Functional Programming in

Qi

Mark Tarver

Special Preprint

Printed in the United States of America

ISBN _

Published by Infinity Press

Dedicated to

John Tarver

Contents

Appendices

To the Reader

This book is a doorway to a radically new programming language based on nearly a decade of research into functional programming design. In technical terms Qi offers efficient pattern-matching, chronological backtracking, optional static typing and an approach to defining types that is in many ways more advanced than any programming language currently available. Translated into simple terms, Qi will help you build shorter, more reliable and more readable programs; it will save you time and it will save you from making errors.

Qi is under the GNU licence, so is no charge for using Qi.[1] Moreover Qi is based on and written in one of the oldest and most versatile programming languages - Lisp. This means that Qi will run on virtually any existing processor that you can name; Intel, Motorola, Athlon, ARM, … and virtually any operating system; Windows ('95, '98, 2000, ME, XP or NT), Linux, UNIX or Solaris. Qi provides excellent communications to the underlying Lisp; you will have access through Qi to the world of Lisp packages and environments; object-oriented programming through CLOS, graphical interface management through CLIM and statistical and numerical analysis through LispStat.

Qi excels in producing type-secure Common Lisp programs from compact and clear Qi source. Qi offers the option of static type checking in the spirit of ML and related languages, but with far more flexibility. You can extend the type system of Qi to incorporate any part of Lisp that can sustain type checking.

Qi is based on a new approach to type checking in which types are defined in a logic-based way that is clear, easy to understand and modify. Using sophisticated compilation techniques, Qi reduces these rules to machine code and applies them at high speed. Through Qi you can learn to use this same mechanism to devise reasoning systems of many kinds, from semantic nets to logic-based reasoning engines, expert systems and intelligent agents that will run at hundreds of thousands of rules per second.

This book is aimed at making you proficient in Qi. Having mastered this book, with the help of the pointers provided, you will be in a position to develop sophisticated reasoning programs in a fraction of the time needed to develop them in many other languages. If you are an active worker in the field of AI (Artificial Intelligence) then Qi will provide you with a powerful tool for rapidly prototyping your ideas. All the programs in this book are available from the Web and they are cited with an 🗀 indicating the path to the relevant program.

[1] Qi can be downloaded from www.lambdassociates.com.

Finally, if all the above jargon means nothing to you, then *whatever your background*, Qi will provide a medium through which you can come to understand something of the fascinating world of programming.   Everything is supplied except curiosity. That depends on you.

To the Teacher

To the prospective teacher contemplating teaching functional programming the question must be "Why Qi?" To this question, there are several answers. First Qi is a *clear* language, at least syntactically, with a syntax that extends to less than a page of BNF. The syntax is modeled after Edinburgh Prolog, which means that students going on to learn Prolog after having learnt Qi have a distinct advantage over students approaching Prolog from any other direction. Many of the issues regarding backtracking and choice points can be met in Qi in a form that is clean, tractable and easy to follow and lays a foundation for understanding the less tractable Prolog cut.

Second, Qi offers the choice of using or dispensing with static typing. The world of functional programming is divided into two camps (luckily, unarmed); protagonists of untyped languages, of which Lisp is the foremost recognised system; and defenders of typed languages of which Standard ML is perhaps the most widely used. Programmers in Lisp defend the breathtaking flexibility of Lisp that has allowed it to survive 40 years of progress and distrust what they view as the control freak mentality behind static typing. Alan Perlis speaks for the Lisp community when he says

"*It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures*."

Programmers in SML often regard Lisp programming as barely acceptable, open to error in a way that prevents programming being placed on a reliable footing. The debate is often raged with surprising vehemence in user groups and in coffee rooms. But any preference for one or the other depends on a value judgement, and one way to arrive at a mature understanding of the advantages of each position is to work in an environment that can support both paradigms. This is exactly what Qi provides.

The Qi type system is deductively presented as an open source program using explicit logic rules. No other language offers this degree of transparency and control over the type discipline of the language. For beginning students, observing and interacting with the Qi type checker is an effective way of appreciating the power of logic in computer science. For researchers, this degree of control allows them to experiment with type disciplines in a way that no other functional language will support.

Finally, Qi inherits and enhances the excellent Common Lisp environment, giving students long-term access to all the riches of Lisp without weighing them down initially with CARs and CDRs. Many of the uglier features of Lisp; its use of NIL for *false*, the lack of pattern-matching and the discrepancies between Lisp and lambda calculus in regard to currying and partial applications, are eliminated in Qi.

If this answers the question of "Why Qi?"; the next question must be "How can this book be taught?". Qi is available as source code that can be downloaded from the Net (www.lambdassociates.org). This code can be loaded into CLisp. CLisp itself runs under almost every OS and with Qi installed takes only a paltry 2Mb of RAM. For teaching, there is a complete set of Powerpoint slides that can be downloaded from the same site.

Functional Programming in Qi is both an introduction to a new language and an illustration of the language's main features through a series of topics that are central to computer science. Part I of the book introduces the untyped core language, which also covers recursion, list handling, non-determinism, procedural attachment, grammars and parsing. The core language is suitable as half-semester introduction to functional programming. At Stony Brook, part I was taught as a component of the first year foundation course in computer science. The application programs in part I are chosen to illustrate key programming features of untyped Qi and are deliberately kept short.

Part II of the book covers the use of types in Qi and the theory of their use. This part includes the use of sequent rules to define types, lambda calculus and the compilation of Qi into lambda calculus, and the type theory of Qi. The material here is more theoretical, and applications are less in evidence. Parts I and II would be taught together in an introduction to functional programming.

Part III of the book deals with more advanced applications in automated reasoning. The application programs in this part are larger and include complete theorem-provers for propositional and first-order logic, a simple expert system and a multi-agent model of a transport system. Unlike part I, not all the code for every program is included in these pages. The book explains how these programs work and leaves the code online for the student to study. However the code for the propositional theorem-prover, the expert system and a multi-agent model are included in their entirety.

Since expert systems and a multi-agents are large and significant fields in their own right, part III is only a taster for people interested in these areas. The treatment is deep enough to show the power of Qi in these areas, and, as in the rest of the book, further reading is included at the end of each chapter for those who want to study further.

There are over 100 exercise questions in this book. Questions that are marked with a single star * are more challenging. Double starred ** questions are suitable for final year projects or graduate proposals. Answers are not included, since the teacher may want to set these as part of the coursework. The questions have no marks attached, so that the teacher has the discretion of choosing the weighting.

Acknowledgements

This book was planned as a two year project, but in fact took ten years to complete. A project of such length must take a huge toll of energy and commitment from the author. During the long years of writing, I became conscious of how much I owed to the support of the people around me.

Amongst the academics, I single out Professor Graham Birtwistle, for his good nature and good advice, coming from a man whose history in computer science reaches back far further than my own. His personal example, in a department that became increasingly impersonal, was a reminder of a different and better set of standards for treating people. I must also add Dr Willi Riha, who as a friend accompanied me in perambulations around Golden Acre Park in discussions on computer science. I wish them good luck in their retirements.

Professor David Hogg was unfailingly supportive and it is thanks largely to him that Qi and Qi/Tk have appeared under the GNU licence.

Dr Antoni Diller and I have been correspondents since Antoni's employment as a research assistant at Leeds on the THRILLS proof assistant. His contribution to the architecture of THRILLS owed much to the profound humour of Lewis Carroll, who, I'm sure, would have appreciated some of the finer points of the design. Antoni supported the production of this book and his sense of the absurd preserved me from taking things too seriously. Professor Maria Faé, both friend and colleague, made an essential contribution to the final chapter of this book through her knowledge of transport engineering. Dr Magdalena Pamesa was a constant source of support in the final stages of this book.

I would also like to extend my thanks to Steffen Bakel of Imperial and Stefan Kahrs of Canterbury University for their valuable comments. Also to Delia Kesner of the University of Paris for her correspondence on types and patterns, and to Bernhard Beckert of the University of Koblenz for his correspondence on tableau theorem proving. Rod Burstall was very supportive during my time at the LFCS and Robin Milner provided useful clarification on the history of SML. I'd also like to thank Werner Hett, for permission to draw from his list of programming challenges. Carl Schapiro of SRI picked out many minor bugs in a preprint of this book as well as bugs in Qi/Tk. itself

Mark Tarver
January 2005.

1 Beginnings

1.1 Declarative Programming

One of the first skills that every student acquires in learning to program is to use **assignments**. An assignment has the form “let X = X+1”, whereby a variable is given a value in a program. In most languages, assignments are foundational; programs cannot be written without them. If we take a language like C or Fortran, and subtract the ability to write assignments, then the result is no longer a viable programming language. These languages are called **imperative**, **procedural** or **non-declarative** languages.

Declarative languages can survive this kind of subtraction. If we take a language like Qi and subtract assignments, then the result is still a viable programming language. In the case of **pure declarative programming languages** like Miranda^{TM}, where there are no assignments, this subtraction leaves the language unchanged. So in a declarative programming language, any computation can be described without the use of an assignment statement. Declarative languages are themselves partitioned into **functional programming languages**, like Qi and Miranda^{TM}, and **logic programming languages**, of which **Prolog** is the most widely used.[2]

This gives an accurate, but shallow characterisation of the procedural/declarative distinction. It does not explain why some programmers have been attracted to the declarative paradigm; as John Hughes (1990) puts it.

[2] The partition is a little fuzzy; during the ’80s several experimental languages explored the connection between logic and functional programming and the result was a series of hybrid languages such as LOGLISP (**Robinson** and **Sibert** (1980)) and LIFE (**Ait-Kaci** and **Podelski** (1993)) that shared features of both paradigms. Other systems like POPLOG (**Sloman** (1985)), provide an environment for communicating between programs written in both styles. The subject of logic programming is not dealt with in this book, but further reading on the subject can be found at the end of chapter 7.

It says a lot about what functional programming isn't (it has no assignments....) but not much about what it is. The functional programmer sounds rather like a mediaeval monk, denying himself the pleasures of life in the hope it will make him virtuous. To those more interested in material benefits, these advantages are totally unconvincing.... It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be. Clearly this characterisation of functional programming is inadequate. We must find something to put in its place - something that not only explains the power of functional programming but also gives a clear indication of what the functional programmer should strive towards.

To begin to meet Hugh's requirements, it is necessary to understand how functional programming originated and how the split between procedural and declarative programming began. We have to return to the beginning of computer science.

1.2 Mathematical Foundations

Computer science theory began with the attempt to make precise the idea of an **algorithm**. The name derives from the ninth-century Persian mathematician **Mohammed ibn Musa al-Khowarizmi** who described simple procedures for carrying out addition, subtraction, multiplication and division in the new Indian decimal system. The OED defines it as "**a procedure consisting of a predetermined series of steps**".

In ordinary speech, an algorithm is sometimes expressed in the native tongue of the speaker. So the rules for subtracting numbers are taught to us using conversation and examples. Conversational vocabulary is often imprecise and frequently rather verbose at the task of describing algorithms. Take this example, of an explanation of how to find square roots using Newton's Method of Approximation (a method studied in chapter 5 of this book).

Take the number you want to find the square root of and take a guess at the square root. If you are happy with your guess, then fine. If you want to have a better approximation than this guess, then take the average of the guess together with the result of dividing the number you want to find the square root of by that guess. That will give you a better guess. Keep on doing this until you have a guess you're happy with.

These instructions might serve their purpose, but they are vague, long-winded and difficult to follow. The example shows very clearly the weakness of relying on English, or any other conversational language, to express algorithms. Though the OED definition is good enough for casual use, it is not formally precise. But not until the twentieth century did mathematicians try to construct a definition of an algorithm that was formally precise.

When they did so, two leading mathematicians, **Alan Turing** in England and **Alonzo Church** in America, came up with different answers. However, the answers, though different, later proved equivalent, in the sense that any procedure that could be represented as an algorithm according to Church's answer, could be represented as an algorithm according to Turing's answer. It was rather as if two biologists trying to characterise the species of tigers, had defined them as "the species of which the largest land carnivore native to India are members" and as "the members of the species *Pantheris tigris*". In a similar way, the definitions of Church and Turing were different, but they characterised the same group of processes.

The **Church-Turing thesis** is that any adequate definition of an algorithm will prove equivalent in this way to the definitions provided by Church and Turing. Since the idea of "adequate" is not formally precise, the Church-Turing thesis is not mathematically provable. However, in practice other definitions of "algorithm" offered since Church and Turing, by for example, **Post** (1943)**,** have proved equivalent to Church and Turing's versions in exactly the way that the Church-Turing thesis predicts. So most computer scientists are happy to accept that these definitions of "algorithm" capture what is important for them about this word.

Both Church and Turing realised that trying to characterise algorithms in terms of what could be said in English was a waste of time. Instead, they both invented different formal languages in which algorithms could be more clearly expressed. In addition, both men provided a series of rules for manipulating this formal language in a set manner. An algorithm was defined by both men as a procedure that could be expressed in their chosen language. However, the language that each chose was very different, and the differences were the foundation for the later procedural-declarative split in programming.

Turing's idea was that an algorithm could be represented as a series of instructions that could be executed by a rather simple machine, that later became known as a **Turing Machine** (figure 1.1). A Turing machine consists of three components.

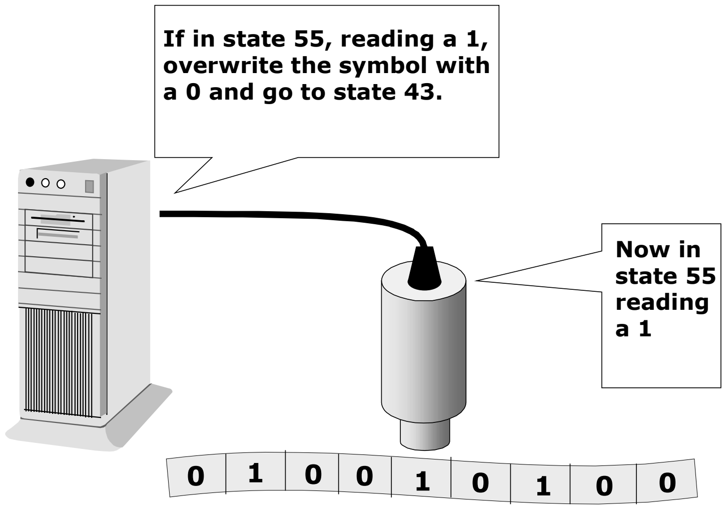

If in state 55, reading a 1, overwrite the symbol with a 0 and go to state 43.
Now in state 55 reading a 1
0 1 0 0 1 0 1 0 0

Figure 1.1 A Turing Machine Executing a Program Instruction

1. An endless stream of tape segmented into an infinite number of adjacent squares. Some of these squares may have symbols on them.
2. A mechanical head that positions itself over a square and which can read the symbol on that square or overwrite it. The head can also move to the square to the left of the square it is currently over or to the square to the right.
3. A program or series of instructions that tell the mechanical head what to do. It is assumed that a Turing machine is always in an internal state. These states are usually identified by the natural numbers 0,1,2, 3.... An instruction has the form

If in state n, reading symbol S_j, then execute action A_k and jump to state q_l.

Action A_k could be the action of halting (the computation ends), or of overwriting the scanned symbol by another symbol or of moving to the left adjacent square or the right adjacent square. A program for a Turing machine is a set S of instructions or quadruples $<n, S_j, A_k, q_l>$ such that if $<n, S_j, A_k, q_l> \in S$ and $<n, S_j, A_k', q_l'> \in S$, then $A_k = A_k'$ and $q_l = q_l'$ (i.e. there are no conflicting instructions about what the machine should do). The machine halts when it is reading symbol S_j in state n where for all A_k and for all q_l, $<n, S_j, A_k, q_l> \notin S$ (i.e. the machine has no instructions about what to do when reading symbol S_j in state n).

Turing's machine contains no restrictions on the number of symbols it can read or print, but only two are necessary - conventionally they are 0 and 1. To set a Turing machine running, the machine is started in its lowest state, scanning the leftmost of a series of 0s and 1s; this series represents the input to the machine. The output is what is left on the tape when the machine halts. An algorithm is defined as any terminating procedure that a Turing machine can be programmed to perform.

Turing's approach is easily appreciated by anybody with a simple working knowledge of the architecture of a computer. The states of the Turing machine correspond to the line numbers of a conventional program; the action of jumping to a new state is mirrored in the JUMP of assembly language. The endless tape is just a model of the bounded memory of the computer and the head of the Turing machine becomes the CPU of the digital computer. When World War II began a few years after Turing published his paper on the Turing machine, there was a demand for real computing engines that could tackle the difficult tasks of cracking enemy codes and calculating bomb and shell trajectories. Turing's pencil-and-paper creation provided a blueprint for engineers, and consequently became immensely influential in the design of the modern digital computer.

Across the Atlantic in America, Church had very different ideas. For many years, mathematicians had worked with the idea of a function. Put simply, a function is a mathematical object that receives one or more inputs and delivers an output. Algorithms too, receive inputs and deliver an output, so it is natural to consider that algorithms are a subclass of the class of functions. Usually when mathematicians wish to explain the properties of a function, they use equations to do it. Here is an equation that converts Fahrenheit to Centigrade.

$C = 5/9 \times (F - 32)$

We can easily distil a procedure from this equation that reads.

To convert Fahrenheit to Centigrade, subtract 32 from Fahrenheit and multiply by 5/9.

A more complex example comes from Boyle's Law, which relates the pressure of a confined gas to the volume it occupies and its temperature. According to Boyle's Law, if a confined gas is kept at pressure P1, occupying volume V1 at temperature T1, and then the gas has its pressure, volume or temperature changed so that it then has pressure P2, volume V2 and temperature T2, the change will satisfy the equation.

$(P1 \times V1) / T1 = (P2 \times V2) / T2$

So, from Boyle's Law, it is possible to construct a function that given the old temperature of the gas, its pressure and volume, and the new pressure and volume, will produce as output the new temperature. The equation is

$T2 = (P2 \times V2 \times T1) / (P1 \times V1)$

Again, we could easily distil a procedure from this equation. Here is one.

To calculate T2, first multiply P1 by V1 to give a result R. Then multiply P2 by V2 by T1 to give a result S. Then divide S by R.

However, we could also distil this procedure.

To calculate T2, first multiply P2 by V2 by T1 to give a result S. Then multiply P1 by V1 to give a result R. Then divide S by R.

The result is the same whichever procedure we choose. The point about calculations involving Boyle's Law is that they are algorithmic or mechanical, whichever order we choose to do them in, and we should always get the same result. This gives another way of looking at algorithms. Rather than identify an algorithm with a procedure consisting of a fixed series of steps, we can think of an algorithm as a function that can be characterised in a certain way. What marks the function as computable is not that the way we compute with it is predetermined (because in the case of Boyle's Law this is not true), but rather the way in which the function can be described. This insight is the beginning of functional programming.

Nevertheless, things are not so simple. For though all algorithms can be represented as functions, not all functions are algorithms. Consider this function over the natural numbers.

f(n) = 1 if there are n successive 7s in the decimal expansion of π.
0 otherwise

This describes a function, but not one we can easily use to compute. For instance, if we wanted to compute the value of f(1000) we would have to search for 1000 successive 7s in the decimal expansion of π. So far 1000 successive 7s have not been detected in the decimal expansion of π, but this does not mean that they are not there. At best, all we can do is expand π looking for 1000 successive 7s and this may take forever. However, the concept of an algorithm requires a terminating procedure that is guaranteed to return a definite answer. Plainly, the function f does not define any such procedure. f is a **non-computable** function.

Once we begin to look for non-computable functions, they swarm in number. Some are bizarre, for example the function that maps a person to 1 if his maternal grandfather hated boiled eggs is still a function (though a queer one); but obviously there is no algorithm for determining one's ancestors' egg-eating habits. The problem is that ordinary language allows us to concoct functions in a very unrestricted way. If the class of computable functions, to which algorithms correspond, is to be isolated from the broader class of functions as a whole, we need a special language and vocabulary to do it.

