a single common notation or term, I've used that. In some cases there are several common notations; this is especially true in the case of propositional and predicate logic; in these cases, I've picked one. In a few cases there are no common notations; an example is sets of contiguous integers which I've written as $\{i, ..j\}$. Certain terms such as 'domain' and 'range' are used with various meanings by various people. I've picked one meaning and used other terms for other meanings, in this case 'source' and 'target'.

A.0 Sets

A.0.0 Sets, elements, equality and subsets

set

A *set* is a collection of mathematical objects. If S is a set and x is a mathematical object, then x is either an element of the set S or it is not. We write $x \in S$ and say 'x is an element of S', 'x is a member of S', or 'S contains x'.

We write $x \notin S$ and say 'x is not an element of S', 'x is not a member of S', or 'S does not contain x'.

$\emptyset$

Some sets are:

- $\emptyset$ is the *empty set*. It contains no objects.[0]

[0] Although the symbol is similar, it should not be confused with the number 0.

- $\{1\}$ is the set containing only the number 1.
- $\{2, 3, 5, 7\}$ is the set containing only the numbers 2, 3, 5, and 7.

Sets are themselves mathematical objects and so we can have sets that contain sets.

- $\{\emptyset, \{1\}, \{1, 2\}\}$ a set containing 3 objects, each of which is a set.

All the sets we've seen so far are *finite* sets meaning that they each contain a finite number of elements. Other sets are *infinite* meaning that they contain an infinite number of elements.

Some infinite sets are:

- $\mathbb{N}$ the set of all natural numbers: 0, 1, 2, 3, etc.
- $\mathbb{Z}$ the set of all integers: 0, -1, 1, -2, 2, -3, 3, etc.
- $\mathbb{Q}$ the set of all rational numbers.
- $\mathbb{R}$ the set of all real numbers.

=

Two sets are considered equal ($S = T$) exactly if they contain exactly the same objects. So for example:

- $\emptyset = \emptyset$
- $\{1, 2, 3\} = \{2, 3, 1\}$
- $\{1, 1, 2\} = \{1, 2, 2\}$
- $\{\emptyset\} = \{\emptyset\}$

Two sets are considered unequal ($S \neq T$) exactly if one contains an object that the other does not. For example:

- $\{0, 1\} \neq \{0\}$. The first contains 1 and the second does not.
- $\emptyset \neq \{\emptyset\}$. The first does not contain $\emptyset$ and the second does.

⊆

A set S is considered a *subset* of a set T exactly if every element of S is an element of T. We write $S \subseteq T$. For example:

- $\{2, 3, 5, 7\} \subseteq \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\emptyset \subseteq S$ for any set S.
- $\mathbb{N} \subseteq \mathbb{Z}$
- $\mathbb{Z} \subseteq \mathbb{R}$
- $S \subseteq S$ for any set S

A.0.1 Operations on sets

$\cup$

If we have two sets S and T then their *union* (written $S \cup T$) is the set containing all objects contained in either S or in T. Thus:

- $\{1, 3, 5\} \cup \{2, 3, 5\} = \{1, 2, 3, 5\}$,
- $\emptyset \cup S = S$ for any set S,
- $S \cup S = S$ for any set S, and
- $S \cup T = T$ exactly if $S \subseteq T$, for any sets S and T.

$\cap$

The *intersection* of two sets S and T (written $S \cap T$) is the set containing all objects contained in both S and in T. Thus

- $\{0, 2, 4, 6, 8, 10, 12\} \cap \{2, 3, 5, 7, 11\} = \{2\}$,
- $\emptyset \cap S = \emptyset$ for any set S.
- $S \cap S = S$ for any set S, and
- $S \cap T = S$ exactly if $S \subseteq T$, for any sets S and T.

$-$

The *difference* of sets S and T (written $S - T$) is the subset of S containing only objects that are not in T. Thus

- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} - \{2, 3, 5, 7, 11\} = \{0, 1, 4, 6, 8, 9\}$
- $S - \emptyset = S$ for any set S
- $S - S = \emptyset$ for any set S

- $(S - T) \cup (S \cap T) = S$ for any sets S and T
- $(S - T) \cap T = \emptyset$ for any sets S and T.

We can also take the union or intersection of a set of sets. If S is a set and $\mathcal{E}$ is a set valued expression,

$$\bigcup x \in S \cdot \mathcal{E}$$

is the set of objects in any set $\mathcal{E}$ obtained by substituting an element of S for the variable x in the expression $\mathcal{E}$. For example

- $\bigcup x \in \mathbb{N} \cdot \{3x + 1, 3x + 2\}$ is the set of natural numbers not divisible by 3.
- For any particular p, $\bigcup q \in \mathbb{N} \cdot \{p \times q\}$ is the set of all multiples of p: $\{0, p, 2p, 3p, \cdots\}$
- And so $\bigcup p \in \mathbb{N} \cdot \bigcup q \in \mathbb{N} \cdot \{(p + 2) \times (q + 2)\}$ is the set of all positive composite numbers $\{4, 6, 8, 9, 10, 12, \cdots\}$.

The power set of a set S is the set of all subsets of S. We write $\mathcal{P}(S)$ for the power set of S. So for example

- $\mathcal{P}(\{1, 2, 3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
- $T \in \mathcal{P}(S)$ exactly if $T \subseteq S$, for any set S and object T.

A.0.2 Set builder notation

set builder notation

A very useful notation for sets is *set builder notation*.

A.0.2.0 Filtering

If $\mathcal{V}$ is a variable, S is a set and $\mathcal{A}$ is some boolean expression describing variable $\mathcal{V}$ then

$$\{\mathcal{V} \in S \mid \mathcal{A}\}$$

represents the subset of S that contains exactly elements that fit the description $\mathcal{A}$. For example:

- $\{x \in \mathbb{R} \mid x > 0\}$ is the set of positive real numbers
- $\{y \in \mathbb{N} \mid y/3 \in \mathbb{N}\}$ is the set of natural number that are multiples of 3.

The boolean expression acts as a filter. You can think of S as a light source and $\mathcal{A}$ as some sort of filter that casts a shadow.

We can pronounce $\{\mathcal{V} \in S \mid \mathcal{A}\}$ as "the set of all $\mathcal{V}$ in S such that $\mathcal{A}$".

A.0.2.1 Mapping

We write

$$\{\mathcal{V} \in S \cdot \mathcal{E}\}$$

where $\mathcal{V}$ is a variable, S is a set, and $\mathcal{E}$ is an expression, for the set of all values of $\mathcal{E}$ where $\mathcal{V}$ is replaced by a member of S. For example

- $\{n \in \mathbb{Z} \cdot n^2\}$ is the set of all squares of integers
- $\{n \in \mathbb{N} \cdot 2n\}$ is the set of even natural numbers
- $\{m \in \mathbb{N} \cdot \{n \in \mathbb{N} \mid n < m\}\}$ is the set of all downward closed sets of natural numbers, i.e.

$$\{\emptyset, \{0\}, \{0, 1\}, \{0, 1, 2\}, \cdots\}$$

You can think of S as a source of light and $\mathcal{E}$ as a lens that alters the image projected from S.

We can pronounce $\{\mathcal{V} \in S \cdot \mathcal{E}\}$ as "the set of all $\mathcal{E}$, where $\mathcal{V}$ is in S" or "the set over $\mathcal{V}$ in S of $\mathcal{E}$"

When we use this mapping notation, there is no need to restrict ourselves to one variable and one set. For example

$$\{i \in \mathbb{R}, j \in \mathbb{R} \cdot f(i, j)\}$$

gives the set of all values that a function f results in. We might also write this as

$$\{i, j \in \mathbb{R} \cdot f(i, j)\}$$

A.0.2.2 Filtering and Mapping Combined

The full set builder notation combines filtering with mapping. First we filter, then we map. The set

$$\{\mathcal{V} \in S \mid \mathcal{A} \cdot \mathcal{E}\}$$

is the set of all values of $\mathcal{E}$ where $\mathcal{V}$ is replaced by value of S that $\mathcal{A}$ describes. For example

- $\{n \in \mathbb{N} \mid n \text{ is prime} \cdot n^2\}$ is the set of all squares of primes $\{4, 9, 25, 49, \cdots\}$
- $\{n, m \in \mathbb{N} \mid \text{neither } n \text{ nor } m \text{ equals } 1 \cdot m \times n\}$ is the set of composite (natural) numbers, i.e. natural numbers that aren't prime and aren't 1.

When there is only one variable, the expression $\{\mathcal{V} \in S \mid \mathcal{A} \cdot \mathcal{E}\}$ can be regarded as an abbreviation for

$$\{\mathcal{V} \in T \cdot \mathcal{E}\}, \text{ where } T = \{\mathcal{V} \in S \mid \mathcal{A}\}$$

On the other hand, we can regard the filtering and mapping notation as abbreviations for the full set builder notation:

$$\{x \in S \mid \mathcal{A}\} \text{ abbreviates } \{x \in S \mid \mathcal{A} \cdot x\}$$

$$\{x \in S \cdot \mathcal{E}\} \text{ abbreviates } \{x \in S \mid \mathtt{true} \cdot \mathcal{E}\}$$

We can pronounce $\{\mathcal{V} \in S \mid \mathcal{A} \cdot \mathcal{E}\}$ as "the set, over all $\mathcal{V}$ in S, such that $\mathcal{A}$, of $\mathcal{E}$". For example

$$\{m, n \in \mathbb{N} \mid m \text{ and } n \text{ are both prime} \cdot m + n\}$$

is "the set, over all natural numbers m and n, such that m and n are both prime, of $m + n$", i.e. the set of all numbers that the sum of two primes.[1]

[1] The mapping notation and the combined notation are not widely used. Most authors would write $\{2x \mid x \in \mathbb{N}\}$ rather than $\{x \in \mathbb{N} \cdot 2x\}$ for the set of even natural numbers. I prefer the notation presented in this book, as it makes the scope of each variable clear. Furthermore, the mapping and filtering notations introduced here will be echoed in other notations that introduce local variables.

A.0.3 Minimum and maximum

We can pick the maximum or minimum values from a set of real numbers using the max and min functions. These functions won't be defined when the set is empty.[2] For infinite sets they might also not be defined. For example $\max(\mathbb{N})$ is not defined, although $\min(\mathbb{N}) = 0$. Set builder notation is very useful for describing maxima and minima. The idea is that we first build a nonempty set of numbers and then take the maximum or minimum value from the set. For example suppose that I is some nonempty set, and that f is a function from I to $\mathbb{R}$. There are several distinct questions about minima relating to a function like f that we might ask:

- What is the smallest value of f?
- What is the smallest value i in I such that $f(i)$ has some particular property?
- What are the values of I where f has the smallest value.

The answer to the first question —what is the smallest value of f?— is answered by

$$\min\{i \in I \cdot f(i)\}$$

It is a number in $\mathbb{R}$. Often this is written as

$$\min_{i \in I} f(i)$$

On the other hand,

$$\min\{i \in I \mid f(i) > 0\}$$

is the smallest value of i such that the function is positive; it is a member of I. Note that

$$\min\{i \in I \mid f(i) > 0\}$$

is undefined, if there is no i in I such that $f(i) > 0$. You might see such an expression written also as

$$\min_{i \in I | f(i) > 0} i$$

[2] For some applications, it may be useful to define $\min(\emptyset) = \infty$ and $\max(\emptyset) = -\infty$, where ∞ and $-\infty$ are objects such that $\infty > x$ and $-\infty < x$, for all $x \in \mathbb{R}$.

Sometimes we want to denote those arguments that give a function its minimum value. Of course there could be more than one such argument, so we have a set

$$\left\{ i \in I \mid f(i) = \min_{i \in I} f(i) \right\}$$

This expression is sometimes abbreviated as

$$\operatorname{argmin}_{i \in I} f(i)$$

and of course there is an analogous argmax. Note that $\operatorname{argmin}_{i \in I} f(i)$ (and $\operatorname{argmax}_{i \in I} f(i)$) is, in general, a set, although, if we know a priori that there is only one member, we might use these notations to denote the sole member of the set.

If f counts the number of times something occurs in a population I, $\operatorname{argmax}_{i \in I} f(i)$ is called the set of *modes* of f.

A.0.4 Sets of consecutive integers

Sets of integers. We use special notations for finite sets of consecutive integers

- $\{i, .., k\} = \{j \in \mathbb{Z} \mid i \leq j \leq k\}$
- $\{i, ..k\} = \{j \in \mathbb{Z} \mid i \leq j < k\}$

Note that $\{i, ..i\} = \emptyset$, while $\{i, .., i\} = \{i\}$.

The set of the first n natural numbers is $\{0, ..n\}$. The set of the first n positive integers is $\{1, .., n\}$.

A.0.5 Pairs, other tuples, and Cartesian products

pair

A *pair* of objects x and y is written (x, y). A pair is an ordered collection of objects so, for example, the pair $(1, 2)$ is not the same as the pair $(2, 1)$. Similarly we have triples (x, y, z), quadruples (w, x, y, z), quintuples (v, w, x, y, z), and in general n-tuples for any $n \geq 2$. In general we can use the word *tuple* for all these.

$\longmapsto$

An alternative notation for a pair is $x \mapsto y$. This is often used when the pair helps define a function or binary relation as discussed in Section A.1.

The Cartesian product of two sets S and T is the set of all pairs (x, y) such that $x \in S$ and $y \in T$. For example:

- $\{0,1,2\} \times \{3,5\} = \{(0,3),(0,5),(1,3),(1,5),(2,3),(2,5)\}$

We can extend the set builder notations to allow multiple variables when forming a set of pairs. For example

- $$\begin{aligned}&\{(x,y) \in \mathbb{N} \times \mathbb{N} \mid x < y\}\\ &= \{x, y \in \mathbb{N} \mid x < y \cdot (x,y)\}\\ &= \{(0,1),(0,2),(1,2),(0,3),(1,3),(2,3),(0,4),(1,4),(2,4),(3,4),\cdots\}\end{aligned}$$

- $\{(x,y) \in \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 = 1.0\}$ is a circle of radius 1.
- $\{(a,b,c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mid a^2 + b^2 = c^2\}$ is the set of Pythagorean triples.

A.0.6 The size of sets

$|S|$

We write $|S|$ for the *size* or *cardinality* of a set. Obviously some sets have sizes in $\mathbb{N}$; we call these *finite sets*. Thus

- $|\emptyset| = 0$
- $|\{13\}| = 1$
- $|\{2,3,5,7,11,13,17\}| = 7$
- if $i \leq j$ then $|\{i,..j\}| = j - i$.

Some sets, such as $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ have sizes that are not in $\mathbb{N}$; not surprisingly we call these infinite sets. The size of $\mathbb{N}$ is written as $\aleph_0$ (aleph null); you can think of this as just a symbol that represents the size of an infinite set. We say that two sets are the same size when we can arrange their members as pairs. Consider a set of horses and a set of saddles. If we can put a saddle on each horse with no saddles or horses left over, then the sizes of the sets are the same. The beauty of this definition is that we don't need to count the saddles or horses. For example we can pair-off the set of natural numbers and the set of squares of the natural numbers thus

$$(0,0)\ (1,1)\ (2,4)\ (3,9)\ (4,16)\ \cdots$$

You can see that there will be no left over squares or naturals in this process and so

$$\left|\left\{n \in \mathbb{N} \cdot n^2\right\}\right| = |\mathbb{N}| = \aleph_0$$

(This example is due to Galileo Galilei.)

Interestingly, according to generally accepted theory, not all infinite sets are the same size. For example it is generally accepted that $|\mathcal{P}(S)| \neq |S|$ and so $|\mathcal{P}(\mathbb{N})| \neq \aleph_0$, that is, you can not pair-off the natural numbers with subsets of the natural numbers in such a way that there are no subsets left over. This is an interesting idea. The number of descriptions of sets in any given language is surely the same as the number of natural numbers.[3] So this means that there are somehow subsets of $\mathbb{N}$ that we can not describe. Does it make sense that mathematics should contain objects that are too complex to be clearly described? Furthermore (according to generally accepted theory), $|\mathbb{R}| \neq \aleph_0$, so there are real numbers that can not be described. What are they and are they of any use? Whether and in what sense there really are infinities of different sizes is a fascinating question which is, for better or worse, not of relevance to this book, and so we will not discuss the matter further.

A.0.7 Set models of numbers and what isn't a set

Sets, as we will see, are very useful for modelling things. In the late 19th and early 20th centuries, philosophers and mathematicians considered whether mathematical objects, such as numbers, could be modelled with sets. Why would they ask that question? Well, they wanted to show that mathematics is purely a matter of definition; all objects of mathematics are defined in terms of something else and all conclusions of mathematics are simply the logical consequences of these definitions. In a way, this would mean that numbers don't really exist at all, they are just mental constructions defined in terms of the simple concept of sets. This was in contrast to Plato's idea

[3] To be formal about this, we could take the set of descriptions to be the programs in some simple programming language. For example

$$\textbf{var } x : \mathbb{N} := 0 \cdot \textbf{while true do } (\mathit{print}\ x; x := x + 2)$$

describes the set of even numbers. Each program is a finite sequence of characters; we can order the set of all programs primarily by length and secondarily alphabetically; each program corresponds to a natural number according to its position in the order.

that the objects of mathematics have some sort of existence independent of the mind.

I'm going to take a detour to look at how numbers and pairs can be modelled as sets. The models in the rest of this subsection won't be used elsewhere in this book, but it is informative to take a look at them.

Each natural number can be modelled as the set of all (models of) smaller natural numbers so:

$$\begin{array}{ll} 0 & \text{is modeled by } \emptyset \\ 1 & \text{is modeled by } \{\emptyset\} \\ 2 & \text{is modeled by } \{\emptyset, \{\emptyset\}\} \\ 3 & \text{is modeled by } \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \end{array}$$

Operations such an addition and multiplication can be modeled as operations on sets. For example, the operation of adding one is modelled by

$$X \cup \{X\}$$

The set of all (models of) natural numbers is an interesting set in that it not only has an infinite number of members, but is infinitely deep!

A pair (x, y) can be modelled by $\{\{X\}, \{X, Y\}\}$ where X and Y are sets that model objects x and y. Thus

$$(0, 1) \quad \text{is modeled by} \quad \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$$

and

$$(1, 0) \quad \text{is modeled by} \quad \{\{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\} \qquad .$$

Now each integer can be modelled by (a model of) a pair in which the first member indicates whether the integer is nonnegative ($\emptyset$) or negative ($\{\emptyset\}$) and the second member models the magnitude of the integer. So

$$\begin{array}{rll} 0 & \text{as an integer is modeled by} & \{\{\emptyset\}\} \\ 1 & \text{as an integer is modeled by} & \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ -1 & \text{as an integer is modeled by} & \{\{\{\emptyset\}\}\} \\ 2 & \text{as an integer is modeled by} & \{\{\emptyset\}, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}\} \\ -2 & \text{as an integer is modeled by} & \{\{\{\emptyset\}\}, \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} \end{array}$$

You can see that we have a different model of the number 1 depending on whether we are talking about natural numbers or integers. This isn't a

problem as long as we are clear about what kinds of things we are talking about.

Rational numbers can be represented by pairs consisting of an integer numerator and a nonzero, natural denominator, where the numerator and denominator have no common factors larger than 1. Thus $\frac{1}{2}$ is represented by $\{\{N\}, \{N, D\}\}$ where N is the model of 1 as an integer ($N = \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$) and D is the model of 2 as a natural number ($D = \{\emptyset, \{\emptyset\}\}$).

One way to represent a real number is by the set of all rational numbers that are not greater than it. So, for example, the set representing π contains $\frac{3}{1}$, $\frac{31}{10}$, $\frac{22}{7}$, $\frac{314}{100}$, and $\frac{3141}{1000}$, but does not contain $\frac{4}{1}$, $\frac{32}{10}$, $\frac{315}{100}$, and $\frac{3142}{1000}$. If we use the set representations of rational numbers, we have a representation of the real numbers that is entirely based on sets. This representation was proposed and investigated by Richard Dedekind in the late 19^{th} century.

Each complex number can be represented as a pair of real numbers. Often mathematicians deliberately ignore that this is a model and simply consider the set of complex numbers and the set of pairs of real numbers to be the same thing.

It is certainly useful to know that all this can be done. For example the properties of complex numbers were considered very mysterious until Gauss pointed out that they can be represented by pairs of real numbers. Likewise, it was a huge advance when René Descartes modelled the points on a plane with pairs of real numbers. Furthermore, the concept of a real number was not well understood until it could be modelled in terms of simpler things, namely the rational numbers.[4] From the point of view of the philosophy of mathematics, it is satisfying to know that numbers and arithmetic can be based on the seemingly simple concept of sets.

However, for the purposes of this book, we will consider that each number is a mathematical object that is *not* a set and that each tuple is an object that is *not* a set or a number. Thus, for the purposes of this book, it makes no sense to write, for example $1 \in 2$ or $3 \in (3, 5)$. We won't consider statements like this to be true or false, but rather to be meaningless or ill-formed, i.e. type errors.