This is what Church provided; he devised the **lambda calculus**[3] as a notation for describing computable functions. Along with this formalism, Church gave a precise set of rules for reasoning with and for simplifying expressions of this notation. To perform a computation, in Church's terms, was to apply a function to an input, and to derive a value. By formally stating the rules for deriving values, Church defined a model of computability. An algorithm was just a computable function. A computable function was an expression that could be written in the lambda calculus so that when it was supplied an input (again written in lambda calculus), by a fixed and finite series of applications of the rules of lambda calculus, an expression could be derived which could be simplified no further and which represented the solution.

In contrasting the approaches of Turing and Church, it is easy to see why Turing, rather than Church, was immediately influential in shaping the course of computing. Turing provided an actual model of a computing engine. Church's definition of computability gave little clue as to how the lambda calculus was to be implemented. Moreover, some of Church's constructions, like his representation of the natural numbers as lambda expressions, were very convoluted. Here, for example, is Barendregt's representation of 2 in pure lambda calculus.

$$(((\lambda\ x\ (\lambda\ y\ (\lambda\ z\ ((z\ x)\ y))))(\lambda\ x\ (\lambda\ y\ y)))(((\lambda\ x\ (\lambda\ y\ (\lambda\ z\ ((z\ x)\ y))))(\lambda\ x\ (\lambda\ y\ y)))(\lambda\ x\ x)))$$

The example is a little unfair, because functional programming languages do not use pure lambda calculus to represent numbers. However, it is easy to understand why an observer would conclude that Church's

[3] Which we study in chapter 12 of this book

definition of computability was of no practical use in computer science. Not until the ’60s did computer scientists begin to explore the consequences of applying Church’s ideas. In the intervening 20 years, engineers and programmers built their work on the paradigm given by Turing. This was to be fateful in the development of programming as a discipline.

1.3 The American Experience

When World War II ended, America emerged as the richest and most powerful nation on the planet, producing at one point 40% of the world’s GDP. Computer engineering gravitated naturally to a country that was rich in émigré scientists and natural resources. Consequently the early years of functional programming belong to America.

These early years are the early years of the American work into **Lisp**. Lisp was the offspring of an intellectual marriage conducted by **John McCarthy** between Church’s work on the lambda calculus and an earlier language called **IPL** (Information Processing Language) that was invented by two American scientists **Newell** and **Simon**. Newell and Simon were pioneering the study of automated deduction and needed a flexible language in which to program the computer to simulate human thought. Conventional languages require all information to be predeclared in variables or arrays. What Newell and Simon wanted was a language in which structures could be dynamically built up and broken down in response to the intellectual requirements of the program. They struck upon the idea of using the list as the fundamental data structure on which to organise their procedures.

The basic idea is that, whenever a piece of information is stored in memory, additional information should be stored with it telling where to find the next (associated) piece of information. In this way the entire memory could be organised like a long string of beads, but with the individual beads of the string stored in arbitrary locations. "Nextness" was not determined by physical propinquity but by an address, or pointer, stored with each item, showing where the associated item was located. Then a bead could be added to a string or omitted from a string simply by changing a pair of addresses, without disturbing the rest of the memory.[4]

In fact, the pair consisting of an item of a list and the pointer to the address where the next item is stored is known as a **cons-cell pair**. Functional programming languages were to copy IPL in making lists the central data

[4] Simon (1991), p 212.

structure in programming. The representation of Newell and Simon using pointers to addresses is the machine basis of the cons representation of a list discussed in chapter 4.

About that time, John McCarthy, then a young mathematician at MIT, was doing preliminary work on a program called Advice Taker that would allow a computer to receive instructions in English. McCarthy knew about IPL and had read Church's work. McCarthy observed that many computable functions could be defined by equations, but only by equations that mentioned that function on both sides of the = sign. One simple example is the function.

factorial(0) = 1
factorial(N) = N $\times$ factorial(N - 1) where N > 0

These equations are not circular (unlike factorial(N) = factorial(N)) because they enable the calculation of a factorial by the calculation of the factorial of the number preceding it. Eventually the simplest case (0) is reached and the computation ends. Such functions are called **recursive** and in McCarthy's new language, recursion was the principle means of defining computable functions. McCarthy also borrowed on use of lists by IPL. But he improved on IPL by adding a reclamation program that allowed the computer to reclaim memory when it was running short, by marking out the symbol structures that the computer no longer needed. This process became known as **garbage collection**.

The resulting language, called **Lisp**[5] (McCarthy 1960), was remarkable. Lisp programs were themselves written as lists of symbols, which meant that Lisp programs could be written which would write Lisp programs. A Lisp program could even modify itself while it was running, or receive as part of its input some other Lisp program and apply it to another part of its input.[6]

The important advance of Lisp was that it freed the programmer from having to consider the architecture of the computer. By allowing the programmer to define algorithms as functions, and abstracting away from the need to consider allocating or reclaiming memory, Lisp left the programmer free to work on the problems that interested him. To give an

[5] Short for LISt Processing Language.

[6] This is called **higher-order programming** and was implicit in Church's lambda calculus; we study this technique in chapter 5.

analogy, Lisp was transparent as regards the architecture of the machine on which it ran: writing Lisp code was like looking at the problem through a clear window. Writing in a procedural language like assembler requires not only thinking about the problem, but also thinking about the architecture of the underlying machine which is explicit in assembly language. Looking at a problem through assembler is like looking at it through a stained or dirty window, or trying to see to the bottom of a pond on which there is a lot of surface glare and reflection.

So it turned out that the very feature of functional programming that has meant Church's ideas had languished in the shadow of Turing, that is the lack of a specific machine for executing computations, was also its greatest strength! But it was strength purchased at a high price. The structure of a procedural language is dictated by the design of the digital computer, which works by executing a series of commands and shifting data from one address to another. In a programming language like assembler, which reflects the internal architecture of the computer, programs are likely to run quickly. In contrast, a language that ignores the architecture of the machine needs a very sophisticated compiler to relate the language to the computer.

In the early years of Lisp, such sophisticated compilers did not exist. Moreover the garbage collection process was itself expensive, causing the computer to hang for vital seconds while memory was reclaimed. Even the list processing feature, one of the most important ideas to come from computer science, was costly because a lot of the computer's memory was tied up in storing pointers from one address to the next. Functional programming acquired an immediate reputation for inefficiency that confined it to research labs for another 20 years. In the value scales of the '60s, when machines were slow and CPU time was expensive, inefficiency was a serious charge.

By the early '70s, Lisp programmers recognised that the performance lag in Lisp programs was a serious bother, and that the foundation of the problem was the distance of the architecture of the computer from the actual Lisp. Clever compilation was one answer and this was done with versions of Lisp like **MacLisp** (developed for the DEC-10 mainframe); but another was to design the architecture of the machine around the Lisp language. These machines became famous as the **Lisp machines**.

In the context of their times, the Lisp machines were startlingly innovative, and they introduced many ideas that have since become part of high street computing; such as windows and the mouse (developed for the Xerox Lisp machine). The first serious Lisp machine was **CADR,** which anticipated the personal computer years before the IBM XT.[7] High tech companies like Symbolics and Lisp Machines Incorporated mushroomed around research into Lisp machines and larger companies like Xerox and Texas Instruments also fielded their own Lisp machines.

But Lisp machines were in the end to fail. At the time of writing, no manufacturer is engaged in the production of Lisp machines. Ironically, these machines were to operate to the detriment of the spread of Lisp as a commonly used programming language. By choosing to work on Lisp machines, whose cost confined their use to favoured researchers in well-funded laboratories, the Lisp machine community isolated their work from mainstream computing. Ordinary programmers could not afford to run programs that used 10Mb of memory, when 1Mb was seen as a lot. While Lispers worked in ivory towers, always looking to the future, procedural programmers working in **C** (another product of the '70s) concentrated on developing programs that ran on shoebox machines. By directing their efforts to standard computer architecture, the procedural programmers established a near unchallengeable dominance on conventional machines.

Stimulated by the investment of billions of $ that the Lisp machine vendors did not have, these shoeboxes were eventually to surpass the performance of the dedicated Lisp machine. By 1990, conventional workstations running efficient Lisp compilers could equal the performance of Lisp machines. By 1996, the humble PC could rival the performance of the workstation. Lisp machine vendors went bankrupt.

Lisp itself survived and in 1984 Lisp was standardised into **Common Lisp**. This had the advantage of providing a common language standard for Lisp implementers to work towards and one that was largely downwards compatible with several existing Lisp dialects. The disadvantages were that the resulting language was very large and many of the bad features of Lisp, that McCarthy himself later acknowledged were in the language, were frozen in the Common Lisp language definition. Across the Atlantic, British computer scientists like David Turner were finding fault with Lisp and British computer science was making its own contribution to the history of functional programming.

[7] At the top of CADR's microcode listing was a quote from the rock opera ***Tommy*** "Here comes a man to bring you a machine all of your own".

1.4 The British Experience

One of the earliest workers in functional programming this side of the Atlantic was **Peter Landin** who in 1964 published a paper describing a method for mechanically evaluating lambda calculus expressions on a computer. The design, called the **SECD machine** (short for Stack-Environment-Control-Dump) influenced many later compilation strategies, including the **Functional Abstract Machine** (**Cardelli**, 1983) that was used to compile Standard ML.

Lisp was designed as a functional language that supported a model of computation called **eager evaluation**. In eager evaluation, all the inputs to a function are first evaluated before the function is set to work computing the final answer. Sometimes this is not a sensible strategy; a simple example is the problem of computing f(factorial(100)), where f is defined as f(X) = 0. Since f always returns 0, no matter what the input, computing the value of factorial(100) is a waste of time. In an eager language like Lisp, factorial(100) would be computed, but in a language that uses **lazy evaluation** it might not. The defining characteristic of lazy evaluation is *procrastination* - only evaluate when you have to. **SASL**, developed by **David Turner** (1976, 1979), was a functional programming language that was driven by lazy evaluation.

Automated deduction had already provided an important input into functional programming through Newell and Simon's work on IPL. In 1979, it provided another. In that year **Milner, Gordon** and **Wadsworth** published their work with the **Edinburgh LCF** marking an important development in both functional programming and automated reasoning. In 1976, Milner had published a paper describing how functional programs could be proved free of type errors in a way that was entirely mechanical. A type error is an error that arises when a function is applied to an input that it cannot compute (the multiplication of two strings is an example).

To accomplish this, Milner used the **unification algorithm** developed for automated reasoning by **Robinson** in 1965. Using Milner's algorithm (called the $\mathcal{W}$ algorithm), a proof that a program was free from type errors could be executed by a computer. This was an important advance in the production of reliable software and in their 1979 publication, Milner, Gordon and Wadsworth introduced the first functional programming language to incorporate this new technology - the **Edinburgh MetaLanguage** or **Edinburgh ML**. Since Edinburgh ML, nearly every new functional programming language has incorporated some form of type checking and these languages form the class of what is called **statically**

typed languages. In chapter 13, we study the application of Robinson's unification algorithm to type checking.

Edinburgh ML became upgraded and standardised as **Standard ML**. An important addition to the upgraded version was **pattern matching**, which was an old idea that went back as far as **SNOBOL** that was invented in Bell Labs in 1962. SNOBOL was a text processing language that allowed the programmer to isolate pieces of text by searching for patterns. Since all functional languages exploit the use of lists, it is much easier to use patterns to isolate the needed elements than to write functions which search for them. The use of **pattern-directed programming** is common in functional programming today and is used throughout this book.

The '80s in Britain were a very fertile time for the implementation of new models for compiling functional languages. This area is rather outside the ambit of this book, but we shall mention **David Turner's** (1979) paper, which suggested that **combinatory logic** could be used to implement functional languages as an alternative to lambda calculus. In many ways, combinators are even simpler than lambda calculus. Following Turner, scientists experimented with new models for compiling functional programming languages; **combinators, supercombinators, graph reduction**, and **dataflow machines** were researched in this decade. The result was that the performance gap between functional and procedural languages was narrowed in that decade.

But while implementers of functional languages poured their efforts into building better compilers and new functional programming languages such as **Haskell**, procedural programming proceeded to capitalise on its lead by expanding into new application areas. An important step in the spread of the procedural paradigm was the invention of the language **C** by **Dennis Ritchie** and **Ken Thompson** in 1973. Ritchie and Thompson were working in Bell labs on the first release of the **UNIX** operating system and they needed a fast, portable programming language that was close to assembler but not tied to specific machines. They designed C - a functional programmer's bad dream. Riddled with assignments, and clinging close to the architecture of the computer, C programs run extremely quickly. The success of the UNIX operating system allowed C to spread to nearly every computer, and its good performance meant that programmers took it up as the language of choice when writing CPU-intensive programs for machines of limited power.

The success of the procedural paradigm at a time when machines were too slow to support the functional approach, has left computing with a culture that is recognisably procedural. UNIX and **Windows XP** are built on the foundations of C, and the shell languages that support these systems are recognisably procedural. The **X-windows** system is likewise C-based and the paradigms for building graphical user interfaces are based on procedural languages like **Visual Basic**. **Java** has emerged as a leading language for building **Internet** applications, and its antecedents are C and **C++.** The 21st century programmer expects access to graphical interface builders, numerical packages, debugging tools, Internet capability, electronic mail, multi-user capability, sound and graphics to be within reach inside his chosen medium. Contemporary functional languages, for all their mathematical sophistication, still present the user with a cursor and a command line that is recognisably '70s in style.[8] The 20 year gap is still evident.

For believers in the functional paradigm, the remedy is more investment of effort and a broader perspective. Functional programming remains under-utilised and misunderstood. There is a need for publicly available functional models of all these things that present the beginner with an attractive environment in which she can have fun. Calvinistic arguments that austerity is good for you, cut no ice with novices who migrate away from functional languages when they cannot do the things they want. There is some evidence to suggest that, rather than concentrating on closing the gap, researchers are concentrating on digging into the foundations of the subject. There is a danger that the area will become introspective; the hobby of a few intense theorists who teach only to select students. This should not be allowed to happen.

1.5 SEQUEL – the predecessor of Qi

The history of Qi began in 1989, when I was working as a research assistant at the LFCS in Edinburgh. At that time, I worked in Lisp, and to improve my productivity, I wrote a 700 line Lisp program which translated pattern-directed functional code into reasonably efficient Lisp.

While cycling in the Lake district in the spring of 1990, I had the insight that since type-checking was inherently deductive in nature, the type discipline of a language could be specified as a series of deduction rules. Using the techniques of high-performance theorem-proving, these rules could be compiled down into an efficient type checker. The type-discipline would be free from the procedural baggage of low-level encoding and presented in a form that was clear, concise, and accessible to experimentation and extension by researchers in type theory.

[8] The Qi/Tk project is one attempt to rectify this situation. See appendix I.

This was a bold conception that seemed eminently reasonable, but which in fact took a further 15 years of development to realise completely.

The prototype development was the language SEQUEL (**SEQUE**nt processing **L**anguage), written in Lisp and introduced publicly in 1992 and to the International Joint Conference on Artificial Intelligence in 1993. SEQUEL anticipated many of the features of Qi, in particular the use of Gentzen's sequent notation to formulate the type rules. SEQUEL actually contained the type theory for a substantial portion of Common Lisp (over 300 Common Lisp system functions were represented in SEQUEL) necessitating a sizeable source code program of more than 23,000 lines. SEQUEL was not only designed to support type-secure Common Lisp programming, it was also supposed to support direct encoding of theorem-provers by giving the user the power to enter logic rules in sequent notation to the SEQUEL system.

SEQUEL sustained Andrew Adams's (1994) reimplementation INDUCT of the Boyer-Moore theorem-prover. His project was written in 6,000 lines of SEQUEL, and generated nearly 30,000 lines of Common Lisp, and gained him a distinction. In 1993, state-of-the-art was a SPARC II, so power was limited. Loading and type checking INDUCT took several minutes.

1.6 The deficiencies of SEQUEL

SEQUEL was a compromise of the ideals of deductive typing for several reasons.

1. Large parts of the implementation were still procedurally encoded for speed.
2. Since SEQUEL was heavily configured to support Common Lisp, the language was not consistent with lambda calculus (since Common Lisp does not support lambda calculus features like currying and partial applications).
3. SEQUEL inherited case-insensitivity from Common Lisp and the use of the empty list NIL to mean false.
4. SEQUEL lacked a formal semantics.
5. SEQUEL also lacked a proof of type correctness.
6. SEQUEL was a system for generating theorem-provers from logic specifications, but it also used unification in interpreting these specifications, which was not always type secure.
7. The use of occurs-check unification in SEQUEL theorem-provers meant that the theorem-provers ran more slowly than needed even if they were only simple propositional theorem-provers.

The whole system was large because, not only did SEQUEL attempt to capture the Common Lisp type system, but it also had two compilers. The first was a compiler from SEQUEL code into Lisp (inherited from earlier work); the second was a compiler for translating sequent rules into Horn clause logic and Horn clause logic into Lisp. Over ten years and two countries all these problems were eliminated.

1.7 The Evolution of Qi in Britain

Until 1996, SEQUEL stayed in operation with minor amendments. Most of my creative time from 1993-1996 was spent writing poetry. SEQUEL continued to support final year projects, but late 1996, I returned to computer science research and changes began to the language that finally transformed it to Qi.

The first modest change was to introduce case-sensitivity and the use of proper booleans in place of T and NIL. Large parts of the implementation were rewritten, reducing the source code by several thousand lines.

The next stage was to replace the Horn clause compiler itself. This was done in two steps, first a compiler was designed that compiled sequent rules into efficient type secure SEQUEL code. Since SEQUEL was a pattern-matching language, the performance of theorem-provers jumped sharply by an order of magnitude. This eliminated the unification problem. Andrew's INDUCT was 10 times slower than Boyer-Moore's NQTHM, but in the nascent Qi, parity was attainable.

This was a very important advance and it lead to the bootstrapping approach of Qi - the type theory of Qi could now be specified as a series of sequent rules that Qi could compile into a practical type checker.

In 1998 overloading in SEQUEL was dropped and the entire list of function types was placed on a hash table. This was in a sense a move away from Lisp and towards a cleaner model and a smaller language. The plus was the elimination of 10,000 lines of code.

In late 1997, I applied for three years unpaid leave from my job as a lecturer in order to concentrate on finishing this work. One year was granted, and from 1998 to 1999 a lot of work was expended in providing SEQUEL with a formal type theory. After a false start, the current type theory was evolved in 1999 and the semantics was developed between 1999 and 2000.

During that same period, a lot of work was done in testing whether deductive typing was really a practical option. Early results were not encouraging, 30 line programs could take as many seconds to type check. The goals of deductive typing - complete declarative specification and a type-checking procedure that was conceptually distinct from the rules that drove it - meant that performance-enhancing hardwired hacks were not allowed. Eventually a technology was developed and the performance benchmarks showed that a 166MHz Pentium under CLisp could just about run the system. At the time of writing, 3.8 GHz machines are on sale, which will no doubt make these old challenges into programming molehills.[9]

The implementation went through two complete rewrites. The first rewrite was very thoroughgoing and introduced partial applications and currying into the language, making it lambda calculus consistent.

In 2001 a robust version of a compiler-compiler (first devised by my colleague Dr Gyuri Lajos) was built and used to encode some of the complex parsing routines needed in Qi. This was the compiler-compiler **Qi-YACC** which was later used in both Qi and Qi/Tk. I also began experimenting with fusing Qi with TCL/Tk, John Ousterhout's graphical and scripting language using Lisp code from Matthias Lindner's Plopp system. In 2002 Qi/Tk was used to present a moving histogram of a transport model.

1.8 The Evolution of Qi in America

In 2002, I left the UK for America, bearing Qi 1.4 with me. Qi 1.4 was used to build almost the whole of Qi 2.0 which was a clean rebuild of the whole system. Qi 2.0 was never released however, because of an improvement to the type checking algorithm **7** used in Qi 1.4.

In 2003, I developed the experimental algorithm **7** * and gained a factor of 7 speedup over 1.4. In late 2003 I put the finishing touches on a correctness proof for the type theory and algorithms **7** and **7** * of Qi. In 2003 Qi won me the Promising Inventor Award from the State University of New York. Carl Shapiro, currently of SRI, attended my functional

[9] It is worth noticing though, that a 166MHz Pentium is about the slowest machine that could feasibly run Qi 1.4 and that this machine appeared about 1997. Therefore the goals of deductive typing that were formulated in my 1990 paper were about seven years short of having the technology they needed to work. The Qi/Tk system needs at least a 500 MHz machine and 1Ghz is better. These machines did not appear until 2000 and 2001; nearly ten years after the first SEQUEL paper.

programming class at Stony Brook that same year. He was to exert a significant influence on development of Qi.

In 2004, Qi 2.0 was revised to work with **7*** and the result was Qi 3.2. After some more debuggings and revisions, both Qi and Qi/Tk 4.0 were beta released in November 2004. Carl was a Lisp enthusiast and thanks to his code, Qi/Tk 4.0 could run graphics and the read-evaluate-print loop in tandem. Carl also did some tricky debugging of the Lindner's code which, because of a write problem in TCL, could cause Qi/Tk to hang unexpectedly .

Carl's interest in SEQUEL made me revisit my old work. In 2005 I constructed a new model for compiling Horn clauses called the **Abstract Unification Machine**. The AUM compiles Prolog into virtual machine instructions for a functional language. AUM technology gave 200-400 KLIPS under CLisp and quadrupled the speed of the type checker. The beta version, Qi 5.0, used a prototype AUM technology. Qi 6.1, which used the AUM, had some new features; including the **commit!** and mode declarations feature mentioned in chapters 11 and 13. The former was modelled after the Prolog cut, and the judicious introduction of cuts into the type checker improved performance again.

Qi 6.1 was also the first version of Qi to use abstract datatypes and the **preclude** and **include** commands for managing large type theories (see chapter 11). Qi 6.1 was released in April 2005, just before the publication of *Functional Programming in Qi* and at the same time as the creation of **Lambda Associates**. At 6,500 lines of machine-generated Lisp, Qi 6.1 is three times smaller than the SEQUEL of 1993. With the encouragement of David Hogg, I placed Qi, Qi-Prolog, and Qi-YACC under the GNU licence and placed the source online.

Further Reading

A discussion of the Turing machine and its properties is found in **Boolos, Burgess** and **Jeffrey** (2002). The lambda calculus is discussed in chapter 12 of this book, and references can be found at the end of that chapter. **McCarthy** (1960) describes the genesis of Lisp. **Backus** (1978), in his famous Turing award lecture, argues for the declarative paradigm, criticising imperative languages as "fat and weak". **Turner** (1982) develops the case for functional programming. **Gabriel** (1990) has an interesting discussion about why declarative programming (in the shape of Lisp) has not displaced the procedural paradigm. Introductions to functional programming and functional programming theory can be found in **Bird** and **Wadler** (1998) and **Field** and **Harrison** (1988). Introductions to Lisp are provided in **Winston** and **Horn** (1989) and **Schapiro** (1986). **Abelson** and **Sussmann** (1996) is an excellent introduction to functional programming in

Scheme - a dialect of Lisp. **Shrobe** (1988) is a good review of the history of the Lisp machines just before their demise. **Wikstrom** (1987) and **Paulson** (1996) introduce SML. An overview of Miranda^{TM} is given in **Turner** (1990) and **Thompson** (1995) is a book length introduction to Miranda^{TM}. **Tarver** (1990) reviews some of the earliest thoughts that led to Qi. **Tarver** (1993) was an early principle publication of this approach. **Haskell** from the University of Glasgow is a recent and popular addition to the functional programming family, **Thompson** (1999) is a good introduction.

Web Sites

The life and work of Alan Turing is beautifully presented in a web site dedicated to that purpose; **Andrew Hodges' The Alan Turing Home Page** (http://www.turing.org.uk/turing/) contains a wealth of information. The University of Arizona (http://www.math.arizona.edu/~dsl/tmachine.htm) provides links to several sites containing downloadable software and applets for simulating Turing machines. http://cm.bell-labs.com/cm/cs/who/dmr/chist.html offers an account of the development of C. **The Association of Lisp Users** (http://www.lisp.org/table/contents.htm) provides an extensive web site for Internet sources on Lisp including links to original papers on the history of Lisp. **New Jersey SML** offers a free high-performance ML (http://www.smlnj.org/). http://www.haskell.org is the web address for all things Haskell.

Part I: the Core Language

2 The Qi Top Level

2.1 Starting Up

If Qi has been installed according to the installation instructions then the Qi top level looks as in figure 2.1

```
Qi, Copyright (C) 2000-2005 Mark Tarver
Version X

(0-)
```

Figure 2.1 The Qi *top level*

The integer prompt shows that you are in the **Read-Evaluate-Print Loop**. Functional programming languages interact with the user at this level. The purpose of this loop is to receive the expressions that you enter, to evaluate them, and to print a response.[10] These expressions can be of various kinds. The simplest expressions that Qi evaluates are the **self-evaluating expressions** that evaluate to themselves. There are five kinds of self-evaluating expression.

1. **Numbers:** this includes **integers** (**-3, 78, 45**), **floating point numbers** (**2.89, 0.7**), **rational numbers** (**1/3, 8/9**) and **complex numbers** ($\sqrt{-1}$).
2. **Symbols**: this includes any unbroken series of symbols (except booleans, numbers, strings and characters, see below) such as **mynameisjack, catch22.**
3. **Booleans**: **true** and **false**.
4. **Strings**: a string is any series of symbols enclosed between a pair of double quotes such as **"my name is jack"**, **"catch 22"**. Notice that strings permit spaces whereas symbols do not.

[10] The symbol **^** within any input, followed by carriage return, will abort the current line input.

5. **Characters**: this is a small set of self-evaluating expressions which includes **#\a, #\b,....,#\z, #\A, #\B, ...,#\Z, #\0,.....#\9.**

An expression is evaluated by typing it to the Qi top level and hitting the return key. Figure 2.2 shows some self-evaluating expressions entered to Qi.

```
(0-) 9
9

(1-) "foobar"
"foobar"

(2-) hello sailor
hello
```

Figure 2.2 Some self-evaluating expressions typed to the top level

The last input in figure 2.2 included an extra expression, **sailor**, which Qi ignored. Only one expression is evaluated at a time, so only the first input is evaluated. Every time an expression is evaluated, the Qi top prompt reappears with an integer in parentheses. Typing **(quit)** will exit the top level, and this terminates a session in Qi.

2.2 Applying Functions

Since typing self-evaluating expressions is not an exciting pastime, we will consider how to apply functions to inputs in Qi. There are three principal ways of writing functional expressions.

1. **Infix**; where the sign for the function comes between the bound variables. This is the usual practice in arithmetic where we write "2 – 1"; the minus sign comes between the "2" and the "1".
2. **Prefix**; the sign comes before the bound variables. So in prefix "2 – 1" would be written "- 2 1".
3. **Postfix**; the sign comes after the bound variables. So in postfix "2 – 1" would be written "2 1 –".

Qi functions are usually written in prefix form.[11] To apply a function f to inputs i_1,...,i_n, we enter (f i_1,..,i_n) using round brackets; (figure 2.3).

[11] Functions can be written to Qi in infix form (e.g. (+ 1 1) can be written as (1 + 1)) using the macroexpand function (see appendix B).

```
(0-) (+ 1 2)
3

(1-) (+ (* 3 4) (+ 1 1))
14
```

Figure 2.3 Some simple applications typed to the top level

There are several ways of evaluating expressions in functional programming languages; one widely used model is **applicative order evaluation**, which is the one Qi usually uses. The evaluation procedure of applicative order evaluation is to first evaluate (from left to right) the inputs to a function before applying the function to the results of this evaluation. Thus in the final case of figure 2.3, Qi first evaluates **(* 3 4)** to **12** and then **(+ 1 1)** to **2** before adding the two together. The result **14** is the **normal form** of the expression **(+ (* 3 4) (+ 1 1))**; that is, the result produced when the evaluation has completed itself without any error being raised. The **arity** of a function is the number of inputs it is designed to receive. In the case of * and + this is 2, and these are referred to as **2-place** functions. Functions like * and + that are built in to Qi are called **system functions**.

2.3 Repeating Evaluations

! is used to repeat evaluations and this feature is tied to the numbering within the Qi prompt. Typing ! followed by an integer will repeat the input numbered by that integer. !! repeats the last input. ! will also locate inputs via the leading symbols used in them; to do this we type in !, following it with the symbols in question. The most recent input matching the symbols following ! is called up, and Qi evaluates this expression. It is not always necessary type in the entire name of the function to call up this expression. Qi will also match on the most recent input which begins with those symbols placed after !.

```
(0-) (sqrt 9.0)
3.0

(1-) !s
(sqrt 9.0)
3.0

(2-) !!
(sqrt 9.0)
3.0
```

Figure 2.4 Using ! *to repeat evaluations*

The % command will print (without evaluating) every previous expression whose prefix matches what follows %. % on its own will print off all functional expressions typed to the top level since the session began. Where *n* is a natural number, %*n* will print off the *n*th expression typed to the top level since logging in to Qi.

```
(4-) %s
0. (sqrt 9.0)
1. (sqrt 9.0)
```

Figure 2.5 Using % to print past inputs

2.4 Input/Output

All programming languages need a facility that allows a program to halt execution and receive the user's input. The function **input** returns the result of evaluating whatever the user types to it.

```
(7-) (input)
(* 7 8)
56
```

Figure 2.6 Using **input** *to interact with an evaluation*

print receives an input, evaluates it, prints the normal form as a **side-effect**, and returns the normal form. The function **output** receives a string, prints the message as a side-effect, and returns the value **"done"**. The **output** function has a number of features to help produce formatted messages. One of these is **~%**, which forces a new line (figure 2.7).

```
(9-) (print [(* 6 5) is 30])
[30 is 30]

(9-) (output "goodbye cruel world,~%I bid you adieu.~%")
goodbye cruel world,
I bid you adieu.
"done"
```

Figure 2.7 Printing a message using print and output

~A creates slots in the message, which can be filled in any way by placing expressions after the message string. The expressions are evaluated to their normal forms, and these normal forms are placed in the slots of the message and the result is printed. Figure 2.8 shows an example.

```
(9-) (output "~A in ~A ~A made ~A ~A.~%" God his wisdom the fly)
God in his wisdom made the fly.
"done"
```

Figure 2.8 Filling the slots in a message

The function **make-string** works like **output,** except that the message string is returned as a result and not printed. A special kind of message is an **error message**. If Qi receives an error message, then the message is printed and the computation is aborted. No value is returned. Both **error** and **make-string** share the formatting conventions as **output.**[12]

```
(10-) (make-string "~A in ~A ~A made ~A ~A.~%"  God his wisdom the fly)
"God in his wisdom made the fly."

(11-) (error "here is an error message! ~A" "Weird bug!")
error: here is an error message! Weird bug!
```

Figure 2.9 Building strings and printing errors

2.5 Strict and Non-Strict Evaluation

All computer languages have some provision for detecting conditions operating within the program and then diverting the control of the program depending on whether these conditions are realised. The classic example is the conditional expression if ... then ... else ... which in one form or another is found in every computer language. Use of conditional expressions requires the use of **boolean expressions** where a boolean expression is one that evaluates to either true or false.

The if ... then .. else ... construction is found in Qi as a 3-place function **if** that receives

1. A boolean expression X.
2. An expression Y whose normal form is returned if X evaluates to **true**.
3. An expression Z whose normal form is returned if X evaluates to **false**.

Though this is all quite simple, it is important to know that conditional expressions are not evaluated using applicative order evaluation. The reason why is easily brought out by example. The 2-place function = returns **true** if the normal forms of its two inputs are the same, and **false** if

[12] Note for Lisp programmers; output and make-string are based directly on the FORMAT command in Lisp and have the same facilities. See Steele (1990) for details.

are they are different. Suppose we evaluate **(if (= 1 0) (* 3 4) 3)** by applicative order evaluation.

(if (= 1 0) (* 3 4) 3)
⇒ **(if false (* 3 4) 3)**
⇒ **(if false 12 3)**
⇒ **3**

The reduction of **(* 3 4)** to **12** in the penultimate step is quite unnecessary, because the result of evaluating **(if false (* 3 4) 3)** cannot depend on the evaluation of **(* 3 4)**. A better evaluation strategy evaluates **(= 1 0)** to its normal form, and then evaluates the appropriate expression.

This latter form of evaluation is an example of **non-strict evaluation.** A strict evaluation strategy requires that every input to a function to be evaluated before the function is applied to the results. Thus, in a non-strict evaluation strategy, it is possible to return a normal form from an expression *E* even when there is a subexpression of *E* that has no normal form. Functional languages use non-strict evaluation for the evaluation of conditionals. For example, the expression **(if (= 1 0) (+ a a) no)** evaluates to **no** in Qi, though the evaluation of **(+ a a)** will raise an error.

2.6 Boolean Operations

A **boolean operation** is a function that receives booleans as inputs, and returns a boolean as a result. The function **and** is a boolean operation which on receiving 2 boolean inputs **X** and **Y**, returns **true** if the normal forms of **X** and **Y** are **true** and returns **false** otherwise (figure 2.10).

```
(0-) (and (= 1 1) (= 2 2))
true

(1-) (and (= 1 2) (= 2 2))
false

(2-) (and true 4)
error: 4 is not a boolean
```

Figure 2.10 Using **and**

Again **and** expressions are not evaluated by applicative order evaluation. In the second input in figure 2.10, since **(= 1 2)** is evaluated to **false**, the value of **(= 2 2)** is immaterial.

The function **or** receives 2 boolean inputs and returns **true** if at least one evaluates to **true**, and returns **false** otherwise. For similar reasons, applicative order evaluation is suspended here too.

```
(3-) (or (= 1 0) (= 2 2))
true

(4-) (or (= 1 0) (= 2 1))
false
```

Figure 2.11 Using **or**

not returns **true** if its input evaluates to **false** and **false** if its input evaluates to **true**.

```
(6-) (not (= 1 1))
false

(7-) (not (= 1 2))
true
```

Figure 2.12 Using **not**

All other Qi function applications are evaluated by applicative order evaluation, which is a strict evaluation strategy.

2.7 Defining New Functions

Equational specifications are a simple way to begin to understand how functions are defined in Qi. For instance, the equation *return-b*(*a*) = *b* defines a function *return-b* that receives *a* and returns *b*. The domain of *return-b* is the set {*a*} and the range of *return-b* is the set {*b*}. Both the range and the domain are finite in size. If we want to add new elements to the domain and range then we add new equations. Often a function can represented to Qi in the manner of a series of equations. The syntax is slightly different from the one used by mathematicians. First, the = is replaced by a -> and the bracket (is placed before the function symbol, so the first equation would appear more as:-

```
(return-b a) -> b
```

Second, all those equations that relate to the same function are grouped in brackets.

```
((return-b a) -> b
 (return-b c) -> b)
```

Third, since all the equations bracketed together must relate to the same function, it is tedious to keep typing the name of this function for each such equation. The name of the function is instead given at the top of its definition. The entries **a -> b** and **c -> b** are **rewrite rules** within the Qi function definition (we shall see shortly that they differ a little from equations). Figure 2.13 shows how **return-b** is defined and used in Qi.

Entering the definition of **return-b** causes Qi to compile it into its **environment**. The environment can be considered to be a set of function names and their associated definitions. Once the definition of a function has been entered to the Qi environment, Qi will allow the use of this function in evaluation.

```
(15-) (define return-b
          a -> b
          c -> b)
return-b

(16-) (return-b c)
b

(17-) (return-b d)
error: partial function return-b
Track return-b ?  (y/n) n
```

Figure 2.13 Defining and using a simple function in Qi

Supplying an input for which there is no covering rule generates an error message. This error message is accompanied by an offer to track the function; this is used for debugging purposes (see appendix I). But, suppose we want to say that *whatever* the input, *return-b* will return b. Since the domain of the function is now infinite in size, we cannot produce an equation for every possible input. Mathematicians use **variables** to cope with cases like this. In Qi, a variable is any symbol beginning with an uppercase letter with the exception of **T** and **NIL**.[13] Using variables, one equation can state that whatever the input, return-b will return b.

return-b(X) = b

[13] T and NIL are used internally by Lisp (the language in which Qi is implemented). NIL is an alternative representation of the empty list (see chapter 4).

If we remedy the definition of **return-b** by using the definition corresponding to the equation *return-b*(X) = b, then Qi overwrites the old definition.

```
(19-) (define return-b
          X -> b)
return-b

(20-) (return-b d)
b
```

Figure 2.14 Overwriting an older definition

2.8 Equations and Priority Rewrite Systems

Equational specifications are a useful way of getting to grips with the task of defining functions. But there are significant differences between a set of equations and a series of rewrite rules in an Qi function definition. Consider the following equations.

$$f(a) = b$$
$$f(a) = c$$

Entering this directly into Qi produces the following definition.

```
(define f
  a -> b
  a -> c)
```

Evaluating **(f a)** produces **b** and never **c**. The second rewrite rule **a -> c** behaves as if it did not exist; the reason being that Qi tries rewrite rules from top to bottom. First **a -> b** is used and then **a -> c**. As soon as **a -> b** is tried with the input **a**, of course **b** is returned. The second rewrite rule is irrelevant and is **starved** by the first rule. This feature, of ordering rewrite rules so that some are tried in preference to others, is a characterising feature of **priority rewrite systems**. If we wanted to show that **(f a)** was identical to **c**, we could easily do this using the equations above, but in Qi **(= (f a) c)** would evaluate to **false**. We could amend the definition of **f** to allow **(= (f a) c)** to be **true** as follows.

```
 (define f
   a -> c
   b -> c)
```

The amended definition now gives **(f a)** and **c** the same normal form. Nevertheless, the price for achieving this is that we have abandoned a simple transcription of the original equations. The problem of fairly transcribing a set of equations into rewrite rules has been intensively studied and it is known that there is no decision procedure for deriving a fair representation of arbitrary set of equations.[14]

Equations are, in a sense, more powerful and less controlled than rewrite rules. However, the same sensitivity to ordering gives Qi capacities that a purely equational specification does not have. For example, suppose we want to define a function *identical* that returns true given two inputs with the same normal form, and returns false otherwise. Using our equational specification approach, we write:-

identical(X,X) = true

This is fine for the positive case. The variable X has been used twice indicating that the two inputs must be the same for true to be returned. But what of the negative case? We cannot write

identical(X,Y) = false

This says that for *all* X and Y, identical(X,Y) returns false. If we want express this function using equations, we have to expand our vocabulary and allow ourselves to use **conditional equations**.

identical(X,Y) = true if X = Y
identical(X,Y) = false if X $\neq$ Y

Qi's priority ordering allows **identical** to be defined in terms of rewrite rules.

```
(define identical
  X X -> true
  X Y -> false)
```

Since the rule **X X -> true** is tried first, any identical inputs must return **true**. If the second rule is tried, it can only be because the first rule failed, in which case the output must be **false**.
Another significant difference between rewrite rules and equations is that rewrite rules, unlike equations, must obey the **variable occurrence**

[14] Meaning by "fair", a set R of rewrite rules, so that a = b follows from the equations when and only when both a and b have unique normal forms a′ and b′ under R, such that a′ and b′ are syntactically identical. There is a semi-decision procedure, the **Knuth-Bendix procedure** (see Further Reading), which, when it terminates, will deliver a fair set of rewrite rules from a set of equations.

restriction. The variable occurrence restriction requires that a variable that appears on the right-hand side of a rule must appear on the left-hand side. So the equation f(X,Y) = g(X) cannot be oriented into the rewrite rule **(g X) -> (f X Y)**. Qi definitions that violate this restriction produce a **free variable** warning (figure 2.15).

```
(0-) (define g
        X -> (f X Y))
warning: free variable Y.
```

Figure 2.15 A function definition that violates the variable occurrence condition

Since free variables often result from clerical errors, Qi accepts the definition[15] but warns that **Y** is free. In this case, the variable **Y** is passed as a symbol to the function **f**. Thus if **f** is defined as

```
(define f
  A B -> B)
```

then for all values of **X**, **(g X)** will evaluate to the symbol **Y**.

2.9 A Fruit Machine Program

A simple Qi program simulates a fruit machine with three wheels.🗀 We suppose that there are five different kinds of fruit; cherries, pears, oranges, pineapples and lemons. Any combination of two in sequence, like two cherries followed by anything will score, and of getting all three the same will score even more. Some fruits are harder get than others; lemons are harder to get than pineapples which are harder to get than oranges, which are in turn harder to get than pears, with cherries the easiest of all.