On the other hand, we *will* use sets and tuples to model concepts such

[4] If we understand real numbers as lengths of lines, the idea of modelling real numbers with rational numbers is very old. The ancient Greeks believed that rational numbers and geometric lengths were the same thing. When one of the Pythagorean brotherhood discovered that $\sqrt{2}$ is not rational they tried to keep this discovery a secret and are said to have gone as far as to have murdered the discoverer.

as relations, functions, sequences and undirected graphs that may be less familiar to you than are numbers. And, like the mathematicians who consider that a complex number *is* a pair of real numbers, we will consider that a function *is* a certain kind of triple, ignoring the fact that there exist other ways of expressing the concept of a function. The next section looks at how to define the concept of 'function' using sets and tuples.

A.1 Relations and functions.

A.1.0 Binary relations, partial functions, and total functions

binary relation

A *binary relation* is any triple (S, T, G) where S and T are sets and $G \subseteq S \times T$. We call S the *source*, T the *target*, and G the *graph* of the binary relation.

source
target
graph

For example the following is a binary relation[5]

$$R0 = (\{1,2,3\}, \{1,2,3\}, \{1 \mapsto 2, 1 \mapsto 3, 2 \mapsto 3\})$$

where the source and target are both the same set $\{1,2,3\}$. If $(x \mapsto y) \in G$ we say that the relation *maps x to y*.

$S \leftrightarrow T$

$S \leftrightarrow T$ is the set of all binary relations with source S and target T.

partial function

A *partial function* (S, T, G) is a binary relation in which, for each $x \in S$, there is at most one y such that $(x \mapsto y) \in G$.You can see that $R0$ is not a partial function because it maps 1 to both 2 and 3. On the other hand

$$P0 = (\{1,2,3\}, \{1,2,3\}, \{1 \mapsto 2, 2 \mapsto 3\})$$

is a partial function.

$S \overset{\text{par}}{\rightarrow} T$

$S \overset{\text{par}}{\rightarrow} T$ is the set of all partial functions with source S and target T.

total function

A *total function* (S, T, G) is a relation in which, for each $x \in S$, there is exactly one y such that $(x, y) \in G$.You can see that $P0$ is not a total function, as 3 maps to no element of the target set. On the other hand

$$T0 = (\{1,2,3\}, \{1,2,3\}, \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 3\})$$

is a total function as well as a partial function and a binary relation. Note that each total function is also a partial function.

$S \overset{\text{tot}}{\rightarrow} T$

[5] Recall that $1 \mapsto 2$ is just another notation for the pair $(1, 2)$.

$S \overset{\text{tot}}{\to} T$ is the set of all total functions with source S and target T.
Some examples

- Consider (S, S, G) where $S = \{0, 1, 2, 3\}$ and

  $$G = \{0 \mapsto 0, 0 \mapsto 1, 1 \mapsto 1, 1 \mapsto 2, 2 \mapsto 2, 2 \mapsto 3, 3 \mapsto 3, \mapsto 0\}$$

  This is a relation. It is not a partial function and not a total function.

- Consider (S, S, H) where $S = \{0, 1, 2, 3\}$ and

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

  It is a total function and also a partial function and a relation.

- Consider $(\mathbb{Z}, \mathbb{Z}, J)$ and

  $$J = \{(a \mapsto b) \in \mathbb{Z} \times \mathbb{Z} \mid b = a \times a\}$$

  This is a total function.

- Consider $(\mathbb{R}, \mathbb{R}, K)$ where

  $$K = \{(x \mapsto y) \in \mathbb{R} \times \mathbb{R} \mid x \times y = 1\}$$

  This is a partial function. It is not total since there is no $(x \mapsto y)$ pair with $x = 0$.

- Consider $(\mathbb{R}, \mathbb{R}, L)$ where

  $$L = \{(x \mapsto y) \in \mathbb{R} \times \mathbb{R} \mid y \times y = x\}$$

  This is not a partial function (nor a total function) since we have $4 \mapsto 2$ and $4 \mapsto -2$.

Exercise 87 List all functions in the following sets

- $\emptyset \overset{\text{tot}}{\to} \{9\}$
- $\{1\} \overset{\text{tot}}{\to} \{9\}$
- $\{1, 2\} \overset{\text{tot}}{\to} \{8, 9\}$

- $\{1,2\} \stackrel{\text{par}}{\to} \{8,9\}$

Exercise 88 Suppose that S and T are finite. How many relations are in $S \leftrightarrow T$? Of these, how many are total functions? How many are partial functions?

A.1.1 Domain and Range

defined for

A binary relation R, with source S, target T, and graph G, is *defined for* an item x in S if there is a y in T such that (x, y) is in G.

domain

The *domain* of a relation $R = (S, T, G)$ is the set of elements for which it is defined

$$\begin{aligned}\text{dom}(R) &= \{x \in S \mid R \text{ is defined for } x\} \\ &= \{x \in S \mid \text{there is a } y \in T \text{ such that } (x, y) \in G\}\end{aligned}$$

For a total function $f : S \stackrel{\text{tot}}{\to} T$, the domain is the same as the source S

$$\text{dom}(f) = S$$

range

The *range* of a relation $R = (S, T, G)$ is the set of elements that appear as the right component of a pair in the graph

$$\text{rng}(R) = \{y \in T \mid \text{there is an } x \in S \text{ such that } (x, y) \in G\}$$

A.1.2 Application

application

defined

$f(x)$

If $f = (S, T, G)$ is a partial function defined for $x \in S$, we write $f(x)$ to mean 'that $y \in T$ such that $(x, y) \in G$'. We call $f(x)$ the *application* of f to x. We say the expression $f(x)$ is *defined* iff f is defined for x. If f is a total function, then $f(x)$ is sure to be defined, provided $x \in S$.

A.1.3 A digression on terminology

The words "relation', function' and 'partial function' are defined differently by different people. Some people define 'function' to mean (what I've called) 'total function'. Some people define 'function' to mean (what I've called)

'partial function'. Some people use the words 'partial function' only for (what I've called) partial functions that are not total.[6]

Furthermore, some people aren't particularly consistent. For example, both the Mathworld and the Wikipedia web-sites' entries on 'function' manage to contradict themselves. (At least at the time I am writing this.) You should be aware that different authors will use these words somewhat differently. To avoid confusion, I will try to avoid using the word 'function' by itself, except when it is either obvious or unimportant whether I mean a total or partial function.

The word's 'domain' and 'range' are often used for what I'm calling the 'source' and 'target'.

A.1.4 Lambda expressions

lambda expression
λ

[This section may be safely skipped.]

A useful notation is that of *lambda expressions*. We write

$$(\lambda x \in S \cdot \mathcal{E})$$

to denote a partial function whose source is S and whose value, when applied to x, is the value of $\mathcal{E}$. In other words it is

$$(S, T, \{(x \mapsto y) \in S \times T \mid y = \mathcal{E}\})$$

for some T. Clearly this notation is vague about what the target set is, but that information is often either clear from context or irrelevant.

For example

$$\left(\lambda x \in \mathbb{Z} \cdot x^2\right)$$

is a function that squares its argument

$$\left(\lambda x \in \mathbb{Z} \cdot x^2\right)(5) = 25$$

[6] This is consistent with the normal English use of the words 'partial' and 'total'. In English 'partial' almost always means 'not total'. A total solar eclipse is not a variety of partial solar eclipse, as throughout a partial eclipse only a 'fraction' of the sun is covered. Mathematicians, however, recognize that $\frac{1}{1}$ is a fraction.

In mathematical usage the word 'partial' is often used to in a way that allows the possibility of 'totality', i.e., the way I am using it: A partial order may also be a total order. A partially correct algorithm may also be totally correct.

And if

$$g = \left(\lambda f \in \mathbb{Z} \xrightarrow{\text{tot}} \mathbb{Z} \cdot (\lambda x \in \mathbb{Z} \cdot f(f(x))\right)$$

then g is a function so that

$$g\left(\lambda x \in \mathbb{Z} \cdot x^2\right)(5) = 225$$

while

$$g\left(\lambda x \in \mathbb{Z} \cdot 2x + 5\right)(3) = 27$$

Aside on the Lambda Calculus. The way that I've defined lambda expressions is '*extensional*', i.e. in terms of a set of pairs. This way of looking at a function means that we look at functions as 'completed' or 'actual' infinities. Originally, lambda expressions were employed as a kind of programming language in which one can write a 'rule' for calculating the value of a function. In this case the lambda expression expresses the intention of the person who wrote it. The lambda expression is regarded, then, as '*intentional*'. In this view, the lambda expression is finite (it takes a finite number of symbols to express it) and the calculation of a value of an application from a lambda expression and an argument is either finite or no value is computed. When using lambda expressions this way, we are using a 'lambda calculus'. Lambda calculus is historically and practically important, but I won't be discussing it further in this book. The lambda expression notation, however, is useful when one wants to use a function, but not to give it a name. **End of Aside.**

A.2 Sequences

finite sequence

A *finite sequence* over a set S is a total function in $\{0, ..n\} \xrightarrow{\text{tot}} S$, where n is some natural number. (Recall that $\{0, ..n\}$ is the set of the first n natural numbers.) Finite sequences are also called strings, when the set S is a set of symbols. See Chapter 7 for more about strings.

For each $n \in \mathbb{N}$, the set of finite sequences of length n over S is

$$S^n = \left(\{0, ..n\} \xrightarrow{\text{tot}} S\right)$$

length
$\|s\|$
S^*

If $s \in S^n$, we say that its *length* is n and write $\|s\| = n$. Considering the case of $n = 0$, we find there is (for each S), one sequence $(\emptyset, S, \emptyset)$ which has an empty domain. This is called the empty sequence.

The set of all finite sequences over S is

$$S^* = \bigcup n \in \mathbb{N} \cdot S^n$$

Note that while the size of the set S^* is infinite (even for finite S), the length of each element of S is finite.

concatenate
$s \hat{} t$

We can *concatenate* two finite sequences s and t to get a string $s \hat{} t$. If $u = s \hat{} t$ then,

$$\begin{aligned} \|u\| &= \|s\| + \|t\| \\ u(i) &= s(i), \text{ for all } i \text{ such that } 0 \leq i < \|s\| \\ u(\|s\| + i) &= t(i), \text{ for all } i \text{ such that } 0 \leq i < \|t\| \end{aligned}$$

one-way infinite sequence

One-way infinite sequences are functions from $\mathbb{N} \xrightarrow{\text{tot}} S$. The set of one-way infinite sequences may be written as S^∞. If $s \in S^\infty$ then $\|s\| = \infty$.

The definition of catenation can be extended to include one-way infinite sequences as follows. If $s \in S^\infty$, $t \in S^* \cup S^\infty$, and $u = s \hat{} t$ then $u = s$; furthermore, if $s \in S^*$, $t \in S^\infty$, and $u = s \hat{} t$, then

$$\begin{aligned} u &\in S^\infty \\ u(i) &= s(i), \text{ for all } i \text{ such that } 0 \leq i < \|s\| \\ u(\|s\| + i) &= t(i), \text{ for all } i \in \mathbb{N} \end{aligned}$$

We use two notations for explicitly listing the elements of a finite sequence.