Value	*Fruit*
0,1,2,3,4	cherry
5,6,7,8	pear
9,10,11	orange
12,13	pineapple
14	lemon

Figure 2.16 Mapping numbers to fruits

[15] An alternative would be to raise an error, which would be logically consistent with rewrite languages. However there are occasions where we want to be able to return variables or (e.g.) proper names, where a strict enforcement of the variable occurrence condition would be a hindrance. The strong-warning function (see appendix A) does enforce error raising here.

🗀 Qi Programs/Chap2/fruit machine.qi.

Written as a Qi function the associations in figure 2.16 appear in figure 2.17.

```
(define return-fruit
  0 -> cherry
  1 -> cherry
  2 -> cherry
  3 -> cherry
  4 -> cherry
  5 -> pear
  6 -> pear
  7 -> pear
  8 -> pear
  9 -> orange
  10 -> orange
  11 -> orange
  12 -> pineapple
  13 -> pineapple
  14 -> lemon)
```

Figure 2.17 Expressing number-fruit mappings in Qi

To spin the wheel, we generate a random number from the interval 0-14 and apply the function **return-fruit** to it. **(random 15)** is used to generate a random number from 0-14 and then **return-fruit** is applied to that number to return a fruit. (figure 2.18). The function **payoff** relates combinations of fruit to payoffs[16].

```
(define payoff
 cherry cherry cherry -> 60
 pear pear pear -> 100
 orange orange orange -> 200
 pineapple pineapple pineapple -> 300
 lemon lemon lemon -> 500
 cherry cherry X -> 10
 X cherry cherry -> 10
 pear pear X -> 20
 X pear pear -> 20
 orange orange X -> 30
 X orange orange -> 30
```

[16] Notice that in the definition of payoff the line cherry cherry cherry -> 60 comes before cherry cherry X -> 10 and X cherry cherry -> 10. Changing the order would starve the rule cherry cherry cherry -> 60.

```
 pineapple pineapple X -> 40
 X pineapple pineapple -> 40
 lemon lemon X -> 50
 X lemon lemon -> 50
 X Y Z -> 0)

(define spin-wheel
 -> (return-fruit (random 15)))
```

Figure 2.18 Relating fruit combinations to winnings

We now construct our **top level function fruit-machine** (figure 2.19).[17] Every time **fruit-machine** is invoked, it does 3 things.

1. It spins all three wheels.
2. It announces what fruits come up.
3. It says what the payoff is.

```
(define fruit-machine
   start -> (announce-payoff
              (spin-wheel) (spin-wheel) (spin-wheel)))

(define announce-payoff
   Fruit1 Fruit2 Fruit3
   -> (output "~A ~A ~A~% You win ~A pence~%"
            Fruit1 Fruit2 Fruit3 (payoff Fruit1 Fruit2 Fruit3)))
```

Figure 2.19 The top level function of the fruit machine program

It is possible to type this to the Qi top level, but this is an extremely impractical way to create programs. The text files containing the source code can be loaded using **load**.

```
(29-) (load "Qi Programs/Chap2/machine.qi.")
loaded

(30-) (fruit-machine start)
orange orange cherry
You win 30 pence
"done"
```

Figure 2.20 Loading and running the fruit machine program

[17] The top level function is the function invoked to start the program.

The final line of the definition of **payoff** was **X Y Z -> 0**. There is no testing the nature of the inputs since the output is decided. One certain sign of indifference here is that the variables that occur on the left-hand side of the **->** do not occur on the right-hand side of the **->**. Moreover, all the variables are different on the left-hand side. Instead of using variables like **X Y Z**, Qi allows the use of the **wildcard** _ to mark the position of an input whose nature is of no interest. Given a wildcard, Qi skips the examination of the corresponding input (figure 2.21).

```
(define payoff
  cherry cherry cherry -> 60
  pear pear pear -> 100
  orange orange orange -> 200
  pineapple pineapple pineapple -> 300
  lemon lemon lemon -> 500
  cherry cherry _ -> 10
  _ cherry cherry -> 10
  pear pear _ -> 20
  _ pear pear -> 20
  orange orange _ -> 30
  _ orange orange -> 30
  pineapple pineapple _ -> 40
  _ pineapple pineapple -> 40
  lemon lemon _ -> 50
  _ lemon lemon -> 50
  _ _ _ -> 0)
```

Figure 2.21 Using wildcards

The rule for using a wildcard is:- if a variable occurs on the left-hand side of the **->** only once, and it does not occur anywhere on the right-hand side of the **->**, then it may be replaced by a wildcard.

Exercise 2

1. Write a *single* expression that does the following. It waits for the user to type in a number followed by carriage return; and then another number followed by another carriage return; and then prints **The sum of the two is X** where the place of **X** is taken by the sum of the two numbers entered.

2. Insert wildcards where appropriate in the following functions.

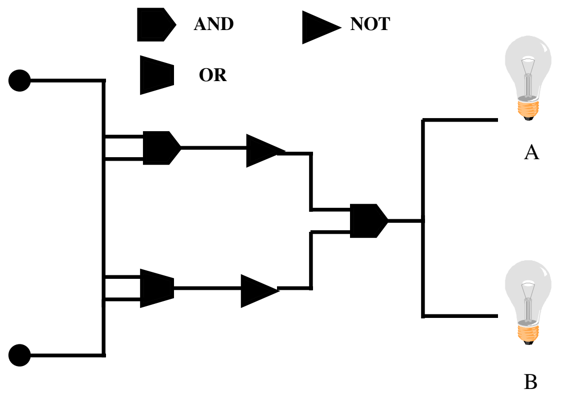
```
(define f
 X X Y -> X
  X Y Z -> (g X Z))

(define g
   X X -> X
   X Y -> Y)
```

3. The Centigrade to Fahrenheit conversion is given by the equation xF^o = (9/5 $\times$ x) + 32C^o. Write a function **cent-to-fahr** that calculates the equivalent Fahrenheit from the Centigrade. Write the inverse function **fahr-to-cent** that calculates the equivalent Centigrade from the Fahrenheit.

4. An **and-gate** is an electronic device that receives two electrical signals that are either high (1) or low (0) and outputs a high (1) signal if and only if both inputs are 1 and outputs 0 otherwise. An **or-gate** is an electronic device that receives two electrical signals that are either high (1) or low (0), outputs a 1 signal if and only if at least one input is 1, and outputs 0 otherwise. An **inverter** is an electronic device that receives an electrical signal that is either high (1) or low (0) and outputs a 1 signal if and only if the input is 0 and outputs 0 otherwise. Write down the Qi definitions of and-gate, or-gate and inverter as functions.

5. Using your answers to question 5, represent the following circuit as a 2-place function so that when 1 is sent down wire A and 1 down wire B, the function produces “Bulb A is lit” and “Bulb B is lit”. (5 marks) Can you simplify this circuit? If so how would you reflect this simplification in your function definition?



6. In neural nets, a neuron is a device that receives a number of numerical inputs $i_1,\ldots i_n$ each of which it multiplies the corresponding weight $w_1,\ldots,w_n$. If the sum

of ($w_1 \times i_1$) + … ($w_n \times i_n$) is greater than a threshold K then the neurone produces a 1 and if the sum is less than K it produces a 0. Design a function that simulates a neurone designed to receive inputs i_1 and i_2, multiplies them by w1 and w2 respectively, and produces an output 0 or 1 depending on K. Your encoding should allow the user to fix the values of w_1 w_2 and K.

7. A spaceman leaves Earth and experiences travel for 10 years at 99.9% of the speed of light. Due to time dilation more than 10 years has passed at home - how much? To calculate the answer you should encode the equation $\text{Time}_{\text{EARTH}} = \text{Time}_{\text{SPACE}} / \sqrt{}\ (1 - V^2)$, where the velocity V is expressed as a fraction of the speed of light. Your encoding should allow the user to query for different values of $\text{Time}_{\text{SPACE}}$ and V.

8. During the Vietnam War, an American pilot jumped from his downed B-52 from a height of 60,000 feet. Encode the formula $d = ½ at^2$, where d = distance travelled, a = acceleration and t = time elapsed, into a function and calculate the distance the pilot fell in 10 seconds. You can take a to be 30 ft/sec^2.

9. *The speed of light is 186,282 miles per second. The nearest star outside our solar system is *Alpha Centauri* which is 4.35 light years away. How long would the following objects take to travel to that star from earth?

   a. A jumbo jet, travelling at 550 mph.
   b. An Apollo rocket travelling at 23,000 mph.
   c. The Voyager probe travelling at 50,000 mph.

   Your program should allow the traveller to fix his destination distance in light years and enter the speed in miles per hour. Your program will print out the time required in weeks, days, hours and minutes.

10. Write a program that receives a number between 1 and 9,999,999 and prints its value in words. Thus given 1234567 as an input, the program should output *one million two hundred and thirty four thousand five hundred and sixty seven*.

11. Build a currency converter in Qi. Your currency converter will convert dollars, pounds and yen. To help you get started, here is the current exchange scale for all these currencies into Euros.

    1 US Dollar = 0.84104 Euro
    1 British Pound = 1.43266 Euro
    1 Japanese Yen = 0.007725 Euro

    Your converter will receive any positive quantity of money in any of the above currencies and convert it into the target currency chosen by the user. Your program will allow the conversion rates to be reset, but only after password authorisation has beeen given to the system.

12. *Roulette was first played in France back in the 17th century. It is now one of the most popular European gambling games. It requires a roulette wheel which contains 37 numbers, from 0 to 36. Players bet on the outcome of a spin of the wheel.

A bet on one number only, called a *straight-up bet*, pays 35 to 1. (You collect 36. With no house advantage you should collect 37).
A two-number bet, called *split* bet, pays 17 to 1.
A three-number bet, called *street* bet, pays 11 to 1
A four-number bet, called *corner* bet, pays 8 to 1.
A six-number bet, pays 5 to 1.
A bet on a *dozen* – the numbers 1-12, 13-24, or 25-36 pays 2 to 1.
A bet on the *even numbers*, pays 1 to 1.
A bet on the *odd numbers*, pays 1 to 1.
A bet on the numbers 1-18, pays 1 to 1.
A bet on the numbers 19-36, pays 1 to 1.
A bet on red or black - pays 1 to 1.
The black numbers are 31, 22, 29, 28, 35, 26, 15, 4, 2, 17, 6, 13, 11, 6, 10, 24, 33 and 20. All the rest are red with the exception of 0. If 0 comes up, the house collects all bets. Implement a program which allows a player to play roulette against the computer.

Further Reading

O'Donnell (1985) (pp 20-29, 90-97) gives a detailed study of the design and implementation of a programming language based on equations. The Knuth-Bendix procedure is a classic method for transforming arbitrary sets of equations into rewrite rules and is described in **Knuth** and **Bendix** (1970). The original account is not very readable and **Bundy** (1983) and **Duffy** (1991) provide more readable introductions. **Plaisted** (1993) gives a nice overview on the field of rewriting. There are regular conferences, devoted specifically to rewrite systems, published in the LNCS series, amongst which a paper by **Baeten and Bergstra** (1986) details the properties of priority rewrite systems. There has been a lot of discussion in functional programming circles about the advisability of using priority rewriting in the definition of functions. Priority rewriting leads to shorter and simpler programs, but it also separates the definition from its equational representation because these rules can overlap and then prioritising must be used to determine which rule is actually used. In mathematics, changing the order of the equations does not affect what is stated. Order insensitivity is also useful in implementing parallel functional programming languages, since each rule can be tested independently. **Jorrand** (1987) is a study of FP2; an experimental parallel functional programming language based on term rewriting. **Wadler** in **Peyton-Jones** (1987), chapter 5, gives a useful discussion of the conditions under which a set of rewrite rules is order-insensitive.

Web Sites

The Lawrence Livermore National Laboratory (http://www.llnl.gov/sisal/) ran the SISAL project, whose goal is "to develop high-performance functional compilers and runtime systems to simplify the process of writing scientific programs on parallel supercomputers and to help programmers develop functional scientific applications." The result was the SISAL functional programming language available through Sourceforge http://sisal.sourceforge.net that is available for UNIX and Windows. pLisp designed by **Thomas Maher** (http://www.techno.net/pcl/tm/plisp/) is an experimental implementation of parallel functional programming for a Lisp-like language. IEEE's Parascope site (http://computer.org/parascope/) is the best place for links to the world of parallel computing.

Stand-alone implementations of the Knuth-Bendix completion procedure are very difficult to find; generally the procedure is embedded as proper part of larger systems designed to handle equational problems. The theorem prover **OTTER** (http://www-unix.mcs.anl.gov/AR/otter/) contains a Knuth-Bendix completion procedure as does MIT's **Larch** system (http://www.sds.lcs.mit.edu/Larch/index.html).

3 Recursion

3.1 Recursion and Infinity

Programmers using procedural languages are familiar with the idea of creating procedures that call themselves. If we wish to count the number of words in a file, then we could set up a procedure which reads the words one at a time, incrementing a counter, until the file is empty. At this point the value in counter is returned as a result. A procedure of this kind has to be set up to call itself as many times as is necessary for all the words in the file to be counted up.

In functional programming there is an elegant mechanism called **recursion** that enables functions to call themselves. Recursion can be illustrated through equational approach of the previous chapter; our example is the problem of defining the factorial function. The factorial of an integer is the result of multiplying that integer by the integers less than it down to 1. By convention factorial(0) = 1. We start writing a series of equations to define the factorial function (figure 3.1).

$$\begin{gathered}
\text{factorial}(0) = 1 \\
\text{factorial}(1) = 1 \times 1 = 1 \\
\text{factorial}(2) = 2 \times 1 = 2 \\
\text{factorial}(3) = 3 \times 2 \times 1 = 6 \\
\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 24 \\
\text{factorial}(5) = 5 \times 4 \times 3 \times 2 \times 1 = 120 \\
\text{factorial}(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720
\end{gathered}$$

Figure 3.1 Part of an infinite series of equations for the factorial function

The problem is that there are an infinite number of such equations. Can we somehow reduce this infinite set of equations to something finite and of a manageable size? Yes, we can, through the device of recursion. If we examine these equations, they all fit a common pattern. Above 0, the factorial of any integer is the result of multiplying that integer by the factorial of the integer one less than it.

$$
\begin{gathered}
\text{factorial}(0) = 1 \\
\text{factorial}(1) = 1 \times \text{factorial}(0) = 1 \times 1 = 1 \\
\text{factorial}(2) = 2 \times \text{factorial}(1) = 2 \times 1 = 2 \\
\text{factorial}(3) = 3 \times \text{factorial}(2) = 3 \times 2 = 6 \\
\text{factorial}(4) = 4 \times \text{factorial}(3) = 4 \times 6 = 24 \\
\text{factorial}(5) = 5 \times \text{factorial}(4) = 5 \times 24 = 120 \\
\text{factorial}(6) = 6 \times \text{factorial}(5) = 6 \times 120 = 720
\end{gathered}
$$

Figure 3.2 Isolating the pattern within the equations in 3.1

This allows us to use two equations in place of the infinite list we started with (figure 3.3).

$$
\begin{gathered}
\text{factorial}(0) = 1 \\
\text{where } X > 0, \text{factorial}(X) = X \times \text{factorial}(X - 1)
\end{gathered}
$$

Figure 3.3 The equations that define the factorial function

The second equation includes the proviso that X > 0. In representing these equations in Qi, we can take advantage of the fact that Qi tries rewrite rules in the order in which they appear in a function definition. If the first and second equations become the first and second rewrite rules in the definition of factorial, the ordering guarantees that Qi tries the second rule (supposing the input to be a natural number), only when the input is greater than 0. In that case the two equations can be combined into one Qi definition without having to explicitly say that X > 0 (figure 3.4).

```
(define factorial
   0 -> 1
   X -> (* X (factorial (- X 1))))
```

Figure 3.4 The factorial function in Qi

Entering this function to Qi shows it to work.

```
(59-) (factorial 6)
720
```

The calculation of the factorial of any integer > 0 depends on calculating the factorial of the integer one less than it. Thus, the Qi factorial function calls itself *n*+1 times for any natural number *n* and having reached zero, executes all the deferred multiplications.

3.2 Tail and Linear Recursion

The equational definition of factorial in figure 3.4 shows the typical anatomy of a recursive function. The first equation factorial(0) = 1 gives the **base case** of the recursive definition; this provides a point where the function ceases to call itself. The base case corresponds to a condition within a procedural loop that causes the loop to be exited. The second equation, (where X > 0, factorial(X) = X $\times$ factorial(X - 1)), corresponds to the **recursive case** where the function calls itself. The recursive case corresponds to the condition within a loop that causes the loop to be repeated. In the example there is one base case and one recursive case, though in other definitions there can be several of each.

Once the base case zero is reached, the computation does not cease since the recursive outputs of **factorial** have to be multiplied to give the answer. We can rewrite our definition of **factorial** so that the answer is returned on reaching the base case (figure 3.5).

```
(define factorial
  X -> (factorial1 X 1))

(define factorial1
  0 Accum -> Accum
  X Accum -> (factorial1 (- X 1) (* X Accum)))
```

Figure 3.5 A tail recursive definition of factorial

Our new definition of **factorial** passes its input **X** to an auxiliary or **help function** called **factorial1** that helps to define **factorial**. The definition of **factorial1** contains an extra input (an **accumulator**) **Accum,** which totals the value calculated for **factorial**. When the base case **0** is reached, returning the accumulator gives the correct answer.

The two different versions of **factorial** are almost identical, and yet they display different patterns of computation. In the first version, the necessary multiplications are postponed till the end of the recursion. Carrying out this process requires that the computer remember the multiplications to be performed later. The length of the chain of operations, and hence the amount of information needed to keep account of the computation, grows in linear proportion to the value of **X**. Such a recursion is called **linear**

recursion. In the second case, the necessary multiplications are performed during the recursion, and the memory used to keep track of the computation is virtually constant. This recursion, **tail recursion**, is the more efficient form of recursion.

Addition is another example of a recursive arithmetic function. The problem is to recursively define an addition function called “plus” over natural numbers, using only operations which add 1 to or subtract 1 from a number. To begin we write a series of equations giving specific inputs to plus and the corresponding results.

plus(X,0) = X
plus(X,1) = 1 + (plus(X,0))
plus(X,2) = 1 + (plus(X,1))
plus(X,3) = 1 + (plus(X,2))

Figure 3.6 Part of an infinite series of equations for addition

Apart from the first equation (the model for the base case), all the other equations have the form

plus(X,Y) = 1 + (plus(X, (Y - 1))).

Now using the equations plus(X,0) = X and plus(X,Y) = 1 + (plus(X, (Y - 1))), we define a recursive function **plus** in Qi (figure 3.7).

```
(define plus
  X 0 -> X
  X Y -> (+ 1 (plus X (- Y 1))))
```

Figure 3.7 The plus function in Qi

This definition is linear recursive; we leave the tail recursive version as an exercise for the reader.

3.3 Tree Recursion

The Fibonacci function over the natural numbers is defined through the equations in figure 3.8.

fib(0) = 0
fib(1) = 1
if X > 0 and X > 1, then fib(X) = fib(X-1) + fib(X-2)

Figure 3.8 The equations defining the Fibonacci function

The transcription into Qi is easy (figure 3.9).

```
 (define fib
  0 -> 0
  1 -> 1
  X -> (+ (fib (- X 1)) (fib (- X 2))))
```

Figure 3.9 The Fibonacci function in Qi

One point of note is that **fib** is invoked twice in the recursion. Recursive functions that call themselves more than once in one recursive call are called **tree recursive** functions, since the computation branches at such a call into two or more calls to the same function.

3.4 Guards

A prime number is a whole number > 1, divisible only by itself and 1 to give another whole number. Our task is to construct a prime number detector which returns true if X is a prime number and false otherwise.

One way to construct such a detector is to start with 2 and try to divide 2 into X. If the result is a whole number > 1, then X is not prime. If the result is not a whole number then we repeat the test, incrementing 2 to 3. We continue, and if we have incremented the divisor to a number greater than the square root of X; then we know we have found a prime. This algorithm requires a function that (a) holds the number X, (b) holds the square root of X and (c) holds the divisor that is initially set to 2. Call this function *prime1?*. We can define the function *prime?* in terms of *prime1?* (and *prime1?* itself) in four equations (figure 3.10).

prime?(X) = *prime1?*(X, *sqrt*(X), 2)
prime1?(X, Max, Div) = false if X/Div is an integer
                                        and 1 < Div ≤ Max
prime1?(X, Max, Div) = true if Div > Max
prime1?(X, Max, Div) = Y if X/Div is not an integer
          and Max ≥ Div and *prime1?*(X, Max, (1 + Div)) = Y

Figure 3.10 Defining a prime detector in equations

The definition of *prime* is easy - merely a transcription of the equation. The definition of *prime1?* is trickier. The equations have conditions attached to them that must be represented. Qi represents these conditions by **guards**. A guard is a functional expression that evaluates to **true** or **false**, which is placed after the rule and preceded by the keyword **where**, and which must evaluate to **true** for the rule to apply (figure 3.11).

```
(define prime?
 X -> (prime1? X (sqrt X) 2))

(define prime1?
   X Max Div -> true      where (> Div Max)
   X Max Div -> false     where (and (integer? (/ X Div))
                                      (and (< 1 Div) (<= Div Max)))
   X Max Div -> (prime1? X Max (+ 1 Div))
              where (and (not (integer? (/ X Div))) (>= Max Div)))
```

Figure 3.11 An unoptimised encoding of the prime detector

Using our knowledge of the evaluation strategy of this program, some of the code in figure 3.11 can be eliminated. The **(< 1 Div)** test is redundant, because **prime1?** is started with a divisor of **2**. The **(<= Div Max)** is redundant too, because if **(> Div Max),** the program would have terminated. Similarly the **(not (integer? (/ X Div))) (>= Max Div)** tests can be dropped (they must be satisfied if the first two rules fail). After eliminating redundant code, the resulting program appears in figure 3.12.[18]

```
(define prime?
 X -> (prime1? X (sqrt X) 2))
```

Qi Programs/Chap3/prime.qi.

[18] Notice that guards do not actually give Qi any extra computational power; they are merely a convenient device for clarifying the control within a program. In fact all Qi function definitions can be coded with only one rewrite rule in each function! To do this, we transfer all the control information that appears to the left of the **->**, to the right in the form of a nested case statement. What is left on the left of the **->** is simply a series of variables called **formal parameters**. Here is a recoding of prime1? along these lines.

```
(define prime1?
   X Max Div -> (if (integer? (/ X Div))
                  false
       (if (> Div Max)  true (prime1? X Max (+ 1  Div)))))
```

Some functional languages such as Lisp, Scheme and SASL require programs to be written in this style. The translation of Qi programs into this form is both purely mechanical and also an important part of the compilation of Qi. The introduction of significant structure to the left of **->** is what **pattern-directed** functional programming is about.

```
(define prime1?
   X Max Div -> true       where (> Div Max)
   X Max Div -> false      where (integer? (/ X Div))
   X Max Div -> (prime1? X Max (+ 1 Div)))
```

Figure 3.12 An optimised encoding of the prime detector

Entering this prime number program to Qi gives us a prime number recognisor.

3.5 Mutual Recursion

1 is odd and not even. For any natural number X, which is greater than 0 or 1, X is even if its predecessor is odd and odd if its predecessor is even. This definition is used in figure 3.13 to create a program that recognises odd and even numbers.🗀 Functions, which are defined in terms of each other like this, are **mutually recursive**.

```
(define even?
  1 -> false
  X -> (odd? (- X 1)))

(define odd?
  1 -> true
  X -> (even? (- X 1)))
```

Figure 3.13 A mutual recursion between even? and odd?

3.6 Counting Change

How many ways can a one pound coin be changed? We can generalise the question and ask "*Given a sum of n pence, in how many ways can it be converted into change?*" The problem seems difficult, but thinking recursively helps to solve it.

To begin, we identify the units of currency existing in Britain of 2003; they are the 2 pound coin, the pound coin and the 50p, 20p, 10p, 5p, 2p and 1p coins. The problem has a simple recursive solution. Let *d* be the value of the highest denomination, and *n* be the value of the sum of money, which

🗀 Qi Programs/Chap3/mutual.qi

we are trying to break down. Then the number of ways *c* of breaking *n* into change is *c* where

c = the number of ways of breaking *n-d* pence into change +
  the number of ways of breaking *n* pence into change using all
  remaining denominations *except* the highest denomination *d*.

What stops the recursion? We say that

1. There is one way of changing 0 pence (give nothing back!).
2. There are no (zero) ways of changing less than 0 pence.
3. If there is no available denomination to use, there are 0 ways of changing any sum of money.

Putting this reasoning together in Qi gives the function **count-change** (figure 3.14).🗀

```
(define count-change
   Amount -> (count-change* Amount 200))

(define count-change*
   0 _ -> 1
   _ 0 -> 0
   Amount _ -> 0           where (> 0 Amount)
   Amount Fst_Denom
   -> (+  (count-change* (- Amount Fst_Denom) Fst_Denom)
        (count-change* Amount (next-denom Fst_Denom))))

(define next-denom
  200 -> 100
  100 -> 50
  50 -> 20
  20 -> 10
  10 -> 5
  5 -> 2
  2 -> 1
  1 -> 0)
```

Figure 3.14 Counting change in Qi

🗀 Qi Programs/Chap3/change.qi.

Entering **(count-change 100)** to Qi provides the answer to the original question.

```
(23-) (count-change 100)
4563
```

3.7 Non-Terminating Functions

Recursion gives the ability to define **non-terminating** functions. A fatuous example is

```
(define silly
  X -> (silly X))
```

This calls itself repeatedly without terminating. Here **(silly X)** is undefined for all values of **X**, since whatever input is submitted to **silly**, no normal form is computed.[19]

There are other examples of functions, which are non-terminating only for certain inputs; for example the expression **(factorial -1)** has no normal form. The last example shows that functions that fail to terminate for all inputs are not always silly. In the factorial case, we could avoid non-termination by testing to see if the input is a negative number. However, there remain functions whose computation is not guaranteed to terminate and which are resistant to all safeguards. A good example comes from number theory. A very famous conjecture in number theory, called "Goldbach's conjecture",[20] suggests that every even number > 2 is the sum

[19] Since waiting for **(silly 0)** to terminate is to wait forever; it is desirable to break such a computation. Qi runs on top of Common Lisp, and different Common Lisps support different facilities for breaking execution. Breaking an Qi computation may force the program out of the Qi read-evaluate-print loop and into the underlying Lisp read-evaluate-print loop. In CLisp, breaking can sometimes be achieved by control-C followed by **x** to escape back to Qi. In less robust implementations of Lisp, control-C in the middle of execution can cause the image to core dump or hang. In Franz Lisp, control-C does nothing. Consult your resident Lisp manual to see how to break executions.

[20] The Prussian mathematician Christian Goldbach suggested Goldbach's conjecture in 1742. It has been tested up to 4×10^{14}. In 2001 a $1 million prize was announced in the Chicago Sun-Times for the first proof of Goldbach's conjecture. Two major publishers funded the money and the competition remained open until March 15th 2002. It was not won.

of two primes. The following program tests the conjecture. The program🗀 introduces the comment facility in Qi, which brackets comments by \.[21]

Currently the truth of Goldbach's conjecture is open to proof or disproof. If conjecture is true, then the program in 3.15 will not terminate. If it is false, then it will halt and print out an integer that will refute the conjecture. Currently, mathematicians do not know what would happen. But from a programmer's viewpoint, the program is very inefficient because it uses **next-prime** to recalculate the same primes repeatedly. In the next chapter, we look at how lists can be used to store results for programs like this.

```
(define slow-goldbachs-conjecture
  \ Begin the conjecture with the first even number > 2.\
  start -> (slow-goldbachs-conjecture1 4))

(define slow-goldbachs-conjecture1
  \ Is N the sum of two primes?  Try the next even number. \
   N -> (slow-goldbachs-conjecture1 (+ N 2))
                    where (sum-of-two-primes? N)
   \ Else return N  and halt.  Goldbach's conjecture is wrong.\
   N -> N)

(define sum-of-two-primes?
  \Test to see if N is the sum of two primes;
   we begin by fixing the first prime at 2. \
   N -> (sum-of-two-primes*? 2 N))

(define sum-of-two-primes*?
    \ Is the prime we are testing > N, then return false;
     N is not the sum of two primes. \
    Prime N -> false       where (> Prime N)
    \ Else find if the prime we are using can be summed with
     another prime to give N.  Start the count at 2. \
    Prime N -> true        where (sum-of? Prime 2 N)
    \ If not take the next prime up and repeat. \
     Prime N -> (sum-of-two-primes*?
                                   (next-prime (+ 1 Prime)) N))
```

🗀 Qi Programs/Chap3/goldbach version 1.qi. The program contains a new system function **do**. In future, rather than repeat the purpose of such simple functions, I leave it to the reader to consult appendix A which details all the system functions used in this book.

[21] Material between ****s is always ignored by the compiler except where **** is used to construct characters (e.g. **#\a**). Characters are formed by prefacing the **** by a hash **#**.

```
(define sum-of?
   \ If N is the sum of two primes then report it and return true
    (do evaluates n inputs, returning the value of the last one)\
  Prime1 Prime2 N
   -> (do (output "~A is the sum of ~A and ~A~%"
                   N Prime1 Prime2) true)
                   where (= N (+ Prime1 Prime2))
    \ If the sum of the two primes is > N return false;
      no point in using bigger primes! \
  Prime1 Prime2 N -> false  where (> (+ Prime1 Prime2) N)
    \ Otherwise try the next prime up  \
     Prime1 Prime2 N
   -> (sum-of? Prime1 (next-prime (+ 1 Prime2)) N))

 (define next-prime
   X -> X          where (prime? X)
   X -> (next-prime (+ 1 X)))
```

Figure 3.15 Goldbach's Conjecture in Qi

Exercise_3

1. Define the following functions.
    a. **expt**, that raises a number M to a natural number power N.
    b. **round_n**, that rounds a floating point number M to N places, rounding downwards.
    c. **perfect?**, that returns **true** if N is perfect. A perfect number is a natural number > 1 that is the sum of its divisors. 28 is perfect since 28 = 1 + 14 + 2 + 7. 6 is perfect since 6 = 1 + 2 + 3.
    d. **triangular?**, that returns **true** if N is a triangular number. A triangular number is a natural number > 0 that is the sum of the first *n* natural numbers for some *n*. 1, 3, 6, 10, 15, 21 are all triangular since 1 = 0 + 1, 3 = 0 + 1 + 2, 6 = 0 + 1 + 2 + 3 and so on.
    e. **square?**, that returns **true** if N is a perfect square. A perfect square is the result of multiplying a natural number by itself.
    f. **gcd**, which computes the greatest common divisor of two natural numbers a and b. This function returns the largest divisor common to a and b. The **gcd** of 12 and 16 is 4.
    g. **lcm**, which computes the least common multiple of two natural numbers a and b. This function returns the smallest number into which a and b both divide to give whole numbers. The **lcm** of 12 and 15 is 60.

2. *Define the 3-place function **nth_root** that receives the natural number N, the positive number M and the natural number P, where P > 0. **nth_root** finds the

Nth root of M to an accuracy proportional to P as follows. Fix the conjectured value C for $^N\sqrt{}$ M at M/2. Raise C to the power N and determine if C^N rounded to P places equals M. If so, return C. If C^N rounded to P places > M, reduce the value of C and repeat. If C^N rounded to P places < M, increase the value of C and repeat. The trick is seeing how to increase or decrease the value of C to converge on a solution.

3. *The **expt** and **nth_root** programs and the formula $a^m * a^n = a^{m+n}$ provide a method for calculating the value of a^b where b is a positive floating number. An approach to evaluating a^b is to add b to itself n times to derive a whole number . We calculate the value of a^{n*b} to derive a decimal number D and use **nth_root** to determine $^n\sqrt{}$ D to a desired accuracy P. For example, $10^{1.5} * 10^{1.5} = 10^3 = 1000$. The decimal value of √1000 can be calculated by **nth_root** program and that is the decimal value of $10^{1.5}$. Implement the program to accuracy P = 2 and print the antilog tables for the base 10, starting at 10^1 and going up to 10^3 by increments of .1, giving the log and its decimal equivalent.

4. Melvin Micro has defined his own version of **if** as follows:-

```
(define melvins_if
  true X _ -> X
  false _ Y -> Y
  Test _ _ -> (error "test ~A must be true or false.~%" Test))
```

He defines factorial as

```
(define factorial
   X -> (melvins_if (= X 0) 1 (* X (factorial (- X 1)))))
```

What happens when Melvin tries to use this function to compute factorials? Give reasons.

5. Melvin Micro has deposited $10,000 in a long term account in the year 2000. At 5% interest per year, he hopes to collect on this account for his retirement. Assuming he takes no money from his account, what will it be worth in 2040?

6. In 2005, the USA produced 10,900 billion dollars of goods and services with a growth rate of around 2.5% per annum. The same year China produced 6,400 billion dollars of goods and services with a growth rate of 9% per annum. Assuming these rates are maintained, at what year does the GNP of China exceed that of America?

7. In probability theory, the probability P(A v B) of either or both of two independent events A or B occurring is given by the equation.

$$P(A \vee B) = 1 - ((1 - P(A)) \times (1 - P(B)))$$

P(A) is the probability of A occurring and P(B) is the probability of B occurring. 1 indicates certainty, 0 impossibility. Assume that the probability of a major global catastrophe in any one year is 1/200. What is the probability of a major global catastrophe in the next fifty years?

8. The factorisation of very large numbers is a key element in computer security. The prime factors of a number *n* are those primes which when multiplied together give *n.* Devise a program to compute the prime factors of the number 123456789.

9. Two regiments of riflemen are facing each other. One side has 1000 men, the other 800. The smaller side fires first and then the larger and so on alternately. A riflemen hits and kills his enemy once in every 10 shots. Once a man is hit and killed he cannot fire back. The fight is to the finish. How many men are left in the stronger side at the end of the fight? What happens if the stronger side fires first?

10. *Implement a calendar program which determines what day of the week a particular date falls on. Use your program to calculate the day of the week on 13th July 2113.

11. A high-speed reconnaissance plane weighs 105,000 lb of which 70,000 lb is fuel. At a cruise speed of 1,800 mph, the plane uses 3lb of fuel per 10,000lb of weight per mile. The airforce need to know how long the plane will cruise and hence what is its effective radius of action. Write a program to compute a solution. Your program should allow the user to set the parameters of this problem for different aircraft.

Further Reading

The Fibonacci function in this chapter executes in exponential space relative to the input *n*. **Henson** (1987, chapter 4) shows how to derive a linear Fibonacci function definition from the tree recursive version using the rules for manipulating function definitions in **Burstall** and **Darlington** (1977). **Bird** (1984) and **Wand** (1980) extend this technique. **Abelson** and **Sussman** (1996) explain how to detect prime numbers using the square-root method used here and also present a much faster method based on the Fermat test. This book also has a good discussion of various types of recursion and the compilation strategies for dealing with them. The problem of showing that the evaluation of an arbitrary expression halts or showing it does not is, of course, unsolvable (being a corollary of the Halting Problem).

The problem of counting change is an old one. **Kac** and **Ulam** (1971) p.34-39 show how to calculate the number of ways of splitting a dollar without the use of a computer; their method uses simultaneous equations and polynomials. The computation used in this book is not particularly efficient since identical computations are repeated. **Memoisation** is a technique for recording the results of a computation so it need not be performed more than once. For more on memoisation see **Field** and **Harrison** (1988).

Web Sites

Chris Caldwell (http://www.utm.edu/research/primes/) maintains a site devoted to primes. http://www.fortunecity.com/meltingpot/manchaca/799/prime.html is a less detailed, but a more accessible account of primes.

4 Lists

4.1 Representing Lists in Qi

In Qi, a list begins with a [and ends with a]. So the list composed of 1, 2 and 3 would be written as [1 2 3]. Spaces are used to separate items; the list [123] is not the same as the list [1 2 3], since [123] is the list containing the single number 123. Qi evaluates lists according to the **list evaluation rule**.

The List Evaluation Rule

The normal form of a list L = $[i_1,...,i_n]$ is the list $[i'_1,...,i'_n]$ arrived at by evaluating the contents of L from left to right where for each i_j $(1 \leq j \leq n)$, i'_j = the normal form of i_j.

Suppose we wish to find the normal form of **[(* 8 9) (+ 46 89) (- 67 43)]** using the list evaluation rule. We evaluate **(* 8 9)** first.

(* 8 9) $\Rightarrow$ 72

Second we evaluate **(+ 46 89)**

(+ 46 89) $\Rightarrow$ 135

Last we evaluate **(- 67 43).**

(- 67 43) $\Rightarrow$ 24

So the result is **[72 135 24].**

Lists can be placed within lists without restriction (figure 4.1).[22] The last example in that figure features one rather special sort of list - the empty list [ ]; a list which contains nothing and which evaluates to itself.[23] Qi requires all brackets to balance before undertaking an evaluation. If there unequal number of (s and)s, or [s and]s, then Qi will wait for the user to balance them. ^ can be used to abort the line input in periods of confusion.

```
[[1]] ⇒ [[1]]
[tom dick harry] ⇒ [tom dick harry]
[[1 dick [3]]] ⇒ [[1 dick [3]]]
[(+ 1 2) [(+ 2 3)]] ⇒ [3 [5]]
[(+ 34 34) [(* 7 8) "foobar" [(- 9 8)]]] ⇒ [72 [56 "foobar" [1]]]
[] ⇒ []
```

Figure 4.1 Some examples of lists and their normal forms

4.2 Building Lists with Cons

There is one basic function for building up lists called **cons**. The function **cons** receives two inputs, an object O and a list L[24] and builds a list L′ through adding O to the front of L. This is called **consing** O to L (figure 4.2).

```
(0-) (cons 1 [2 3])
[1 2 3]

(1-) (cons tom [dick harry])
[tom dick harry]

(2-) (cons 1 [ ])
[1]
```

[22] This is true of untyped functional languages, but not of typed ones. We shall see in chapter 10 that, with type checking, lists of objects of different types are not allowed.

[23] The empty list can also be written as **( )** or **NIL** in Qi. Both these representations are inherited from Lisp; the use of **NIL** for the empty list is the one of two exceptions to the rule that any symbols beginning with uppercase symbols are variables. The other is **T**.

[24] In Lisp, it is permitted to cons an object to something which is not a list (e.g. **(cons 1 2)**). The result is known as a **dotted pair** and is generally printed in Lisp as **(1 . 2)**. Lists are therefore a proper subset of the set of dotted pairs where the second element of the pair is a list. Qi preserves Lisp's liberty of allowing dotted pairs, printing **(cons 1 2)** as **[1 . 2]**. With type checking switched on, dotted pairs that are not lists will be registered as having a type error.

```
(3-) (cons tom (cons dick (cons harry [ ])))
[tom dick harry]
```

Figure 4.2 Using the cons function

Using **cons**, a supply of self-evaluating expressions, and the empty list **[]**, any finite list can be built up. For instance, suppose we wish to construct the list **[[a b] c [d e]]**. The list **[a b]** is formed by consing **a** to the result of consing **b** to **[]**.

[a b] = (cons a (cons b [ ]))

[d e] is the result of consing **d** to the result of consing **e** to **[ ]**

[d e] = (cons d (cons e [ ]))

The list **[[d e]]** is formed by consing **[d e]** to **[ ].**

[[d e]] = (cons [d e] [ ])

The list **[[a b] c [d e]]** is the result of consing **[a b]** to consing **c** to **[[d e]].**

[[a b] c [d e]] = (cons [a b] (cons c [[d e]]))

By making the necessary substitutions in the right-hand-side of this equation, the expression **[[a b] c [d e]]** is expressed in terms of consing operations.

[[a b] c [d e]] = (cons (cons a (cons b [ ])) (cons c (cons (cons d (cons e [ ])) [ ])))

When a list is described like this, using only conses, **[ ]** and self-evaluating expressions we say that it is in **cons form**. For practical purposes, writing lists in cons form is tedious and unnecessary. However, when we come to reason about programs, it is important to know that any list can be recast in cons form. The translation of a list to its cons form version is purely mechanical. Simply keep applying the following rules throughout any list representation until they can no longer be applied.