- In the first notation, the elements are listed in square brackets so that
  - $[\,]$ is the empty string.
  - $[a]$ is a string of length 1 with $[a](0) = a$
  - $[a, b, c, a, b]$ is a string of length 5 with $[a, b, c, a, b](0) = a = [a, b, c, a, b](3)$, $[a, b, c, a, b](1) = b = [a, b, c, a, b](4)$, and $[a, b, c, a, b](2) = c$.
- The second notation is commonly used in formal language theory:
  - The empty string ϵ is the sole function from $S^0 = \left(\emptyset \xrightarrow{\text{tot}} S\right)$.
  - We write strings of length 1 or more by listing the elements in double quotes. E.g. "*abcab*" .

By analogy with set builder notation (of Section A.0.2), we can invent a *sequence builder notation*. For example,

$$\left[n \leftarrow [0, ..] | n \text{ is prime} \cdot n^2\right]$$

is the one-way infinite sequence of squares of primes. $[4, 9, 25, \ldots]$. Notations similar to this are common in many programming languages (ML, Haskell, and Python, for example), but not commonly used in mathematics.

A.3 Graphs

Consider a social networking website such as facebook. There is a set of accounts and any two accounts either are or are not 'friends'. We have a set of accounts and a set of friendships. We can consider each friendship to be represented by a set of two accounts. Such a network is an example of a graph. Graphs describe how things connect to other things. Because we can look at connections in greater or lesser detail, there are a number of kinds of mathematical objects that share the name 'graph.'

undirected graph

An *undirected, simple graph with loops* is a pair $G = (V, E)$ where V is a set, called the set of *vertices*, and E is set, called the set of *edges*, of sets of vertices of size 1 or 2. I.e.

vertex

edge

$$E \subseteq \{(a, b) \in V \times V \cdot \{a, b\}\}$$

An edge of size 1 is called a *loop.* (A loop connects a vertex to itself.) If we disallow loops, then we have a *undirected simple graph without loops.* Of course the phrase 'with loops' doesn't mean there are necessarily loops in the graph, just that they are possible. Our example a social networking website is an example of an undirected simple graph without loops.

Consider a network of computers. We wish to describe how they are connected. As a first cut we might describe which computers are connected to which others. An undirected simple graph is sufficient for this. But what if some of the communication links are one-way? Maybe HAL can send information directly to TRON, but not the other way around. In this case we need a tool that is slightly sharper.

directed graph

A *directed, simple graph with loops* is a pair $G = (V, E)$ where V is a set, called the set of *vertices*, and E is set, called the set of *edges*, of pairs of vertices I.e.

$$E \subseteq V \times V$$

In this case, an edge (a, a) is called a *loop*. If we disallow loops, then we have a *directed, simple graph without loops.*

Consider our computer network again. Suppose that we wish to show that if two links are cut, all computers can still communicate with each other. In this case we need to consider how many communication links there are in each direction between each pair of computers. We need an even sharper tool.

multigraph

The definitions above do not allow multiple edges between the same pair of vertices — except that there might be distinct directed edges in opposite directions (a, b) and (b, a). Multigraphs solve this problem. A *directed multigraph with loops* is tuple

$$(V, E, \overleftarrow{}, \overrightarrow{})$$

where V is a set, called the set of *vertices*, E is a set called the set of *edges*, and $\overleftarrow{}$ and $\overrightarrow{}$ are functions from $E \stackrel{\text{tot}}{\to} V$, called respectively the *source* and *target* functions. Consider our computer network again. Suppose each link is given a unique number; the numbers can serve as edges. If there are two communication links from HAL to TRON, numbered m and n then we have $\overleftarrow{m} = \overleftarrow{n} = \text{HAL}$ and $\overrightarrow{m} = \overrightarrow{n} = \text{TRON}$.

If $\overleftarrow{e} = \overrightarrow{e}$, e is called a *loop*. If we disallow loops, of course, we have a *directed multigraph without loops.*

Thus graphs come in eight flavours according to 3 dichotomies: simple or multi-, directed or undirected, with or without loops. I haven't bothered to define undirected multigraphs, as they aren't often used. Authors often use the term *graph* by itself to mean any of these various possibilities and are sometimes vague about which one they intend.

Graphs are often accompanied by labelling functions that map from either the vertices or the edges to some set. For example an edge-labelled directed multigraph is a tuple $(V, E, \overleftarrow{}, \overrightarrow{}, \lambda)$ where $(V, E, \overleftarrow{}, \overrightarrow{})$ is a directed multigraph, and $\lambda : E \stackrel{\text{tot}}{\to} R$ for some set R.

A.4 Categories

category

[[To be deleted if categories are never used.]] A *category* is a 6-tuple $(V, E, \overleftarrow{}, \overrightarrow{}, id, ;)$ where V is a set, called the *objects*, E is a set, called the *arrows*, $(V, E, \overleftarrow{}, \overrightarrow{})$ is a directed multigraph with loops, id is a total function from objects to arrows, and ; is a partial function, called *composition*, mapping pairs of arrows

to arrows.[7] Furthermore following must be true, for all arrows e, f and g, and objects a:

- $id(a)$ is a loop on a, i.e., $\overleftarrow{id(a)} = a = \overrightarrow{id(a)}$, for all $a \in V$
- $e; f$ is defined exactly if $\overrightarrow{e} = \overleftarrow{f}$,
- $\overleftarrow{e; f} = \overleftarrow{e}$ and $\overrightarrow{e; f} = \overrightarrow{f}$ if $e; f$ is defined,
- $id(\overleftarrow{e}); e = e = e; id(\overrightarrow{e})$, and
- $(e; f); g = e; (f; g)$ if all compositions are defined (i.e., if $\overrightarrow{e} = \overleftarrow{f}$ and $\overrightarrow{f} = \overleftarrow{g}$).

A.5 Propositional logic

boolean value
$\mathbb{B}$

We assume a set $\mathbb{B}$ of size 2. $\mathbb{B} = \{\mathfrak{true}, \mathfrak{false}\}$. You can think of $\mathfrak{true}$ and $\mathfrak{false}$ as being primitive mathematical objects like numbers. Like numbers, they are not sets. The set $\mathbb{B}$ is called the set of *boolean values*.

A.5.0 Implication, follows from, and negation (NOT)

A.5.0.0 Implication

implication
$\Rightarrow$

Define a function $(\Rightarrow) \in \mathbb{B} \times \mathbb{B} \xrightarrow{\text{tot}} \mathbb{B}$ so that

$$(\mathfrak{false} \Rightarrow q) = \mathfrak{true}, \text{ for all } q \in \mathbb{B}$$
$$(\mathfrak{true} \Rightarrow q) = q, \text{ for all } q \in \mathbb{B}$$

In table form

p	q	$p \Rightarrow q$
false	false	true
false	true	true
true	false	false
true	true	true

[7] Strictly speaking this definition defines a 'small category'. In general, there may be so many objects or arrows that we can't define a set that holds them all. For example, we might want every set to be an object, but then V would need to be a set of all sets, something that is known to cause trouble and thus is generally avoided. There is a category of sets, but it is not a small category and so doesn't fit our definition.

The function $\Rightarrow$ is called *implication*. Its left operand is called the *antecedent* and its right operand is called the *consequent*. For $p \Rightarrow q$, we generally say p *implies* q or *if* p *then* q.

$$\text{if } (p \Rightarrow q) = \mathfrak{true} \text{ and } (q \Rightarrow r) = \mathfrak{true} \text{ then } (p \Rightarrow r) = \mathfrak{true} \text{ Transitivity}$$

follows from
$\Leftarrow$

A.5.0.1 Follows from

We can define a function $(\Leftarrow) \in \mathbb{B} \times \mathbb{B} \xrightarrow{\text{tot}} \mathbb{B}$ which is the similar to implication, but with the operands switched. The defining equation is

$$(p \Leftarrow q) = (q \Rightarrow p)\,, \text{ for all } p, q \in \mathbb{B}$$

We say p *follows from* q or p *if* q or p *is implied by* p.

not
$\neg$

A.5.0.2 Negation

Define $(\neg) \in \mathbb{B} \xrightarrow{\text{tot}} \mathbb{B}$ such that

$$\neg p = (p \Rightarrow \mathfrak{false})\,, \text{ for all } p \in \mathbb{B}$$

In table form

p	$\neg p$
$\mathfrak{false}$	$\mathfrak{true}$
$\mathfrak{true}$	$\mathfrak{false}$

We usually say *not* p for $\neg p$.

We can observe some algebraic laws about negation and implication. The reader should check that these are true for all boolean values of p and q.

$$\neg\neg p = p \text{ Involution}$$
$$(p \Rightarrow q) = (\neg q \Rightarrow \neg p) \text{ Contrapositive}$$

A.5.1 Conjunction (AND) and disjunction (OR)

disjunction
$\vee$

Define *disjunction* (OR) by

$$(\vee) \in \mathbb{B} \times \mathbb{B} \xrightarrow{\text{tot}} \mathbb{B}$$
$$(p \vee q) = \neg p \Rightarrow q$$

and *conjunction* (AND) by

conjunction
$\wedge$

$$(\wedge) \in \mathbb{B} \times \mathbb{B} \xrightarrow{\text{tot}} \mathbb{B}$$
$$(p \wedge q) = \neg\,(p \Rightarrow \neg q)$$

In table form we have

p	q	$p \vee q$	$p \wedge q$
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	true

The operands of $\wedge$ are called **conjuncts** and The operands of $\vee$ are called **disjuncts**.

Here are some laws involving conjunction and disjunction

$$\left.\begin{array}{l}(\text{true} \wedge p) = p \\ (\text{false} \vee p) = p\end{array}\right\} \text{Identity}$$
$$\left.\begin{array}{l}(\text{false} \wedge p) = \text{false} \\ (\text{true} \vee p) = \text{true}\end{array}\right\} \text{Domination}$$
$$\left.\begin{array}{l}(p \wedge p) = p \\ (p \vee p) = p\end{array}\right\} \text{Idempotence}$$

$$\left.\begin{array}{l}(p \wedge q) = (q \wedge p) \\ (p \vee q) = (q \vee p)\end{array}\right\} \text{Commutativity}$$
$$\left.\begin{array}{l}((p \wedge q) \wedge r) = (p \wedge (q \wedge r)) \\ ((p \vee q) \vee r) = (p \vee (q \vee r))\end{array}\right\} \text{Associativity}$$
$$\left.\begin{array}{l}(p \wedge (q \vee r)) = ((p \wedge q) \vee (p \wedge r)) \\ (p \vee (q \wedge r)) = ((p \vee q) \wedge (p \vee r))\end{array}\right\} \text{Distributivity}$$

$$\left.\begin{array}{l}(p \wedge \neg p) = \text{false} \\ (p \vee \neg p) = \text{true}\end{array}\right\} \left\{\begin{array}{l}\text{law of contradiction} \\ \text{law of excluded middle}\end{array}\right.$$
$$\left.\begin{array}{l}\neg(p \wedge q) = (\neg p \vee \neg q) \\ \neg(p \vee q) = (\neg p \wedge \neg q)\end{array}\right\} \text{De Morgan's laws}$$

$$(p \Rightarrow q) = (\neg p \vee q) \text{ Material implication}$$
$$(p \Rightarrow q) = (\neg q \Rightarrow \neg p) \text{ Contrapositive law}$$
$$(p \wedge q \Rightarrow r) = (p \Rightarrow (q \Rightarrow r)) \text{ Shunting}$$
$$(p \wedge q \Rightarrow r) = ((p \Rightarrow r) \vee (q \Rightarrow r)) \text{ Distributivity}$$
$$(p \vee q \Rightarrow r) = ((p \Rightarrow r) \wedge (q \Rightarrow r)) \text{ Distributivity}$$
$$(p \Rightarrow q \wedge r) = ((p \Rightarrow q) \wedge (p \Rightarrow r)) \text{ Distributivity}$$
$$(p \Rightarrow q \vee r) = ((p \Rightarrow q) \vee (p \Rightarrow r)) \text{ Distributivity}$$
$$\text{if } p \Rightarrow q \text{ and } q \Rightarrow r \text{ then } p \Rightarrow r \text{ Transitivity}$$

A.5.2 Equivalence and exclusive-or.

equivalence
$\equiv$

Define

$$(p \Leftrightarrow q) = (p \Rightarrow q) \wedge (q \Rightarrow p)$$
$$(p \not\Leftrightarrow q) = \neg(p \Leftrightarrow q)$$

These three operators are respectively the *equivalence*, *exclusive-or*. Note that for boolean p and q, $p \Leftrightarrow q$ is just the same as $p = q$.

A.5.3 Duality

As can be seen, laws about conjunction and disjunction come in pairs. In general if you have a law

$$\mathcal{E} = \mathcal{F}$$

involving only propositional variables, boolean values, and propositional operators, there is an equivalent law

$$\mathcal{E}' = \mathcal{F}'$$

where $\mathcal{E}'$ and $\mathcal{F}'$ are obtained from $\mathcal{E}$ and $\mathcal{F}$ (respectively) by replacing each operator with its dual according to the following table

$\wedge$	$\vee$
$\neg$	$\neg$
$\Leftrightarrow$	$\Leftrightarrow$
$\not\Leftrightarrow$	$\not\Leftrightarrow$
$\Rightarrow$	$\not\Leftarrow$

where

$$(p \not\Leftarrow q) = \neg\,(p \Leftarrow q) = (\neg p \wedge q)$$

A.5.4 Other notations

There is a diversity of notations used for propositional logic.

Many text books, especially those concerned with digital circuits, use $+$ for disjunction and $\cdot$ or nothing at all for conjunction. For example the distributivity of conjunction over disjunction is expressed by

$$p\,(q + r) = pq + pr$$

Clearly prior experience with the algebra of numbers helps one remember such a law. On the other hand, the distributivity of disjunction over conjunction is expressed as

$$p + qr = (p + q)\,(p + zr)$$

As this example shows, prior experience with the algebra of numbers may also be a bit of a psychological roadblock. The same textbooks generally also use 1 for $\mathfrak{true}$ and 0 for $\mathfrak{false}$. This notation was introduced to engineering by Shannon [[ref]] in one of the earliest works on computer engineering. Shannon was following Whitehead [[ref]] who was following Boole [[ref]]. Whitehead and Boole both wanted to explore the relationships propositional logic has with other algebras, so using the same notation for propositional logic, arithmetic, linear algebra, etc. made sense. However: when dealing with both numbers and boolean values, as we do in this book, the overloading of notation gets confusing: is $1 + 1$ equal to 1 or to 2?[8]

Programming languages have generally used notations that need only a limited character set. Pascal and several other languages use the key words `and` and `or`. Fortran uses `.AND.` and `.OR.`, while C, C++, and Java use the notations `&&` and `||`.[9]

[8] To add confusion, some works, especially those in cryptography, use $+$ for exclusive-or, giving $1 + 1 = 0$.

Boole himself was careful to avoid this particular question. For example to express inclusive-or ($\vee$) he would write $p + q(1 - p)$ and to express exclusive-or he would write $p(1 - q) + q(1 - p)$. (Boole used $1 - p$ for $\neg p$. Later writers would simplify the notation as $\overline{p}$.) Thus Boole avoided ever adding two 1s. Boole would not have agreed with

$$p + qr = (p + q)\,(p + r)$$

Nor did he accept $p + p = p$, even though he did accept $pp = p$. For Boole the laws of logic were a subset of the laws of ordinary arithmetic.

[9] When the 95 printing characters for the ASCII character set were being chosen in the

In this book, we will follow the notation that is standard in much of the literature of mathematics, logic, computer science, and computer engineering, which is to use $\wedge$ for conjunction and $\vee$ for disjunction. The use of $\vee$ for disjunction was popularized by Russell and Whitehead in their Principia Mathematica [[ref]], the use of $\wedge$ for conjunction followed later. This notation emphasizes the relationships with set theory and lattice theory.

This Book	Digital Logic	C/C++/ Java	C/C++/Java bitwise	Other
$\Rightarrow$				$\supset, \rightarrow$
$\wedge$	$\cdot$	&&	&	$\cdot$
$\vee$	$+$	||	|	
$\Leftrightarrow$		==		$\leftrightarrow$
$\not\Leftrightarrow$	$\oplus$	!=	^	$+$
$\neg$	—	!	~	$\sim$

A.6 Predicate Logic

In natural language, one often wants to express facts such as

- All flavours of ice-cream are good.
- Some people like peanut butter.
- The Q output is always equal to the D input of the previous time cycle.
- The system will be in the initial state within 5 seconds of the reset button being depressed.

To treat such sentences mathematically, we extend our logic with two "quantifiers"

- $\forall$, pronounced "for all", and

early 1960s the inclusion of the backward slash character was justified by its usefulness in writing 'and' and 'or' as /\ and \/. It seems that one of the leading forces in the defnition of ASCII was an fan of the Algol language, which included the characters $\wedge$ and $\vee$ in its official definition, but left undefined how to represent them in actual computer code. Nevertheless, no other major programming languages adopted this notation, probably because few computers used ASCII representation internally until the 1980s.
[[Ref http://www.trailing-edge.com/~bobbemer/BACSLASH.HTM]]

- $\exists$, pronounced "exists".

You can say that $\forall$ and $\exists$ have the same relationship to $\wedge$ and $\vee$ (respectively) as $\sum$ has to $+$.

We will extend our 2-valued propositional logic to deal with the quantifiers.

A.6.0 Substitution

A.6.0.0 Free and bound occurrences of variables

In Engineering, we often use variables to represent quantities in the real-world and boolean expressions containing variables to represent constraints on those quantities, imposed by nature or by an engineered system. For example, we might write

$$0 \leq x < 1$$

to express that the x coordinate of the position of something (say a robot's hand) is constrained within certain limits. A constraint

$$0 \leq y < 1$$

means something quite different. So we can conclude that the names matter. We call such occurrences of a variable "free".

Now consider the following pairs of expressions

- $z = \sum_{i=0}^{N} f(i)$ and $z = \sum_{j=0}^{N} f(j)$
- $z < \int_0^\infty f(u)\, du$ and $z < \int_0^\infty f(v)\, dv$
- $\{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 \leq 1.0\}$ and $\{(a, b) \in \mathbb{R} \times \mathbb{R} \mid a^2 + b^2 \leq 1.0\}$

In each case, the two parts of the pairs express the same constraint: they are equivalent. So in these cases the choice of variables i, j , u, v, x, y, a, and b doesn't matter. Such occurrences of variables are called "bound".

An analogous situation comes up in software. The two subroutines

```
void f() { ++i ; }
```

and

```
void f() { ++j ; }
```

are not equivalent, whereas the two subroutines

```
int g(int i) { return i+1 ; }
```

and

```
int g(int j) { return j+1 ; }
```

are equivalent.

substitution

A.6.0.1 Single variable substitution

Suppose that $\mathcal{E}$ is an expression and that $\mathcal{V}$ is a variable. We'll write $\mathcal{E}[\mathcal{V} : \mathcal{F}]$ for the expression obtained by replacing every free occurrence of x in $\mathcal{E}$ with $(\mathcal{F})$.

Examples

- $(x/y)[x : y + z]$ is $(y + z)/y$
- $(0 \leq i < N \wedge A[i] = 0)[i : i + 1]$ is $(0 \leq (i + 1) < N \wedge A[(i + 1)] = 0)$

When the parentheses would be redundant we can replace $\mathcal{E}$ with $\mathcal{F}$.

Example

- $(0 \leq i < N \wedge A[i] = 0)[i : i + 1]$ is $(0 \leq i + 1 < N \wedge A[i + 1] = 0)$

A.6.0.2 Multiple variable substitution

We sometimes need to replace a number of variables at once.

We'll write $\mathcal{E}[\mathcal{V}_0, \mathcal{V}_1, \cdots, \mathcal{V}_{n-1} : \mathcal{F}_0, \mathcal{F}_1, \cdots, \mathcal{F}_{n-1}]$ to mean the simultaneous replacement of n distinct variables by n expressions.

Example

- $(x/y)[x, y : y, x]$ is (y/x)
- whereas $((x/y)[x : y])[y : x]$ is (x/x)

A.6.0.3 Substitution and bound variables

We would like that making the same substitution in two equivalent expressions will give two equivalent expressions.

Thus we have to be a bit careful about exactly how substitution is defined.

In making substitutions we do not substitute for bound occurrences of variables. For example in the expression

$$\sum_{i=0}^{N-1} f(i)$$

the variable i is bound, so we don't substitute for it. Thus

$$\left(\sum_{i=0}^{N-1} f(i)\right)[f, i : g, j+1] \text{ is } \sum_{i=0}^{N-1} g(i)$$

Furthermore, it may be necessary to rename bound variables in order to avoid variables in $\mathcal{F}$ from being "captured". For example

$$\left(\sum_{i=0}^{N-1} (k \times i)\right)[k : i+1] \text{ is } \left(\sum_{j=0}^{N-1} (k \times j)\right)[k : i+1]$$

$$\text{which is } \sum_{j=0}^{N-1} ((i+1) \times j)$$

Note that I had to rename i to j to avoid conflict with the i in the replacement expression.