Cons Form Translation Rules

(i) [ ] is translated as [ ].
(ii) [e_1,...,e_n] where ($n \geq 1$) is translated as (cons e_1 [,...,e_n]).

4.3 | is Shorthand for Cons

A shorter way of expressing consing (as an alternative to writing "**cons**") is to use **|**; every expression to the left of **|** is consed in turn to the list on the right of **|** (figure 4.3).

```
(0-) [1 | [2 3]]
[1 2 3]

(1-) [tom dick harry | [ ]]
[tom dick harry]
```

Figure 4.3 Using **|** *in* Qi *to cons expressions*

We explain how to evaluate expressions containing **|** by showing how such expressions can be translated into expressions not containing **|**, but **cons** instead. The translation involves applying the following rules to every part of any given expression until the **|**s are gone.

| Elimination Rules

(i) [e_1 | e_2] is translated to (cons e_1 e_2).
(ii) [e_1 ,..., e_n | e_{n+1}] ($n \geq 2$) is translated to (cons e_1 [,..., e_n | e_{n+1}]).

We apply these rules to **[tom dick harry | [ ]].**

```
[tom dick harry | [ ]]
= (cons tom [dick harry | [ ]])                by rule (ii)
= (cons tom (cons dick [harry | [ ]]))         by rule (ii)
= (cons tom (cons dick (cons harry [ ]))) by rule (i)
⇒ [tom dick harry].
```

4.4 Head and Tail Access Parts of Lists

cons builds lists out of component parts. **head** and **tail** allow these component parts to be extracted from the assembled lists. **head** receives a non-empty list and returns the first element in it; **tail** receives a non-empty list and returns all *but* the first element of that list (figure 4.4).

```
(0-) (head [1 2 3])
1

(1-) (tail [1 2 3])
[2 3]

(2-) (head [[tom] dick harry])
[tom]

(3-) (tail [tom [dick harry]])
[[dick harry]]
```

Figure 4.4 Using head and tail to extract elements from a list

Using **head** and **tail** in composition, we can access any element in a list.

```
(4-) (head (tail [1 2 3]))
2

(5-) (head (tail (tail [1 2 3])))
3
```

4.5 Characters

In Qi, any object can be mapped into its essential characters; for instance the symbol **cat** is composed of three characters, represented as **#\c #\a** and **#\t**. The string **"cat"** is represented by a series of 5 characters - **#\"**, **#\c**, **#\a**, **#\t**, and **#\"**. The function **explode** decomposes any object into its constituent characters.

```
(6-) (explode cat)
[#\c #\a #\t]

(7-) (explode "the cat")
[#\" #\t #\h #\e #\Space #\c #\a #\t #\"]

(8-) (explode [cat])
[#\[ #\c #\a #\t #\]]
```

Figure 4.5 Using explode to extract characters

The decomposition of an object into characters is useful when we want to analyse the component parts of a symbol or string. For example, here is a function that recognises Qi variables. **element?** searches a list to see if the first object supplied to it is found in the list.[25]

```
(define qi_variable?
  X -> (element? (head (explode  X))
        [#A #B #C #D #E #F #G #H #I #J #K #L
       #M #N #O #P #Q #R #S #T #U #V #W #X #Y #Z]))
```

4.6 Recursive List Processing

The function **element?** returns true if its first input X occurs somewhere in the list that is its second input Y and false otherwise. How may this function be defined? Intuitively, nothing is a member of the empty list, and if the element is found to be at the head of Y then we may return true. If neither of these obtain, then we have to search in the rest of the list. We express this reasoning as an equational specification in figure 4.6.

$$\text{member}(X, [\,]) = \text{false}$$
$$\text{member}(X,Y) = \text{true if } X = \text{head}(Y)$$
$$\text{member}(X,Y) = \text{member}(X, \text{tail}(Y)) \text{ if } X \neq \text{head}(Y)$$

Figure 4.6 The equational specification of the member function

Turning this into an Qi function is easy (figure 4.7).

```
(define member
   _ [ ] -> false
   X Y -> true              where (= X (head Y))
   X Y -> (member X (tail Y)))
```

Figure 4.7 The member function in Qi

join (figure 4.8) joins two lists $[x_1,...,x_n]$ and $[y_1,...,y_n]$ together into a single list $[x_1,...,x_n, y_1,...,y_n]$.

join([ ],X) = X
join(X,Y) = cons(head(X),join(tail(X),Y)) if X ≠ []

[25] There is also a higher-order function **read-to-stringlist** which converts a list of characters into a list of strings. See appendix A for details.

```
 (define join
   [ ] X -> X
   X Y -> (cons (head X) (join (tail X) Y)))
```

Figure 4.8 The equational specification and the Qi *code for the join function*

The function **rev** reverses the order of the elements in a list (figure 4.9).

$$\mathrm{rev}([\,]) = [\,]$$
$$\mathrm{rev}(X) = \mathrm{join}(\mathrm{rev}(\mathrm{tail}(X)),[\mathrm{head}(X)]) \text{ if } X \neq [\,]$$

```
(define rev
  [ ] -> [ ]
  X -> (join (rev (tail X)) [(head X)]))
```

Figure 4.9 The equational specification and the Qi *code for the reverse function*

The | is used as a shorthand for cons and all lists can be represented in cons form. Qi will allow the list to be represented using | on the left-hand side of arrow. For example, the function **foo** defined as follows.

```
(define foo
  [a b] -> [b])
```

can also be written as:-

```
(define foo
  [a | [b]] -> [b | [ ]])
```

or even as

```
(define foo
  (cons a (cons b [ ])) -> (cons b [ ]))
```

since **[a | [b]]**, **[a b]** and **(cons a (cons b [ ]))** are the same (as is **[b]**, **[b | [ ]]** and **(cons b [ ])**). Figure 4.10 shows a recoding using **|** of the previous list processing functions.

```
(define member
  _ [ ] -> false
  X [X | _] -> true
  X [_ | Y] -> (member X Y))
```

```
(define join
  [ ] X -> X
  [X | Y] Z -> [X | (join Y Z)])

(define rev
  [ ] -> [ ]
  [X | Y] -> (join (rev Y) [X]))
```

Figure 4.10 Recoding some list processing functions using |

We will now look at two more list processing functions; the first returns the Cartesian product of two lists and the second generates the list equivalent of the powerset of its input. The Cartesian product of X and Y is formed by pairing every element of X with every element of Y. So **(cartesian-product [1 2 3] [a b])** evaluates to **[[1 a] [2 a] [3 a] [1 b] [2 b] [3 b]].** To form the Cartesian product we take the first element of X and form all possible pairs with the elements of Y; then the next element of X in the same way, and so on until X is emptied. The sets of pairs are then joined into one set (figure 4.11);🗀 (**append** is the system function version of **join**).

```
(define cartesian-product
  [ ] _ -> [ ]
  [X | Y] Z -> (append (all-pairs-using-X X Z)
                              (cartesian-product Y Z)))

(define all-pairs-using-X
  _ [ ] -> [ ]
  X [Y | Z] -> [[X Y] | (all-pairs-using-X X Z)])
```

Figure 4.11 Cartesian products in Qi

The second example defines a function, **powerset,** that finds the powerset of a finite set (i.e. the set of all its subsets). Thus powerset({a,b,c}) = {{a,b,c},{a,b},{a,c},{b,c}, {a},{b},{c},{}}. We represent a set as a list that contains no duplicated elements. Here is an Qi definition in figure 4.12.🗀

🗀 Qi Programs/Chap4/cartprod.qi.

🗀 Qi Programs/Chap4/powerset version 1.qi.

```
(define powerset
  [ ] -> [[ ]]
  [X | Y] -> (append (cons-X-to-each-set X (powerset Y))
                         (powerset Y)))

(define cons-X-to-each-set
   _ [ ] -> [ ]
   X [Y | Z] -> [[X | Y] | (cons-X-to-each-set X Z)])
```

Figure 4.12 An inefficient coding of powerset

4.7 Local Assignments

The previous definition of powerset is inefficient because **(powerset Y)** is computed twice within the same expression. A better strategy would be to compute the normal form of **(powerset Y)** once and store it for use the second time. We do this with **local assignments** using **let**. The **let** construction
(**let** *V E A*) receives three inputs *V*, *E* and *A* where;

1. *V* is a variable, which is assigned a value which is …
2. the normal form *N* of *E*, and (**let** *V E A*) returns ……
3. the normal form of the expression that results from substituting *N* for all free instances of *V* within *A*.

Here is **powerset** redefined in figure 4.13 using **let.**[🗀]

```
(define powerset
  [ ] -> [[ ]]
  [X | Y] -> (let Powerset (powerset Y)
          (append (cons-X-to-each-set X Powerset) Powerset)))
```

Figure 4.13 An efficient coding of powerset using local assignments

The assignment of **(powerset Y)** to **Powerset** is local because the assignment works within the scope of the **let** symbol. We cannot usefully employ **Powerset** outside **(let Powerset (powerset Y) (append (cons-X-to-each-set X Powerset) Powerset))** because **Powerset** would have no value attached to it and a **free variable** warning would be raised.

[🗀] Qi Programs/Chap4/powerset version 2.qi.

4.8 Goldbach's Conjecture Revisited

The previous chapter closed with an example of a possibly non-terminating program to test Goldbach's conjecture. The wasteful recalculation of primes meant that the program ran unfeasibly slowly given even a three digit number on which to test the conjecture. Lists provide the solution; we carry around in the program a list of all the primes calculated up to the number being tested. We close this chapter with an improved version using lists.🗀

```
(define goldbachs-conjecture
   \begin with 4 and the list of primes < 4 \
  start -> (goldbachs-conjecture1 4 [3 2]))

(define goldbachs-conjecture1
  \ If N is not the sum of two primes < N, return N  \
  N Primes -> N  where (not (sum-of-two? Primes N))
 \ else add 2 to N, and if N+1 is prime add N+1 to the list of primes < N
and  recurse;  else simply add 2 to N and recurse  \
  N Primes  -> (if (prime? (+ N 1))
                    (goldbachs-conjecture1 (+ N 2) [(+ 1 N) | Primes])
                    (goldbachs-conjecture1 (+ N 2) Primes)))

(define sum-of-two?
  \ No primes left?  Then return false  \
  [ ] _ -> false
  \ If the first prime x summed with any other prime = N? return true  \
  [X | Primes] N  -> true  where (x+prime=n X [X | Primes] N)
  \ No?  Then recurse.  \
  [_ | Primes] N -> (sum-of-two? Primes N))

(define x+prime=n
 \ no primes left, return false  \
  _ [ ] _ -> false
  \ X + the first prime = N?, so return true  \
  X [Prime | _] N -> (do (output "~A is the sum of ~A and ~A~%" N Prime X)
                         true)        where (= (+ X Prime) N)
  \ recurse and try the other primes  \
   X [_ | Primes] N -> (x+prime=n X Primes N))
```

Figure 4.14 A more efficient coding of Goldbach's Conjecture

🗀 Qi Programs/Chap4/goldbach version 2.qi

Exercise 4

1. Give the cons-form representation of each of the following.
   [a b c], [[a] [b] [c]], [a [b [c]]], [[a | [b]] c], [a | [b | [c]]]

2. Using only head and tail, isolate **c** in each of the expressions in question 1.

3. Define each of the following.

   a. A function **total** that returns the total of a list of numbers.
   b. A function **remdup** that removes all duplicated elements in a list.
   c. A function **first_n** that receives a list **L** and a natural number **N** and returns the list of the first **N** elements of **L.**
   d. A function **flatten** that receives a list **L** and flattens it by removing all internal brackets - so **(flatten [a [b] [[c]]])** ⇒ **[a b c].**
   e. *A function **permute** that returns all the permutations of a list; **(permute [a b c]) = [[a b c] [a c b] [b a c] [b c a] [c a b] [c b a]].** The order of the permutation are does not matter.
   f. *A function **exchange** that exchanges the *m*th and *n*th elements of a list **L**; so **(exchange 3 5 [a b c d e f])** ⇒ **[a b e d c f].**

4. Classify the recursive function definitions used to answer the previous question as tail-recursive, tree-recursive, mutually recursive or linear recursive

5. Define a binary number as a non-empty list of 0s and 1s.

   a. *Define **binary-add** that adds two binary numbers.
   b. *Define **binary-subtract** that subtracts two binary numbers.
   c. Write a function **decimal-to-binary** that converts a decimal natural number into binary.
   d. Write a function **binary-to-decimal** that converts a binary number into decimal.

6. Rewrite the counting change program of the previous chapter so that it receives a list of the denominations in use and returns the answer. So our original result of **4563** would be returned **by (count-change 100 [200 100 50 20 10 5 2 1]).**

7. ***Write a program that enumerates the powerset of a set [1,2,3,….20] by printing the elements of this powerset down the screen. Your program will also supply the millionth element of this enumeration in a second or less. The order of the elements in the enumeration does not matter.

8. *The ancient English game of *cribbage* is played in five and six card variations. We shall ignore some of the subtleties of scoring in this question. In cribbage a hand is scored as follows. Sets of cards whose total pip value is 15 score 2.

Any number of cards can be in a set. For the purpose of totalling, face cards count as 10, ace as 1. Three or more cards in sequence (called a *run*) count 1 point for each card in the sequence. An ace is low in cribbage, so ace, two, three is a run, but not queen, king, ace. In five card cribbage, three or more cards of the same suit (called a *flush*) count one point for each card in the flush. In six card cribbage, *four* or more cards of the same suit count one point for each card in the flush. A *pair* (two cards of the same rank) counts 2 points. Here is a hand in six card cribbage.

J♥, Q♥, K♥, K♠, K♣, 5♥

The value of this hand is as follows. 10 points for cards totalling 15 (pair the 5 off with every other card); 6 points for runs (two runs of J, Q, K); 4 points for a flush (the four hearts) and 6 points for pairs (three distinct pairs made from the three kings) giving 26 points in all. Devise a program that scores hands in five and six card cribbage.

9. Cribbage requires the player to discard two cards from his hand. In six card cribbage this gives 30 possible discards and in five card cribbage, 20 possible discards. Devise a function that goes through all the possible discards and works out which is the best discard. The best discard will maximise the value of the remaining cards left in the player's hand (three cards in the case of five card cribbage and four in the case of six card cribbage).

10. *** Nullos* is a very skilful, but simple card game. There are two players, each is dealt 13 cards. Non-dealer leads a card. His opponent replies with a card of the same suit if possible, and if not, with any card she chooses. Whoever plays the highest card of matching suit wins the trick and plays the next lead. Ace is high. The object of the game is not to win tricks however, but to lose the last trick. Whoever loses the 13th trick wins the game. Implement a program to play and win at Nullos.

11. *The Roman numeral system was used for many centuries in the ancient world. The table below shows Roman numerals and their decimal equivalents.

Roman	Decimal
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

"The Roman counting system used tallies for numbers ending in 3 (III = 3, 33 = XXXIII). However, four strokes seemed like too many, so the Romans moved on to the symbol for 5 - V. Placing I in front of the V, or placing any smaller number in front of any larger number, indicates subtraction. So IV means 4. After V comes a series of additions - VI means 6, VII means 7, VIII means 8. IX means to subtract I from X, leaving 9. Numbers in the teens, twenties and thirties follow the same form as the first set, only with X's indicating the number of tens. So XXXI is 31, and XXIV is 24. 40 is XL, and 60, 70, and 80 are LX, LXX and LXXX. CCCLXIX is 369. As you can probably guess by this time, CD means 400. So CDXLVIII is 448." Implement a program that takes a Roman numeral from 1-3000 as a list and outputs the corresponding decimal number. (From http://www.novaroma.org/via_romana/numbers.html).

12. *(For readers with some logic). Implement a tautology tester which, by use of truth-tables, returns **true** if the input is a tautology and **false** if not. Thus **(tautology? [[p & q] => q])** should return **true**.

Further Reading

The cons-cell representation of a list is the basis for the representation of a list within the architecture of the digital computer. **Allen** (1978) contains an account of cons-cell representations. There are other alternatives. **Abelson & Sussman** (1996) has a discussion of streams which can emulate lists of infinite length. Miranda^{TM} uses schemas based on set comprehension as well as lists. The language SETL discussed **in Schwartz J.T., Dewar R.B.K., Dubinsky, E., and Schonberg, E**., (1986) uses the set instead of the list as its basic data structure. http://galt.cs.nyu.edu/~bacon/other-setl.html provides many links to ongoing work with SETL.

5 Higher Order Programming

5.1 Higher Order Functions

Functions often seem insubstantial since they have no physical existence; the square root function, for instance, cannot be given a location; there is no moment when it was created, since it stands outside time and space. For people of a philosophical persuasion, this makes functions very strange objects. However a willingness to countenance functions as substantial objects is implicit in functional programming, where functions are treated as potential inputs or outputs to other functions, just as other objects like numbers or strings. In functional programming, functions enjoy the same rights of processing as other data objects; i.e. functions are **first-class objects**.

Higher-order functions receive functions as inputs or return them amongst the output. In Qi **map** is a higher-order system function. Its definition is given below

```
(define map
   F [ ] -> [ ]
   F [X | Y] -> [(F X) | (map F Y)])
```

map receives a 1-place function **F** and a list **L.** If **L** is empty, then **L** is returned. But if **L** is of the form **[X | Y]** then the function **F** is applied to the first element **X** and the result is consed to the effect of recursing on the tail of the list using **map**. Figure 5.1 shows the use of **map.**

```
(0 -) (define double
         X -> (* X 2))
double

(1-) (map double [1 2 3 4])
[2 4 6 8]
```

Figure 5.1 Using the map function

5.2 Abstractions, Currying and Partial Applications

The function that doubles a number can be represented in Qi as **(/. X (* X 2))**. The /. is the Qi ASCII substitute for the Greek lambda, and **(/. X (* X 2))** is referred to as an **abstraction**. We can most easily read the notation of abstractions by taking /. to mean "the function that receives an input…". Thus **(/. X (* X 2))** is read as "the function that receives an input **X** and returns the result of multiplying **X** by **2**". Figure 5.2 shows the use of an abstraction.

```
(1-) (/. X (* X 2))
#<CLOSURE :LAMBDA [X] [* X 2]>

(2-) ((/. X (* X 2)) 5)
10
```

Figure 5.2 Using abstractions

The evaluation of an abstraction in Qi delivers an object called a **closure**, which is a compiled representation of the original abstraction with zero or more pieces of data attached to it.[26] In Qi, closures are represented as expressions beginning **#<CLOSURE** …>.[27] These expressions are **internal expressions**; they cannot be read and evaluated in their computer-printed form (i.e. typing **#<CLOSURE :LAMBDA [X] [* X 2]>** will generate an error).

As shown, abstractions can be applied to expressions just like normal functions. They differ from conventional Qi functions in being applicable to only one input, for exactly one variable must follow the /.. However it is

[26] More precisely, a closure is a function represented as a pair composed of a lambda abstraction together with an assignment of values to the free variables of that abstraction. See chapter 12 of this book for a discussion of lambda expressions and Field and Harrison (1988) p. 200 for a discussion of closures.

[27] This representation may vary according to the Lisp platform being used.

possible to represent *n*-place functions with abstractions, where *n* is any value, using **currying**. An example with the 2-place function + will illustrate the idea.

The 2-place function + can be represented as the result of the interaction of two 1-place functions. The first function *f* receives a number *x* and produces a function g that waits to receive another number *y*. Whatever number *y* is supplied to *g*, *y* is added to *x* to produce the result *x* + *y*. Thus the function + is denoted by the expression.

"the function that receives an input *x* and returns a function that receives an input *y* that then returns *x* + *y*."