A.6.0.4 Notations

Different authors use different notations for substitution.

- In Hoare's Axiomatic basis paper [[ref]], he doesn't use any notation at all.
- In Hehner's practical theory paper [[ref]], he writes

$$(\text{substitute } \mathcal{F} \text{ for } \mathcal{V} \text{ in } \mathcal{E})$$

- Some writers write $\mathcal{E}(\mathcal{V}/\mathcal{F})$ while others write $\mathcal{E}(\mathcal{F}/\mathcal{V})$ or $(\mathcal{V}/\mathcal{F})\mathcal{E}$

- A common notation is $\mathcal{E}_{\mathcal{F}}^{\mathcal{V}}$.
- I use $\mathcal{E}[\mathcal{V} : \mathcal{F}]$ because it is short, hard to mistake for anything else, and does not involve subscripts or superscripts

A.6.0.5 One-point laws

The substitution notation lets usto express some useful laws called "one-point laws".

Consider an expression $(\mathcal{V} = \mathcal{F}) \Rightarrow \mathcal{A}$, where $\mathcal{A}$ is a boolean expression, $\mathcal{F}$ is an expression,, and $\mathcal{V}$ is a variable. When $\mathcal{V} \neq \mathcal{F}$ then the value of $\mathcal{A}$ doesn't matter, the implication will be true regardless of the value of $\mathcal{A}$. In the case of $\mathcal{V} = \mathcal{F}$, we need only worry about the value of $\mathcal{A}$ under the assumption that $\mathcal{V} = \mathcal{F}$. Similar reasoning applies to an expression $(\mathcal{V} = \mathcal{F}) \wedge \mathcal{A}$.

The one point laws can be expressed as:

$$((\mathcal{V} = \mathcal{F}) \Rightarrow \mathcal{A}) = ((\mathcal{V} = \mathcal{F}) \Rightarrow \mathcal{A}[\mathcal{V} : \mathcal{F}])$$

and

$$((\mathcal{V} = \mathcal{F}) \wedge \mathcal{A}) = ((\mathcal{V} = \mathcal{F}) \wedge \mathcal{A}[\mathcal{V} : \mathcal{F}])$$

For example we can simplify $i = 0 \wedge s = \sum_{k \in \{0,..i\}} f(k)$ to $i = 0 \wedge s = \sum_{k \in \emptyset} f(k)$ and then to $i = 0 \wedge s = 0$.

The more general principle is that if $\mathcal{C}$ implies $\mathcal{A} = \mathcal{B}$ then $(\mathcal{C} \wedge \mathcal{A}) = (\mathcal{C} \wedge \mathcal{B})$ and $(\mathcal{C} \Rightarrow \mathcal{A}) = (\mathcal{C} \Rightarrow \mathcal{B})$.

A.6.1 The Quantifiers $\forall$ and $\exists$

for all
$\forall$

Suppose that S is the finite set $\{0, 1, 2, 3\}$ and $\mathcal{A}$ is a boolean expression. We write

$$\forall x \in S \cdot \mathcal{A}$$

to mean

$$\mathcal{A}[x : 0] \wedge \mathcal{A}[x : 1] \wedge \mathcal{A}[x : 2] \wedge \mathcal{A}[x : 3] \quad ,$$

exists
$\exists$

and we write

$$\exists x \in S \cdot \mathcal{A}$$

to mean

$$\mathcal{A}[x : 0] \vee \mathcal{A}[x : 1] \vee \mathcal{A}[x : 2] \vee \mathcal{A}[x : 3] \quad ,$$

just as we would write

$$\sum_{x=0}^{3} \mathcal{E}$$

to mean

$$\mathcal{E}[x:0] + \mathcal{E}[x:1] + \mathcal{E}[x:2] + \mathcal{E}[x:3]$$

where $\mathcal{E}$ is some numerical expression.[10]

quantifier

The symbols $\forall$ and $\exists$ are called *quantifiers*. The $\forall$ is the *universal quantifier*, and the $\exists$ is the *existential quantifier*. $\forall x \in S \cdot \mathcal{A}$ may be pronounced "for all x in S it is the case that $\mathcal{A}$". $\exists x \in S \cdot \mathcal{A}$ may be pronounced "there exists an x in S for which $\mathcal{A}$".

universal quantifier

existential quantifier

As long as the set S is finite, $\forall$ and $\exists$ are convenient notations, but not very interesting, as they don't allow us to do any thing new. However, if we allow S to be an infinite set, then we have something really interesting. For example consider the set $\mathbb{N} = \{0, 1, 2, ...\}$ then

$$(\forall x \in \mathbb{N} \cdot \mathcal{A}) = \mathcal{A}[x:0] \wedge \mathcal{A}[x:1] \wedge \mathcal{A}[x:2] \wedge \cdots$$

and

$$(\exists x \in \mathbb{N} \cdot \mathcal{A}) = \mathcal{A}[x:0] \vee \mathcal{A}[x:1] \vee \mathcal{A}[x:2] \vee \cdots$$

In general

- $\forall \mathcal{V} \in S \cdot \mathcal{A}$ is $\mathfrak{true}$ if $\mathcal{A}[\mathcal{V} : y]$ is $\mathfrak{true}$ for every value $y \in S$, otherwise it is $\mathfrak{false}$.
- $\exists \mathcal{V} \in S \cdot \mathcal{A}$ is $\mathfrak{true}$ if $\mathcal{A}[\mathcal{V} : y]$ is $\mathfrak{true}$ for at least one value $y \in S$, otherwise it is $\mathfrak{false}$.

A.6.1.0 Some examples:

- All flavours of ice-cream are good:

  $$\forall f \in F \cdot good(iceCream(f))$$

  where F is the set of all flavours of ice-cream, *iceCream* is a function mapping a flavour to a variety of ice-cream, and *good* is a "predicate" (boolean function) indicating a variety is good.

[10] To be more consistent with the other notations in this book, we might write $\sum x \in \{0, ..4\} \cdot \mathcal{E}$.

- Some people like peanut butter:

$$\exists p \in P \cdot like(p, peanutButter)$$

  where P is the set of all people and *like* is a predicate indicating that its first argument likes its second argument.

- The Q output is always equal to the D input of the previous time cycle:

$$\forall t \in \mathbb{N} \cdot Q(t+1) = D(t)$$

  where Q and D indicate the values of Q and D in a given cycle. We use $\mathbb{N}$ as a time domain, as is appropriate for discrete time systems.

- The system will be in the initial state within 5 seconds of the reset button being depressed:

$$\forall t \in \mathbb{R}^+ \cdot reset(t) \Rightarrow (\exists u \in \mathbb{R}^+ \cdot t \leq u \leq t + 5 \wedge initial(u))$$

  where *reset* is a predicate indicating the reset button is depressed and *initial* indicates that the system is in its initial state. Here I have used $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x \geq 0\}$ to model time, as is appropriate for real-time systems. Implicitly the unit for time is seconds.

In the last example, it should be noted how $\Rightarrow$ is used with $\forall$ to indicate that we are only interested in certain time. I.e. $\forall t \in \mathbb{R}^+ \cdot reset(t) \Rightarrow \mathcal{E}$ says that property $\mathcal{E}$ holds for all times t where $reset(t)$ is true. Similarly $\wedge$ is to focus $\exists$ on certain times. I.e., $\exists u \in \mathbb{R}^+ \cdot t \leq u \leq t + 5 \wedge \mathcal{E}$ means that there is a u such that $t \leq u \leq t + 5$ where $\mathcal{E}$ is also true.

A.6.1.1 Relationship to set theory

Recall: The notation $\{x \in S \mid \mathcal{A}\}$ means the subset of S with elements x such that $\mathcal{A}$ is true.

We can understand $\forall$ and $\exists$ in terms of set notation:

$$(\forall x \in S \cdot \mathcal{A}) = (\{x \in S \mid \mathcal{A}\} = S)$$
$$(\exists x \in S \cdot \mathcal{A}) = (\{x \in S \mid \mathcal{A}\} \neq \emptyset)$$

I find that looking at quantifications this way sometimes helps make them easier to understand. For example, one thing that puzzles some students is

the behaviour of quantifiers when S is empty. Suppose that U is the set of all unicorns, and that *black* and *white* are boolean functions indicating that something is, respectively, black or white. Could it be that

$$\forall u \in U \cdot black(u)$$

and that

$$\forall u \in U \cdot white(u)$$

are both true? I.e. that all unicorns are black and also that all unicorns are white? Well if $U = \emptyset$ then we have

$$\begin{aligned} & \forall u \in U \cdot black(u) \\ =& (\{u \in U \mid black(u)\} = U) \\ =& (\{u \in \emptyset \mid black(u)\} = \emptyset) \\ =& (\emptyset = \emptyset) \\ =& \mathfrak{true} \end{aligned}$$

and of course a similar argument shows $(\forall u \in U \cdot white(u))$ is true. Some people find this surprising. Another way to look at it is this: Count the number of unicorns that are black. If there are the same number of black unicorns as the number of unicorns, then it makes sense that all unicorns are black.

Above we used the filtering notation for sets. We can also understand the quantifiers in terms of the mapping notation:

$$(\forall x \in S \cdot \mathcal{A}) = (\mathfrak{false} \notin \{x \in S \cdot \mathcal{A}\})$$
$$(\exists x \in S \cdot \mathcal{A}) = (\mathfrak{true} \in \{x \in S \cdot \mathcal{A}\})$$

In English when we say something like "all unicorns are black" we often really intend something like

$$\{\mathfrak{true}\} = \{u \in U \cdot black(u)\}$$

In took logicians in the nineteenth century a while to realize that the definition

$$(\forall x \in S \cdot \mathcal{A}) = (\mathfrak{false} \notin \{x \in S \cdot \mathcal{A}\})$$

is 'simpler' than the alternative, subtly different, and —from our point of view– incorrect definition

$$(\forall x \in S \cdot \mathcal{A}) = (\{\mathfrak{true}\} = \{x \in S \cdot \mathcal{A}\})$$

And it often takes students a while to get used to this too. Indeed it may seem pointless to worry about the case when S is empty, since no one sensible would bother to write a quantification over an empty set. However, sensible examples do come up when the set is dependant on some variable. Consider this definition that a sequence a is a 'prefix' of a sequence b

$$\|a\| \leq \|b\| \wedge (\forall i \in \{0, .. \|a\|\} \cdot a(i) = b(i))$$

When a is the empty sequence, its length is 0 and $\{0, .. \|a\|\} = \emptyset$. The definition correctly implies that an empty sequence is a prefix of every sequence.

The definition of the quantifiers in terms of set builder notation suggests a generalization of the notation:

$$(\forall x \in S \mid \mathcal{P} \cdot \mathcal{A}) = (\mathfrak{false} \notin \{x \in S \mid \mathcal{P} \cdot \mathcal{A}\})$$
$$(\exists x \in S \mid \mathcal{P} \cdot \mathcal{A}) = (\mathfrak{true} \in \{x \in S \mid \mathcal{P} \cdot \mathcal{A}\})$$

This generalization isn't really needed, as we can always write

$$(\forall x \in S \mid \mathcal{P} \cdot \mathcal{A}) \text{ as } (\forall x \in S \cdot P \Rightarrow \mathcal{A})$$

and

$$(\exists x \in S \mid \mathcal{P} \cdot \mathcal{A}) \text{ as } (\exists x \in S \cdot P \wedge \mathcal{A}) \quad .$$

A.6.1.2 Nested quantifiers

Quantifiers can be nested. You have to be careful about how you nest quantifiers. Let's look at an example. Suppose that S is a set of students and C is a set of courses.

- It is the same to say $\forall s \in S \cdot \forall c \in C \cdot takes(s, c)$ as it is to say $\forall c \in C \cdot \forall s \in S \cdot takes(s, c)$. The first says 'every student takes every course', while the second says 'every course is taken by every student': the meanings are the same.

- It is the same to say $\exists s \in S \cdot \exists c \in C \cdot takes(s, c)$ as it is to say $\exists c \in C \cdot \exists s \in S \cdot takes(s, c)$. The first says 'some student takes some course', while the second says 'some course is taken by some student': again the meanings are the same.

- But consider $\exists s \in S \cdot \forall c \in C \cdot takes(s, c)$. This means that some 'there is some student who is taking all of the courses'. On the other hand $\forall c \in C \cdot \exists s \in S \cdot takes(s, c)$ means that 'for each course there is some student who is taking it', i.e. no course has an enrollment of 0. The meanings are quite different.

Exercise 89 Write as clearly as possible, in English, the meanings of $\forall s \in S \cdot \exists c \in C \cdot takes(s, c)$ and $\exists c \in C \cdot \forall s \in S \cdot takes(s, c)$. Do these two expressions have the same meaning? Do they mean the same as any earlier expression?

For another example, suppose $f : \mathbb{R} \xrightarrow{\text{tot}} \mathbb{R}$ and that $a \in \mathbb{R}$. We can express that f approaches a in the limit by

$$\forall e \in \mathbb{R} \cdot e \geq 0 \Rightarrow (\exists x \in \mathbb{R} \cdot \forall y \in \mathbb{R} \cdot y > x \Rightarrow |f(y) - a| \leq e)$$

If words: f approaches a in the limit exactly if, for any positive quantity e, no matter how small, there is a number x such that to the right of x the value of f is within e of a.

A.6.1.3 Negated quantifications

What does it mean to combine negation with quantification? Is it the same to say $\forall s \in S \cdot \neg P$ as to say $\neg \forall s \in S \cdot P$? Let's look at an example. Suppose that S is the set of students taking Prof. Einstein's course in quantum theory and that $likes : S \xrightarrow{\text{tot}} \mathbb{B}$ is a function so that $likes(s)$ is $\mathfrak{true}$ if student s likes Prof. Einstein's course and $\mathfrak{false}$ otherwise. What does $\forall s \in S \cdot \neg likes(s)$ mean? It means that every student does not like the course. To put it another way: it is not the case that there is a student who likes the course. And we can write that formally as $\neg \exists s \in S \cdot likes(s)$. So we have

$$(\forall s \in S \cdot \neg likes(s)) = (\neg \exists s \in S \cdot likes(s))$$

How about $\neg \forall s \in S \cdot likes(s)$. What does that mean? It means that it is not the case that all students like the course. To put it another way: There is at least one student who does not like the course. We can write this as $\exists s \in S \cdot \neg likes(s)$. So we have

$$(\neg \forall s \in S \cdot likes(s)) = (\exists s \in S \cdot \neg likes(s))$$

If you consider the case where there are just two students in the course, say Alice and Bob, then these example boil down to De Morgan's laws. Consider

$$\begin{aligned} & (\forall s \in S \cdot \neg likes(s)) \\ = & \\ & \neg likes(Alice) \wedge \neg likes(Bob) \\ = & \text{ De Morgan} \\ & \neg \left(likes(Alice) \vee likes(Bob) \right) \\ = & \\ & (\neg \exists s \in S \cdot likes(s)) \end{aligned}$$

Thus we also call the following identities De Morgan's laws.

$$\begin{aligned} (\forall s \in S \cdot \neg \mathcal{A}) &= (\neg \exists s \in S \cdot \mathcal{A}) \\ \neg \left(\forall s \in S \cdot \mathcal{A} \right) &= (\exists s \in S \cdot \neg \mathcal{A}) \end{aligned}$$

Exercise 90 Use the definitions of the quantifiers given in Section A.6.1.1 to prove these laws.

A.6.1.4 Restricting variables

The set S in $\exists x \in S \cdot \mathcal{A}$ restricts our interest to the members of S; whether $\mathcal{A}$ is true or false outside of this set doesn't matter in trying to figure of whether $\exists x \in S \cdot \mathcal{A}$ is true. Consider $x \in S \wedge \mathcal{A}$. If x is not in S, then the expression is false, and the value whether $\mathcal{A}$ is true or false in this case doesn't matter in trying to figure out whether $x \in S \wedge \mathcal{A}$ is true. Suppose $S \subseteq T$. Then the following expressions are equivalent:

$$(\exists x \in S \cdot \mathcal{A}) \qquad \text{and} \qquad (\exists x \in T \cdot x \in S \wedge \mathcal{A})$$

For example, suppose that $prime : \mathbb{N} \xrightarrow{\text{tot}} \mathbb{B}$ is a function indicating that a number is prime. We can define a set $Prime = \{n \in \mathbb{N} \mid prime(n)\}$. Now you can see that the following expressions are equivalent

$$\forall x \in \mathbb{N} \cdot \exists y \in Prime \cdot y > x \wedge prime(y + 2)$$

and

$$\forall x \in \mathbb{N} \cdot \exists y \in \mathbb{N} \cdot prime(y) \wedge y > x \wedge prime(y + 2)$$

Similarly we can use implication to restrict variables introduced by $\forall$. In general, if $S \subseteq T$,

$$\forall x \in S \cdot \mathcal{A}$$

is equivalent to

$$\forall x \in T \cdot x \in S \Rightarrow \mathcal{A}$$

For example

$$\forall x \in \mathit{Prime} \cdot x = 2 \vee \mathit{odd}(x)$$

can also be written

$$\forall x \in \mathbb{N} \cdot \mathit{prime}(x) \Rightarrow x = 2 \vee \mathit{odd}(x)$$

Why does this work? Well consider the cases where x is not prime. In these cases the body of the quantification simplifies to

$$\mathfrak{false} \Rightarrow x = 2 \vee \mathit{odd}(x)$$

which simplifies to $\mathfrak{true}$.

A.6.1.5 Alternative notations

It is quite common to leave out the $\in S$ part when the domain of the variable is understood by some other means.

Many writers also leave out the $\cdot$ or replace it by some other symbol. There is a wide variety of conventions for using parentheses. You are likely to see any of the following in other works.

$$\begin{array}{l}
\forall x \cdot \mathcal{A} \\
\forall x\, \mathcal{A} \\
\forall x(\mathcal{A}) \\
\forall x, \mathcal{A} \\
[\forall x \in S,\ \mathcal{A}] \\
\forall x[\mathcal{A}] \\
(x)\mathcal{A}
\end{array}$$

A.6.2 Laws

There are a number of laws of predicate calculus. Some useful ones are summarized in this section.

Identity laws:

$$\begin{aligned}(\forall x \in S \cdot \mathfrak{true}) &= \mathfrak{true}\\ (\exists x \in S \cdot \mathfrak{false}) &= \mathfrak{false}\\ (\forall x \in S \cdot \mathfrak{false}) &= (S = \emptyset)\\ (\exists x \in S \cdot \mathfrak{true}) &= (S \neq \emptyset)\\ (\forall x \in \emptyset \cdot \mathcal{A}) &= \mathfrak{true}\\ (\exists x \in \emptyset \cdot \mathcal{A}) &= \mathfrak{false}\end{aligned}$$

Change of variable: Provided y does not occur free in $\mathcal{A}$,

$$\begin{aligned}(\forall x \in \mathbb{N} \cdot \mathcal{A}) &= (\forall y \in \mathbb{N} \cdot \mathcal{A}[x : y])\\ (\exists x \in \mathbb{N} \cdot \mathcal{A}) &= (\exists y \in \mathbb{N} \cdot \mathcal{A}[x : y])\end{aligned}$$

De Morgan's laws

$$\begin{aligned}(\forall x \in S \cdot \mathcal{A}) &= \neg(\exists x \in S \cdot \neg\mathcal{A})\\ (\exists x \in S \cdot \mathcal{A}) &= \neg(\forall x \in S \cdot \neg\mathcal{A})\end{aligned}$$

Domain splitting

$$\begin{aligned}(\forall x \in S \cup T \cdot \mathcal{A}) &= (\forall x \in S \cdot \mathcal{A}) \wedge (\forall x \in T \cdot \mathcal{A})\\ (\exists x \in S \cup T \cdot \mathcal{A}) &= (\exists x \in S \cdot \mathcal{A}) \vee (\exists x \in T \cdot \mathcal{A})\end{aligned}$$

Splitting

$$\begin{aligned}(\forall x \in S \cdot \mathcal{A} \wedge \mathcal{B}) &= (\forall x \in S \cdot \mathcal{A}) \wedge (\forall x \in S \cdot \mathcal{B})\\ (\exists x \in S \cdot \mathcal{A} \vee \mathcal{B}) &= (\exists x \in S \cdot \mathcal{A}) \vee (\exists x \in S \cdot \mathcal{B})\end{aligned}$$

Trading

$$\begin{aligned}(\forall x \in S \cdot \mathcal{A} \Rightarrow \mathcal{B}) &= (\forall x \in \{x \in S \mid \mathcal{A}\} \cdot \mathcal{B})\\ (\exists x \in S \cdot \mathcal{A} \wedge \mathcal{B}) &= (\exists x \in \{x \in S \mid \mathcal{A}\} \cdot \mathcal{B})\end{aligned}$$

One-point laws: Provided x does not appear free in $\mathcal{F}$ and that $\mathcal{F} \in S$,

$$(\forall x \in S \cdot (x = \mathcal{F}) \Rightarrow \mathcal{A}) = \mathcal{A}[x : \mathcal{F}]$$
$$(\exists x \in S \cdot (x = \mathcal{F}) \wedge \mathcal{A}) = \mathcal{A}[x : \mathcal{F}]$$