Translating this into the notation of abstractions gives **(/. X (/. Y (+ X Y)))**. Now suppose we tell Qi to apply this expression to the number 5.

```
(1-) ((/. X (/. Y (+ X Y))) 5)
#<CLOSURE :LAMBDA [Y] [+ X Y]>
```

Figure 5.3 Applying an abstraction to produce a function

Qi returns a closure in which the variable **X** is assigned 5 in memory. The closure represents a function that adds 5 to its input. If **((/. X (/. Y (+ X Y))) 5)** is itself applied to 3, then the result is the same as applying this closure to **3**, which produces 8.

```
(2-) (((/. X (/. Y (+ X Y))) 5) 3)
8
```

Figure 5.4 Using an abstraction to add two numbers

When an abstraction is applied to an input to produce a function (as in figure 5.3) then the application is called a **partial application**.[28] Since the 2-place function symbol '+' and the abstraction '**(/. X (/. Y (+ X Y)))**' both denote the same operation of addition, then it is reasonable to expect that we should be able to enter '**(+ 5)**' as a partial application in Qi and return a

[28] Partial applications are constructed dynamically in Qi, by examination of the arity of the function and the number of values to which it is applied. This is computationally expensive, though the Qi compiler only does this when a higher-order function is invoked. As a consequence, Qi functions like **map** may run more slowly than functions like the Lisp **MAPCAR**. Users who want performance and are not worried about having partial applications can always use the Lisp **MAPCAR** or define their own mapcar (to save free variable warnings) in the manner described in appendix B.

closure just as in figure 5.3. This is correct; we can substitute '**+**' **for** '**(/. X (/. Y (+ X Y)))**' and gain the same function as in 5.3.

```
(3-) (+ 5)
#<CLOSURE :LAMBDA [Z1413] [+ 5 Z1413]>

(4+) ((+ 5) 3)
8
```

Figure 5.5 Currying + to add two numbers

Nearly every application can be written to Qi in the form (*f i*), consisting of a function *f* and a single input *i*. When an expression is presented in this way it is said to be in **curried** form and the operation of placing it in this form is **currying.**[29]

5.3 Using Abstractions in Higher Order Programs

Abstractions can be incorporated into function definitions. This function receives a list of numbers and doubles each one.

```
(define double-everything
   X -> (map (/. Y (* Y 2)) X))
```

Figure 5.6 Using an abstraction to double each number in a list

A function *f* may cite an abstraction *g*, where the identity of the function *g* depends on the inputs supplied to *f*. This function, **xn-everything**, when supplied with a number **N** and a list **L**, multiplies every element of **L** by **N** (figure 5.7).

```
(define xn-everything
  N L -> (map (/.  Y (* Y N)) L))
```

Figure 5.7 A more sophisticated use of an abstraction

The identity of the abstraction **(/. Y (* Y N))** depends on the value for **N** that is input to **xn-everything**. If **N** = 2, then the function behaves exactly as **double-everything.** Notice that though * is a *2-place* function, **(/. Y (* Y N))** is a *1-place* function, since the first input **N** is fixed when **xn-everything** is called.

[29] After Haskell B. Curry, the mathematical logician who is usually credited with inventing this technique, although Schonfinkel was the first to use it.

Higher-order functions are one of the best aspects of functional programming, and their proper use is a hallmark of the experienced functional programmer. In this chapter, we examine the use of higher-order functions through studied examples. In particular we shall see:

1. How to get rid of unnecessary definitions using abstractions.
2. How to take advantage of similarities in different processes to shorten our code.
3. How to pass functions as data.

5.4 Getting Rid of One-Off Functions

Part of the Cartesian product program involved a function **all-pairs-using-X** that paired an element **X** with each element of a list **L**. This program is repeated in figure 5.8.

```
(define cartesian-product
  [ ] _ -> [ ]
  [X | Y] Z -> (append (all-pairs-using-X X Z) (cartesian-product Y Z)))

(define all-pairs-using-X
  _ [ ] -> [ ]
  X [Y | Z] -> [[X Y] | (all-pairs-using-X X Z)])
```

Figure 5.8 The Cartesian product program from the last chapter

all-pairs-using-X is a highly specialised function. Rather than clutter our program with a one-off function like this, we can arrange for the work to be done by an abstraction (figure 5.9).

```
(define cartesian-product
  [ ] _ -> [ ]
  [X | Y] Z -> (append (map (/. E [X E]) Z) (cartesian-product Y Z)))
```

Figure 5.9 The Cartesian product recoded using abstractions

5.5 Taking Advantage of Similarities

Higher-order functions can also help in removing code when two or more programs share properties. Rather than encode the two programs separately, we can draw out the similarities using higher-order functions and so increase the reusability of our code. We shall look at a case study by comparing the **bubble sort algorithm** with **Newton's method of approximations**.

The bubble sort algorithm is probably the simplest method for sorting a list of numbers into ascending or descending order. To sort numbers into descending order (high to low), we pass through the list of numbers comparing each number **X** with its successor **Y** in the list. If **Y** is larger, then we swap the positions **X** and **Y** in the list (this is called *bubbling* since the higher numbers bubble up the list). We keep bubbling until no change can be produced in the list, at which point it is sorted. Here it is in Qi (figure 5.10).🗀

```
(define bubble-sort
   X -> (bubble-again-perhaps (bubble X) X))

(define bubble
  [ ] -> [ ]
  [X] -> [X]
  [X Y | Z] -> [Y | (bubble [X | Z])]           where    (> Y X)
  [X Y | Z] -> [X | (bubble [Y | Z])])

(define bubble-again-perhaps
  \ no change as a result of bubbling - then the job is done \
   X X -> X
  \ else bubble again \
   X _ -> (bubble-sort X))
```

Figure 5.10 Bubble sort in Qi

Now let us look at square roots. How does one compute square roots? The most common way is to use Newton's method of approximations which says that if we want to find the square root of **N**, we take a guess **Guess** and average the result with **(Guess + N)/Guess**. This average will then be a better estimate of the value of √**N** than the original Guess. By repeating the process as many times as desired, we will derive the value of √**N** to a given degree of accuracy.

Suppose we decide to use Newton's method of approximations to derive the value of √**N** to two decimal places. We start our guess at √**N** by guessing that √**N = N/2.0**. Every time we derive a better value for √**N** using Newton's method, we round the better value to two decimal places. Eventually the better value will be different from the old one by a difference

🗀 Qi Programs/Chap5/bubble version 1.qi.

of less than two decimal places, at which point our rounding will produce the same value in both cases. This value will be returned as our value for √**N**. The Qi program is given in figure 5.11.

```
(define newtons-method
  N -> (let Guess (/ N 2.0)
         (run-newtons-method N
           (round-to-2-places (average Guess (/  N Guess))) Guess)))

(define run-newtons-method
  _ Sqrt Sqrt -> Sqrt
  N Better_Guess _
  -> (run-newtons-method  N
       (round-to-2-places (average Better_Guess (/  N Better_Guess)))
         Better_Guess))

(define average
  M N -> (/ (+ M N) 2.0))

(define round-to-2-places
  N -> (/ (round (* 100.0 N)) 100.0))
```

Figure 5.11 Newton's method in Qi

The code for bubble sort and Newton's method share nothing between them, which means that the coding is not optimal because their respective algorithms *do* share something in common. The common element is that they both apply a procedure to their input until no change can be produced in output. To make use of this observation to shorten the code we need the idea of a **fixpoint**.

X is a fixpoint of a function *f* when *f*(X) = X. When *f* is repeatedly applied to Y, and by this means a fixpoint X^Y of *f* is eventually found, we say that X^Y is the fixpoint of *f* derived from Y*.* Figure 5.12 shows the system function **fix,** (as it would appear if written in Qi), that given f and Y, tries to find X^Y.

```
(define fix
  F X -> (fix-help F (F X) X))

(define fix-help
  _ Fix Fix -> Fix
  F X _ -> (fix-help F (F X) X))
```

Figure 5.12 The fixpoint system function in Qi

Using **fix**, the top level of the bubble sort program can be rewritten; the **bubble-again-perhaps** function is now redundant (figure 5.13).🗀

```
(define bubble-sort
   X -> (fix bubble X))
```

Figure 5.13 Rewriting the top level of the bubble sort program

Applying **fix** to Newton's method is a little more difficult since **run-newtons-method** is a 3-place function, not a 1-place function like **bubble.** However the number **N** whose square root we are to find is fixed when **newtons-method** is called and it always stays the same while the program is running. The **newtons-method** function can use a specialised version of **run-newtons-method** for that particular value of **N.** Since **N** is fixed, this specialised function could be a 2-place function.

Still one too many places! However, consider that **run-newtons-method** carries both the old value and the better value around in order to compare them for equality. If we use **fix** to derive the final value, such a comparison is not needed. Since the specialised version of **run-newtons-method** is defined only when **newtons-method** is called, it must be created "on the fly". Creating functions on the fly is a job for abstractions and so we use them to recode the original Qi program (figure 5.14).🗀 The recursion and comparison is driven by **fix** and **run-newtons-method** can be defined using one rule.

```
(define newtons-method
  N -> (fix (/. M (specialised-run-newtons-method N M)) (/ N 2.0)))

(define specialised-run-newtons-method
  M N -> (round-to-2-places (average N (/ M N))))
```

Figure 5.14 Newton's method recoded using abstractions

The combined length of the two revised programs is less than half that of the originals. Higher-order functions can save us a lot of tedious code if we can learn to recognise the cases in which they are applicable. But if we want to seize these opportunities, we must be alert for high-level similarities between algorithms.

🗀 Qi Programs/Chap5/bubble version 2.qi.

🗀Qi Programs/Chap5/newton version 2.qi.

5.6 Using Functions as Data

Higher order functions allow us to treat functions as data just like numbers or lists. This technique, called **procedural attachment,** is used in the next example - how to build a spreadsheet.

Imagine that we want to maintain a list of details about an employee Jim, including his salary and the amount he pays in tax. Suppose that he pays 25% of his wages in tax. If we enter his tax as a cash figure, then if he has a wage increase then the cash figure for tax will be wrong. Business managers cope with problems like this using *spreadsheets*; Jim's tax is stored, not as a number, but as a function that allows the calculation of his tax from other details on the spreadsheet. Our task is to model a spreadsheet in Qi.

Our spreadsheet will consist of a series of *rows*, each row consisting of an *index* and a series of *cells*. The index is a unique name that marks out that row and a cell is composed of an *attribute* (like tax or job) and a *value*. Values can be either *fixed values* (numbers or symbols) or they can be *dependent values* (functions) which when applied to the appropriate inputs, will return fixed values. To assess a spreadsheet, we replace all the dependent values by fixed values.

Figure 5.15 shows a sample spreadsheet in which Jim's wages are pegged to Frank's and his tax is pegged to .8 of whatever Frank pays. Frank is down as earning £20,000 a year and paying .25 of this in tax. We want to evaluate this expression so that the dependent values are replaced by fixed values, giving figures for wage and tax for both Jim and Frank.

```
[[jim [wages (/. Spreadsheet (get frank wages Spreadsheet))]
      [tax (/. Spreadsheet (* (get frank tax Spreadsheet) .8))]]
 [frank  [wages 20000]
  [tax (/. Spreadsheet (* .25 (get frank wages Spreadsheet)))]]]
```

Figure 5.15 A sample spreadsheet in Qi

Figure 5.16 shows a spreadsheet program in Qi.[🗀] The top level function **assess-spreadsheet** works along each row in turn, assigning fixed values to the cells; the entire spreadsheet is carried in this process, since a dependent value may access any part of the spreadsheet. **assign-fixed-values** takes a row and assigns a fixed value to every cell. **assign-fixed-**

[🗀]Qi Programs/Chap5/spreadsheet.qi.

values will receive a cell and a spreadsheet; if the value in the cell is fixed, the cell remains the same. If it is dependent, then the value in the cell is applied to the spreadsheet to derive a fixed value.

The **get** function works along each row, carrying the full spreadsheet with it. We need to do this because a dependent value will access other values before returning a fixed value, and we will then need the spreadsheet again (e.g. Jim's tax accesses Frank's tax which accesses Frank's wages). We cope with this by creating two copies of the spreadsheet, one to work through row by row, and the other to hold in memory in case the whole spreadsheet is needed. This technique, of duplicating inputs to keep one for processing and one to remember, is called **spreading the input**. Finally **get-cell** looks for a cell corresponding to the attribute, and retrieves the value. If the value is fixed, then it is returned, if not then it is applied to the spreadsheet to retrieve the appropriate value.

```
(define assess-spreadsheet
  Spreadsheet -> (map (/. Row (assign-fixed-values Row Spreadsheet))
                      Spreadsheet))

(define assign-fixed-values
   [Index | Cells] Spreadsheet
   -> [Index |   (map (/. Cell (assign-cell-value Cell Spreadsheet)) Cells)])

(define assign-cell-value
   [Attribute Value]  _
          -> [Attribute Value]        where (fixed-value? Value)
   [Attribute Value] Spreadsheet -> [Attribute (Value Spreadsheet)])

(define fixed-value?
  \number?, symbol? and string? are system functions - see appendix A\
   Value -> (or (number? Value) (or (symbol? Value) (string? Value))))

(define get
   Index Attribute Spreadsheet
  -> (get-row Index Attribute Spreadsheet Spreadsheet))

(define get-row
   \ looks for the right row using the index \
   Index Attribute [[Index | Cells] | _] Spreadsheet
   -> (get-cell Attribute Cells Spreadsheet)
   Index Attribute [_ | Rows] Spreadsheet
   -> (get-row Index Attribute Rows Spreadsheet)
   Index _ _ _ -> (error "Index ~A not found" Index))
```

```
(define get-cell
  Attribute [[Attribute Value] | _] Spreadsheet
  -> (if (fixed-value? Value) Value (Value Spreadsheet))
  Attribute [_ | Cells] Spreadsheet
  -> (get-cell Attribute Cells Spreadsheet)
  Attribute _ _  -> (error "Attribute ~A not found" Attribute))
```

Figure 5.16 A spreadsheet program in Qi

Figure 5.17 shows our spreadsheet program in action.

```
(5-) (assess-spreadsheet
      [[jim
        [wages
           (/. Spreadsheet (get frank wages Spreadsheet))]
        [tax (/. Spreadsheet (* (get frank tax Spreadsheet) .8))]]
       [frank
        [wages 20000]
        [tax (/. Spreadsheet (* .25 (get frank wages Spreadsheet)))]]])

[[jim [wages 20000] [tax 4000.0]] [frank [wages 20000] [tax 5000.0]]]
```

Figure 5.17 Assessing a spreadsheet

Exercise 5

1. Define the following higher-order functions.

   a. A function **count** which takes a 1-place boolean function **F** and a list **L** and returns the number of elements in **L** that have the property **F**.
   b. A function **some?** which takes a 1-place boolean function **F** and a list **L** and returns **true** if some element of **L** has the property **F** and **false** otherwise.
   c. A function **all?** which takes a 1-place boolean function **F** and a list **L** and returns **true** if every element of **L** has the property **F** and **false** otherwise.
   d. A function **compose** that receives a non-empty list of 1-place functions and forms their composition. Thus **(compose [(/. X (+ 1 X)) sqrt])** would generate a function that added one to the square-root of a number.
   e. A function **test-speed** that receives a list input **L** and a natural number **N**. The list **L** is composed of a function symbol **F** followed by a series of elements **$X_1,...,X_j$**. **test-speed** should apply **F** to the inputs **$X_1,...,X_j$**, performing this operation **N** times, and timing the whole computation (hint: look at the function **apply** and **time** in Appendix A).

f. A function **partition** that receives a list **L** and an equivalence relation **R** on that list and generates the partition of **L** by **R** as a list of lists.

2. A stream is an object that mimics a possibly infinitely long list and can be represented as a two-element list **[X F].** **X** is the first element in the stream and **F** is the function that applied to any element of the stream gives the next element. So the stream of natural numbers can be represented as **[0 (/. X (+ 1 X))]**.

   a. Write a function**, stream_nth**, that given any stream, gets the nth element of any stream.
   b. Devise a 2-place function **stream_some?** that receives a 1-place boolean function **F** and a stream, returning **true** if there is an object **X** that in the stream such that **(F X) = true**.
   c. If two streams A and B are each infinitely long, then appending them together makes no computational sense because the stream that is tagged at the end will remain inaccessible to **stream-nth**. But we can interleave the streams by generating a stream composed of two-element lists where the nth element of this stream is composed of a list containing the nth element of A and the nth element of B . Write a function **interleave** which does just that.

3. We can make our streams more useful by including a function to test for the termination of a stream. Streams then can be allowed to be of finite length. A stream is now defined as a triple **[X F T]**, where **T** is a test for termination, returning **true** if the element is the end of stream. Any infinite stream can be defined by taking **T** to be **(/. X false)**. The stream of natural numbers is **[0 [(/. X (+ 1 X)) (/. X false)].** For an finite stream, any attempt to take the successor of the final element should generate an error.

   a. Redevise the 2-place function **stream_some?** so that it works with our new streams.
   b. Write a function stream **stream_all?** that receives a 1-place boolean function **F** and a stream, returning **true** if every object **X** in the stream is such that **(F X) = true**.
   c. Rewrite the **interleave** function to work with our new representation.
   d. Write a function **expand** that takes a stream and generates the list that corresponds to it. So **(expand [0 [(/. X (+ 1 X)) (/. X (= X 1000))])** generates the list of numbers **[0 1 2 3 …. 1000].**

4. It is natural to think of lists or streams as data objects and functions like **head** and **tail** as procedures which act on data-objects. But we can change this perception.

   a. Implement the list **[1 2 3]** as a higher-order function that receives the message **head** and returns **1** and the message **tail** and returns the corresponding higher-order function that represents the list **[2 3].** Implement [ ] as a function that returns an error given the message **head** or the message **tail.**

b. Now implement the functions **first** and **rest**. **first** receives the higher-order function that represents the list and derives the first element from it. **rest** receives the higher-order function that represents the list and derives the corresponding higher-order function that represents the tail of the list.

c. Finally define **constr** as the analog of **cons** that builds a new list representing function from an old one. Demonstrate that **(first (constr X Y)) = X**.

5. *For this question you need some background knowledge in the theory of search and some knowledge of resolution theorem-proving. Russell and Norvig (1995, 2002) provides a good introduction to these topics.

   a. Implement *breadth-first search* as a higher-order function **breadth-first** that takes the following inputs.

      i. A *start state* S.

      ii. A *state successor function* O that generates a list of states from a given state.

      iii. A 2-place function R that recognises when states are essentially the same and returns **true** when they are. (R is used to filter out redundant states.)

      iv. A function T that returns either **true** or **false** and that recognises only *goal states*.

      Your function should return either **true** or **false** depending on whether a solution can be reached by breadth-first search.

   b. Test your system by implementing a simple resolution theorem-prover where S is the initial clause set, O maps a set C of clauses to all the possible sets gained by resolving two clauses in C, R recognises when two clause sets are the same (they contain the same clauses) and T recognises a solution (the clause set contains the empty clause).

   c. Implement *depth-first search* as a higher-order function **depth-first** that takes the same inputs as **breadth-first** but omits R.

   d. Implement *hill climbing* and *best-first* search with all the parameters as depth-first plus an evaluation function E that maps a state to a number. Test your program with resolution by setting E to favour a *unit preference strategy.*

Further Reading

Hughes (1990) defends functional programming because of its ability to support higher-order programming. **Abelson** and **Sussman** (1996) has a very good discussion of higher-order programming. The classic reference for sorting algorithms is **Knuth** (1998).

Web Sites

Alejo Hausner (http://www.cs.princeton.edu/~ah/alg_anim/version2/sorts.html) at Princeton includes a description of several classical sorting algorithms (including bubble sort) which are implemented in Java.

6 Assignments

6.1 Simple Assignments

An assignment is an expression whose evaluation causes an object to be associated with a symbol in the environment so that we can call upon this object through invoking the symbol in an appropriate way. These objects are called **global values**. In Qi, there are three kinds of assignments.

1. The simple assignment of values to variables.
2. Assignments that create arrays.
3. Assignments that build property lists.

If *S* is a symbol and *E* an expression, the simple assignment (**set** *S E*) does two things.

1. The normal form *E** of *E* is returned.
2. As a side-effect, *E** becomes the value of the **global variable** *S;* so that the
   expression (**value** *S*) will return *E**.

Figure 6.1 gives a short script that uses **set** and **value**.

```
(0-) (set *number* 6)
6

(1-) (value *number*)
6

(2-) (define f
        X -> (+ X (value *number*)))
f

(3-) (f 6)
12
```

```
(4-) (set *number* 5)
5

(5-) (f 6)
11
```

Figure 6.1 Using simple assignments

Notice that a function that uses global values can return different results for the same input if the value of the global variable is changed.