Commutative: Provided x is not free in T and y is not free in S,

$$(\forall x \in S \cdot \forall y \in T \cdot \mathcal{A}) = (\forall y \in T \cdot \forall x \in S \cdot \mathcal{A})$$
$$(\exists x \in S \cdot \exists y \in T \cdot \mathcal{A}) = (\exists y \in T \cdot \exists x \in S \cdot \mathcal{A})$$

Distributive laws: Provided x is not free in $\mathcal{A}$

$$\mathcal{A} \wedge (\exists x \in S \cdot \mathcal{B}) = (\exists x \in S \cdot \mathcal{A} \wedge \mathcal{B})$$
$$\mathcal{A} \vee (\forall x \in S \cdot \mathcal{B}) = (\forall x \in S \cdot \mathcal{A} \vee \mathcal{B})$$
$$\mathcal{A} \Rightarrow (\forall x \in S \cdot \mathcal{B}) = (\forall x \in S \cdot \mathcal{A} \Rightarrow \mathcal{B})$$

Distributive laws: Provided $S \neq \emptyset$ and x is not free in $\mathcal{A}$

$$\mathcal{A} \wedge (\forall x \in S \cdot \mathcal{B}) = (\forall x \in S \cdot \mathcal{A} \wedge \mathcal{B})$$
$$\mathcal{A} \vee (\exists x \in S \cdot \mathcal{B}) = (\exists x \in S \cdot \mathcal{A} \vee \mathcal{B})$$
$$\mathcal{A} \Rightarrow (\exists x \in S \cdot \mathcal{B}) = (\exists x \in S \cdot \mathcal{A} \Rightarrow \mathcal{B})$$

A.6.3 'Universally true' and 'Stronger Than'

universally true

If a boolean expression $\mathcal{A}$ is true regardless of the values of its free variables, it is said to be *universally true*.

If x, y, and z are integer variables, and p and q are boolean variables, each of the following expressions is universally true.

$$\begin{array}{l} \text{true} \\ x \geq x \\ x + 42 > x \\ x \in \{x, y, z\} \\ p \wedge q \Rightarrow p \end{array}$$

stronger than

A boolean expression $\mathcal{B}$ is considered to be *stronger than* a boolean expression $\mathcal{A}$ if

$$\mathcal{B} \Rightarrow \mathcal{A}, \qquad \text{is universally true}$$

For example

$$0 < x < y \text{ is stronger than } 0 \leq x < y,$$

$$0 < x < y \text{ is stronger than } 0 < x \leq y$$

and furthermore all three of these expressions are stronger than

$$0 \leq x \leq y$$

weaker than

If $\mathcal{B}$ is stronger than $\mathcal{A}$, we also say that $\mathcal{A}$ is *weaker than* $\mathcal{B}$.

It is possible that given two expressions, neither is stronger than the other. For example

$$0 \leq x < y \text{ is not stronger than } 0 < x \leq y, \text{ and}$$

$$0 < x \leq y \text{ is not stronger than } 0 \leq x < y.$$

Note that every expression is stronger than itself! Perhaps a better expression would be 'stronger than or just as strong as', but that would be quite long-winded.

equivalent

Two expressions that are equivalently strong (i.e. $\mathcal{A}$ is stronger than $\mathcal{B}$ and $\mathcal{B}$ is stronger than $\mathcal{A}$) are said to be *equivalent* expressions, they express the same thing about their free variables.

Here are some general laws about how the boolean operators interact with the relation of being stronger than.

$$\mathcal{A} \text{ is stronger than } \mathcal{A} \vee \mathcal{B}$$
$$\mathcal{A} \text{ is stronger than } \mathcal{B} \Rightarrow \mathcal{A}$$
$$\mathcal{A} \wedge \mathcal{B} \text{ is stronger than } \mathcal{A}$$

Monotonicity properties: If $\mathcal{B}$ is stronger than $\mathcal{A}$ then

$$\mathcal{B} \wedge \mathcal{C} \text{ is stronger than } \mathcal{A} \wedge \mathcal{C}$$
$$\mathcal{B} \vee \mathcal{C} \text{ is stronger than } \mathcal{A} \vee \mathcal{C}$$
$$\mathcal{C} \Rightarrow \mathcal{B} \text{ is stronger than } \mathcal{C} \Rightarrow \mathcal{A}$$

Anti-monotonicity properties: If $\mathcal{B}$ is stronger than $\mathcal{A}$ then

$$\neg\mathcal{A} \text{ is stronger than } \neg\mathcal{B}$$
$$\mathcal{A} \Rightarrow \mathcal{C} \text{ is stronger than } \mathcal{B} \Rightarrow \mathcal{C}$$

A.7 Precedence and associativity

As you know, mathematics uses "precedence conventions" to reduce the need for parentheses. For example we all know that

$$w \times x + y \times z$$

means

$$(w \times x) + (y \times z)$$

rather than

$$w \times (x + y) \times z$$

as $\times$ has "higher" precedence than $+$.

Furthermore we know that

$$a - b + c \text{ means } (a - b) + c$$

rather than $a - (b + c)$ as $-$ and $+$ are "left associative".

Some operators are associative meaning it doesn't matter how we add parentheses. E.g.

$$((a \wedge b) \wedge c) = (a \wedge b \wedge c) = (a \wedge (b \wedge c))$$

On the other hand

$$a \leq b < c \text{ means } (a \leq b) \wedge (b < c)$$

and we say that $\leq$, $<$, $=$, etc are "chaining"

The following table shows many of the operators used in the book in order

of precedence (highest to lowest)

$x(y)$	LA
$-x \quad \neg x$	
$x \times y \quad x/y$	LA
$x + y \quad x - y$	LA
$\cap$	A
$\cup$	A
$x = y \quad x \leq y \quad x < y \quad x \in y$	Ch
$x \wedge y$	A
$x \vee y$	A
$x \Rightarrow y \quad$ NA $\qquad x \Leftrightarrow y \quad x \not\Leftrightarrow y$	A
$x : y$	NA
if B **then** x **else** $y \quad$ **while** B **do** x	
$x; y$	A
$x \sqsubseteq y$	Ch
$\forall v \in S \cdot x \quad \exists v \in S \cdot x$	

where

LA	Left associative
RA	Right associative
A	Associative
NA	Nonassociative
Ch	Chaining

The low precedence of the quantifiers basically means that the scope of a quantified variable extends to the right to the end of the formula, unless there is explicit parenthesization or punctuation to stop it. I recommend putting quantifications in parentheses except when there is no possible confusion.

That $\wedge$ has higher precedence than $\vee$ is conventional, but I recommend using extra parentheses, e.g. to write

$$p \wedge q \vee r \text{ as } (p \wedge q) \vee r$$

Index

(,), 234
−
    applied to sets, 229
:
    in substitution, 254
:=, 22
;, *see* sequential composition
[ ], 244
⇐, 248
⇒, 247
∩, 229
∪, 229
∅, 227
∈, 244
∃, 253, 256
∀, 252, 256
∈, 227
λ, 242
|||, 243
⟨⟩, 10
||
    set, 235
↔, 239
¬, 248
⟼, 234
∉, 227
$\overset{\text{par}}{\rightarrow}$, 239
$\overset{\text{tot}}{\rightarrow}$, 240
⊑, 14
⊆, 228
∨, 248
    applied to specifications, 29
∧, 249
    applied to specifications, 29
$\{i, .., j\}$, 234
$\{i, ..j\}$, 234
*
    of a set, 244

abort, 35
alphabet, 83
alternation
    multiway, 33
    two-way, 33
AND, 249
angle-bracket notation, 10
antecedant, 248
application, 241
argmax, 234
argmin, 234
assert, 35
assignment, 22
    parallel, 23

behaviour, 7
behavioural specification, 7
belongs to, 7
binary relation, 239
boolean values, 247

cardinality, 235

category, 246
composition
    sequential, 25
concatenate, 84
concatenation
    of finite sequences, 244
conjunction, 249
    of specifications, 29
consequent, 248
contains, 227

D-flip-flop, 9
defined
    of a function, 241
defined for, 241
determined, 17
deterministic, 17
Deterministic Finite State Recognizer, 112
DFR, 112
DFR recognition algorithm, 112
difference, 229
directed graph, 245, 246
disjunction, 248
    of specifications, 29
dom, 241
domain, 241

element of, 227
empty set, 227
empty string, 84
ϵ, 84
equivalent, **266**
existential quantifier, 257
exists, 253, 256

finite sequence, 243
finite set, 228
follows from, 248

for all, 252, 256
function
    application, 241
    partial, 239
    total, 239

graph
    directed multi-, 246
    directed simple, 245
    of a relation or function, 239
    undirected simple, 245

if, 248
if
    multiway, 33
    two-way, 33
implementability
    preservation of, 37
implementable, 18, 19
implication, 247
infinite sequence
    one-way, 244
infinite set, 228
input variables, 16
intersection, 229
iteration, 35

lambda expression, 242
length
    of a sequence, 243
    of a string, 84
loop
    of a graph, 245, 246

magic, 35
maps to, 239
matrices, 27
max, 233
member, 227

min, 233
monotonic, 38
multigraph, 246

$\mathbb{N}$, 228
NDFR, 98
NDFR recognition algorithm, 110
nondeterministic, 17
Nondeterministic Finite State Recognizer, 98
not, 248

One-finger recognition algorithm, 104
OR, 248
output variables, 16
overdetermined, 17

pair, 234
partial function, 239
preserve implementability, 37

$\mathbb{Q}$, 228
quantifier, 257
quantifiers, 252

$\mathbb{R}$, 228
range, 241
refinement, 13
refines, 11
Regular expression, 87
Regular language, 87, 95
relation
    binary, 239
rng, 241

sequence, 243
    finite, 243
sequential composition, 25
set, 227
    builder notation, 230
    cardinality, 235
    contains, 227
    difference, 229
    element of, 227
    empty set, 227
    finite, 228
    infinite, 228
    intersection, 229
    member of, 227
    size, 235
    subset, 228
    union, 229
set builder notation, 230
signature, 6
size
    of a set, 235
skip, 22
source
    of a relation or function, 239
specification, 7, 8
state, 21
state space, 21
stepwise refinement, 11
stonger than, **265**
string, 84
    concatenate, 84
    empty, 84
    length, 84
subset, 228
substitution, 254
symbol, 83
system, 5

target
    of a relation or function, 239
Thompson's construction, 101
total function, 239
tuple, 234

underdetermined, 17
undirected graph, 245
unimplementable, 18, 19
union, 229
universal quantifier, 257
universally true, 41, **265**

variable, 6

weaker than, **266**
while, 35

$\mathbb{Z}$, 228

Colophon

This book was prepared with Scientific Workplace in the LaTeX2e language and uses the Computer Modern fonts.

Typeset January 22, 2018