Although conventionally, when computer scientists talk about assignment statements, they generally do not include function definitions under that category, the use of **define** is, strictly speaking, another example of an assignment. The effect of **define** is to change the environment in which the computation is carried out and in that sense it is an assignment. The important difference between function definitions and other assignments is that, generally, function definitions do not and cannot change during the execution of a program, whereas simple assignments can change the value assigned. The basic similarity between assignments and function definitions can be brought out by assigning an abstraction to a global variable. The variable can then be used to invoke the function in almost the same way as a regular function definition. Figure 6.2 shows how a simple assignment can do this.

```
(0-) (set *factorial*
      (/. X (if (= X 0) 1 (* X ((value *factorial*) (- X 1))))))
#<CLOSURE :LAMBDA [X]
  [if [= X 0] 1 [* X [apply [value '*factorial*] [- X 1]]]]>

(1-) ((value *factorial*) 9)
362880
```

Figure 6.2 Using a simple assignment to hold an abstraction

Simple assignments can make programs shorter, if they are used sensibly. For example, the **get** function of the previous chapter required the spreading of the input, **Spreadsheet**, in order to retain some memory of what the entire spreadsheet was like. However, if we assign the spreadsheet to a global variable (called ***spreadsheet***), then this spreading is unnecessary and the program sheds an auxiliary function and becomes simpler (figure 6.3).

```
(define get
  Index Attribute [[Index | Cells] | _]
  -> (get-cell Attribute Cells)
  Index Attribute [_ | Rows] -> (get Index Attribute Rows)
  Index _ _ -> (error "Index ~A not found" Index))

(define get-cell
  Attribute [[Attribute Value] | _]
  -> (if (fixed-value? Value) Value (Value (value *spreadsheet*)))
  Attribute [_ | Cells]  -> (get-cell Attribute Cells)
  Attribute _  -> (error "Attribute ~A not found" Attribute))
```

Figure 6.3 Using simple assignments in place of spreading

6.2 Arrays

The function **make-array** receives a list *L* of positive integers; the length *l* of *L* is the number of dimensions of the new array *A.* For each N_i ($1 \leq i \leq l$) of L, N_i+1 is the size of the *i*th dimension of *A*. The function **make-array** then constructs the array. For example the expression **(set *ordinance_survey* (make-array [999 999]))** will construct an array of dimensions 1000 × 1000 (the origin of each dimension is 0). To assign a value to an array address we use **put-array**. This function receives three inputs; *A*, *L* and *E*.

1. The array *A*.
2. *L* is a list of non-negative integers; each integer N_i of *L* must not be greater than the size of the *i*th dimension of *A*.
3. The expression *E*.

The function **put-array** then puts the normal form *E** of *E* into the array address designated by *L* and returns *E** as a value. For example, having created the array ***ordinance_survey***, I can place the name **holmfirth** in the address **536, 567** of **ordinance_survey** by evaluating **(put-array (value *ordinance_survey*) [536 567] holmfirth).** To retrieve an object from an array, the function **get-array** is used. This receives exactly the same kind of inputs as **put-array**;

1. An array *A*.
2. A list *L* of non-negative integers; each integer N_i of L not greater than the size of the *i*th dimension of *A*.
3. An expression *E*.

The function **get-array** then returns the element E_i stored at the address in A designated by L, if anything exists at that address or the normal form of E, if nothing exists at that address. So having placed **holmfirth** in address **536, 537** of the array **ordinance_survey**, I can retrieve it by evaluating **(get-array (value *ordinance_survey*) [536 567] unknown).** Supposing I type **(get-array (value *ordinance_survey*) [999 999] unknown)**, and nothing is stored at address **999, 999, unknown** is returned.

6.3 Property Lists

Arrays give constant time access, in contrast to lists which often require linear time searches through the elements of the list to retrieve the needed item. The disadvantage of an array is its inflexibility; if the items are not indexed by their addresses, then searching an array is no faster than searching a list. Property lists offer an alternative; being nearly as fast as arrays, but much more flexible in allowing non-integer indexes. The function **put-prop** receives:-

1. A symbol *S*.
2. An integer or symbol *P* which is the name of a pointer to
3. an expression *E*.

and

1. Creates a pointer called *P* from *S* to the normal form *E** of *E*. This is a side-effect.
2. Returns *E** as the value.

The function **get-prop** receives

1. A symbol *S*.
2. An integer or symbol *P.*
3. An expression *E* and returns ….
4. the expression *F*, if there is a pointer *P* from *S* to *F,* or the normal form of *E* otherwise.

Thus the expression **(get-prop mark sex unknown)**, entered at the very beginning of an Qi session will evaluate to **unknown (**since no pointer **sex** has been created from **mark** to anything). If **(put-prop mark sex male)** is entered then **male** will be returned. Repeating **(get-prop mark sex unknown)** will now return **male.**

6.4 Building Semantic Nets

Semantic nets are a fun way of illustrating the usefulness of property lists. They were introduced into Artificial Intelligence by **Quillian** (1968) as a model for the way humans hold information about objects and the relations between objects. Semantic nets are usually represented by **labelled directed graphs**, that is, as diagrams consisting of arrows (called **arcs)** with names or **labels** attached to the arrows. These arrows connect points or **vertices** that also have names.

A semantic net which records the fact that I (Mark Tarver) am a man might consist of an arc labelled **is_a** leading from the vertex **mark_tarver** to a vertex **man.**

$$\begin{tikzcd} \textbf{mark\_tarver} \arrow[r, "\textbf{is\_a}"] & \textbf{man} \end{tikzcd}$$

A statement of the form "All Xs are Ys" can be viewed as stating a relation between concepts; so "All men are human" becomes:-

$$\begin{tikzcd} \textbf{man} \arrow[r, "\textbf{type\_of}"] & \textbf{human} \end{tikzcd}$$

which states that the concept **man** is a **subtype** of the type of **human**, or (same thing) that **human** is a **supertype** of the type of **man.**

From "Mark Tarver is a man" and "All men are human" we can derive "Mark Tarver is human". Semantic nets allow for simple inferences like this by means of a search. Such a search starts from the **mark_tarver** vertex and attempts to reach the vertex **human**, travelling in the direction of the arrows. If it succeeds it returns **yes** otherwise it returns **no.**

Property lists are very good at representing the links in a semantic net. From **mark_tarver** we construct a pointer that points to all the concepts $C_1,...,C_n$ which Mark Tarver is an instance. We call this pointer **is_a.** From each $C_1,..,C_n$ we can (if we want) create pointers in the same way, called **type_of** to show the supertypes of $C_1,..,C_n$.

Figuratively speaking, to find if Mark Tarver belongs, or falls under, a concept C, we place **mark_tarver** in a 'box', and then we add to the box all the concepts C_1, ...,C_n under which Mark Tarver falls in virtue of the **is_a** relation (i.e. all the concepts pointed to from **mark_tarver** by the pointer **is_a**). Then for each of $C_1,...,C_n$, we add to the box every concept C_j which is a supertype of some C_i already in the box. To do this we use the **type_of** pointer. We repeat this process until the box cannot be filled any more; at which point we look in the box to see if C is in the box. If it is, then Mark Tarver supposedly falls the under the concept C and the answer is “yes”. If not, the answer is “no”. The program that computes this algorithm is given in figure 6.4; our box is a list that initially contains one object (**[Object]**), which is where our query begins.[🗀]

```
(define query
  [is Object Concept] -> (if (belongs? Object Concept) yes no))

(define belongs?
  Object Concept -> (element? Concept (fix spread-activation [Object])))

(define spread-activation
  [ ] -> [ ]
  [Vertex  |  Vertices]
 -> (union (accessible-from Vertex) (spread-activation Vertices)))

(define accessible-from
  Vertex -> [Vertex |  (union (is_links Vertex) (type_links Vertex))])

(define is_links
  Vertex -> (get-prop Vertex is_a []))

(define type_links
  Vertex -> (get-prop Vertex type_of []))

(define assert
  [Object is_a Type]  -> (put-prop Object is_a [Type | (is_links Object)])
  [Type1 type_of Type2]
                    -> (put-prop Type1 type_of [Type2 | (type_links Type1)]))
```

Figure 6.4 A simple semantic net in Qi

We can test this program.

```
(1-) (assert [Mark_Tarver is_a man])
[man]

(2-) (assert [man type_of human])
[human]
```

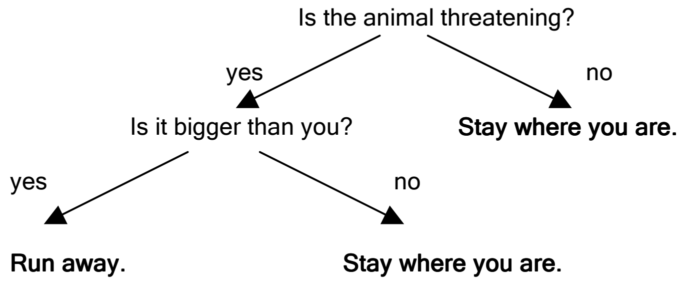
[🗀] Qi Programs/Chap6/semantic net.qi. **union** and **element?** are system functions, see appendix A for their definition.

```
(3-) (query [is Mark_Tarver human])
yes
```

Figure 6.5 Using a semantic net to record information and ask questions

Exercise 6

1. *A **binary decision tree** poses a series of yes/no questions about the nature of an object and returns a verdict in its leaves. For example,



   Binary decision trees can be adapted to provide **hashing functions** where the answers to the decision tree provide a binary number. Thus given a lion as an input to the above tree, the answers to the questions "Is the animal threatening?" and "Is it bigger than you?" are "yes" and "yes". Treating "yes" as 1 and "no" as 0, the binary number for this input is 11. Devise a system that represents a binary decision tree as a tree whose nodes are 1-place boolean functions. Your system should allow an tree to be 'applied' to an object to generate a binary number. Use the binary number as an index to a 1-dimensional array, so that the object is stored with a list of similar objects in that array address. Thus **lion** would be stored in the address 3 of the array and **hamster** in address 0.

2. Define a function **retract** that allows you to remove assertions from the semantic net, so that stating **(retract [Mark_Tarver is_a man])** removes the corresponding assertion.

3. *Rewrite the function **assert** to include the backpointers **instance_of** and **super_type** so that if "Mark Tarver is a man" is stated then a backpointer **instance_of** then points from **man** to a list containing Mark Tarver. Similarly if **[man type_of human]** is asserted then a backpointer **super_type** points from **man** to a list containing **human.** Amend the query program to allow the semantic net to cope with questions like **(query [what is human]),** so that it returns the list of all the humans of which it knows.

4. Amend the program so it can answer questions of the form "Why do you think that?". For instance, given the input **(query [why Mark_Tarver human])**, the program should explain that it considers I am a man because any man is human and I am a man.

5. Extend the program to cope with disjunctive and conjunctive queries. For instance, **(query [[Joe is_a man] or [Mark_Tarver is_a man]])** should return **yes. (query [[Joe is_a man] and [Mark_Tarver is_a man]])** should return **no.**

6. *Extend the program to cope with conditional queries. For instance, **(query [if [Joe is_a man] then [Joe is_a human]])** should return **yes**. To do this, you might want to **assert** the antecedent of the query during query time and test for the conclusion, and then **retract** the antecedent at the end of the computation.

7. *Extend the program to cope with conditional assertions. For example, **(assert [[if [Joe is_a Mason] then [Joe is_a rich]]).** One way of doing this is to extend the semantic net by adding a **production system**[30] which contains a rule like "During a query, if **[Joe is_a Mason]** is derivable then assert **[Joe is_a rich]**".

8. Extend the program to cope with conjunctive assertions like **(assert [[man type_of human] and [human type_of animal]])**.

9. *Extend the program to cope with disjunctive assertions like **(assert [[man type_of human] or [human type_of animal]])**. One way of doing this is to split the network into two, one containing the assertion **[man type_of human]** and the other **[human type_of animal].** The **query** program returns an answer just when the answer follows from both semantic nets.

10. *One problem with splitting the net in the manner of 7. is that the number of semantic nets rises exponentially in relation to the number of disjuncts asserted. Can you rebuild your program so that this splitting takes place dynamically at query time rather than at assertion time?

11. *Our semantic net can cope with 2-place predicates if we see such predicates as being true or false of pairs of objects. Pairs can be represented as two element lists. So "Mark is taller than Dave" becomes **(assert [[Mark Dave] is_a taller])**. Relations have properties too, transitivity being a property of the relation 'taller than'. Extend the semantic net program so that **(assert [taller type_of transitive])** can be entered. The program should be able to infer "Mark is taller than Bill" from "Mark is taller than Dave" and "Dave is taller than

[30] A production system is a series of rules of the form 'If situation S obtains, then perform action A'. Some production systems have a **conflict resolution strategy** which decides which rule to use if more than one rule applies.

Bill". To do this, you might want to extend the production system to include rules of the form "If 'X is taller than Y' is derived and 'Y is taller than Z' is derived then derive 'X is taller than Z' ". Do the same for symmetrical relations.

12. Our program has no idea of negation. We can handle this by adding a pointer called **is_not**. Thus **(assert [Mark_Tarver is_not woman])** asserts I am not a woman. Extend the semantic net so that it processes queries involving negation. To yes/no questions like **(query [Joe is_a man]])**, the net now has 3 possible responses (a) **yes** ('Joe is a man' is derivable) (b) **no** ('Joe is not a man' is derivable) or (c) **unknown**, **(**neither 'Joe is a man' nor 'Joe is not a man' is derivable).

13. *The **frame problem** is a famous problem in Artificial Intelligence. It revolves around the idea of *change.* For example, suppose we are told "Mark Tarver is wearing a hat" and "Mark Tarver is in his office". The answer to the query "Is Mark Tarver's hat in his office" is "Yes". However, supposing I move to my car, the answer to the question becomes "No". The frame problem is the problem that an intelligent system faces when it is forced to represent changing information, where a change in the the truth-value one item of information affects another. Build into your semantic net the techniques used to build the spreadsheet program of chapter 5. Show your system can cope with the kind of query about wages and tax that the spreadsheet program could solve.

Further Reading

Quillian (1968) is the original reference for semantic nets. **Hendrix** (1979) developed an advanced version of semantic nets, called **partitioned semantic nets**, which allows for mixed quantifiers. **Minsky** (1975) suggested putting more structure into the nodes, so that instead of being simple symbols they could consist of a series of slots with values in them; such a system is called a **frame system**. **Frames** are designed to represent **natural kinds**; that is, sets of objects that share a cluster of common qualities. Sophisticated frame systems often make use of the techniques of procedural attachment discussed in chapter 5. There have been several knowledge representation languages, based on frames, the original being **KRL** (**Bobrow** and **Winograd** (1977)), and later versions such as **CYCL** (**Lenat** and **Guha** (1991)). CYCL is a highly sophisticated frame language. General introductions to AI generally include sections on semantic nets and frames; **Russell** and **Norvig** (2002) provide a good overview.

Web Sites

http://members.aol.com/mind2ls/semanticnet.htm runs a set of links to resources on semantic nets.

7 Non-Determinism

7.1 Non-deterministic Algorithms

An important class of algorithms is grouped under the heading of **non-deterministic algorithms**. In a deterministic algorithm, every step is determined and there is no possibility of choice. Non-deterministic algorithms present a series of steps in which, at some point, there is an undetermined choice as to what step to take out of a set of possibilities, and the algorithm leaves it to us to determine how the right choice is made. The stage of a non-deterministic algorithm where such a choice arises is called a **choice point**.

The problem in programming a non-deterministic algorithm into a computer is that computers do not *choose* anything; they only do what they are told to do. One way of representing non-determinism to a computer is to get it to order the choices available and to try each one in turn. If a choice turns out to be unsuccessful, then the computer **backtracks** to the choice point and tries the next choice in the ordering. In this chapter, we will see how to do this in Qi through an examination of **backward chaining**.

7.2 Backward Chaining

Suppose we given a set of assumptions to work with, where each assumption comes in one of three forms.

1. *It is a statement that is symbolised by a symbol of some kind.*

So the statement “Water is composed of hydrogen and oxygen” could be represented by p. Alternatively, we could adopt for “Water is composed of hydrogen and oxygen” some symbol that gives some clue as to what it symbolises (e.g. p_{water}, or even water-is-composed-of-hydrogen-and-oxygen). The point is that the internal structure of the symbol is logically unimportant, rather like the *x*s and *y*s of algebra. We call these kinds of statement **atomic statements** or simply **atoms**.

2. *It is a statement of the form* (P & Q) *where* P *and* Q *do not contain* ←.

This kind of statement is read as "P and Q both hold". Call these statements **conjunctions**.

3. *It is a statement of the form* (P ← Q), *where* P *is an atom and* Q *is an atom or a conjunction.*

This kind of statement is read as "P holds if Q holds" or "If Q is true, so is P". Call these statements **implications**.

Now suppose that every conclusion we wish to prove is either an atom or a conjunction, but which never contains any implications (i.e. ← is never found in a conclusion) and every assumption is either an implication or an atom. We now have a simple formal system for performing limited kinds of reasoning, **Maier** and **Warren** (1988) call this simple system **Proplog**. A basic pattern of valid inference in Proplog is the following.

Inference Pattern #1

Given, as an assumption, an implication (P ← Q), you can prove P if you can prove Q.

In addition, we have two more inference rules.

Inference Pattern #2

You can prove (P & Q) if you can prove P and you can prove Q.

Inference Pattern #3

Given an assumption P, you can prove P.

These three simple patterns of inference can be combined within a simple procedure for proving a conclusion C from a list of assumptions Δ that are either atoms or implications.

A Backward Chaining Proof Procedure for Proplog

Do until success or failure

1. Begin with the conclusion P you are trying to prove.
2. If P is in Δ then P is proved (inference pattern #3). If there are no more conclusions to prove, signal success and then stop. Otherwise, attempt to prove the next conclusion.
3. If the conclusion is a conjunction (P & Q), prove P from Δ and prove Q from Δ (inference pattern #2).
4. If P is not in Δ but (P ← Q) is in Δ, try to prove Q (inference pattern #1).
5. If P is an atom and P is not in Δ, and (P ← Q) is not in Δ, signal a failure and then stop.

This is a method of proving conclusions in Proplog. This **proof procedure** works by starting with what has to be proved and working backwards until a conclusion is reached that can be directly proved from Δ by inference pattern #3. This particular proof procedure is called **backward chaining**.

Here is a simple schematic problem to illustrate the approach. The assumptions are

(q ← p), (q ← r), (r ← (s & t)), (r ← u), (r ← (s & v)), v, s.

We can prove q by backward chaining as follows:-

```
Begin with q.
  Prove q by proving r.
    Prove r by proving (s & v).
      Prove s.
      Prove v.
      Signal success and halt.
```

A proof like this is sometimes represented as a **proof tree** (figure 7.1) in which the original problem is the root of the tree. The elements of the tree are **goals** (things we are trying to solve). When a goal is solved, we draw a line from it to a box □ which marks it as a **success node**. When a goal is replaced by **subgoal**(s) (as r is replaced by the subgoal (s & v), we draw lines from the goal to the subgoals.

$$\begin{tikzcd}
& q \arrow[d] & \\
& r \arrow[d] & \\
& (s \,\&\, v) \arrow[dl] \arrow[dr] & \\
s \arrow[d] & & v \arrow[d] \\
\square & & \square
\end{tikzcd}$$

Figure 7.1 A completed proof tree

The snag comes in programming the computer to do proofs like this, because in the above proof we unconsciously avoided the obvious dead ends (like trying to prove q by using q ← p (figure 7.2)). These dead-ends are called **failure nodes** and are sometimes marked by a single daughter node ■. A failure node is a node of the proof tree which cannot be closed with a □, nor can it be extended to produce subgoals. A proof tree cannot count as a completed proof if it contains a failure node.

$$\begin{tikzcd}
q \arrow[d] \\
p \arrow[d] \\
\blacksquare
\end{tikzcd}$$

Figure 7.2 A proof tree with a failure node

Since the computer has no intelligence of its own, we need to program some technique into it to avoid being stuck in failure nodes. The problem is that our proof procedure is non-deterministic; it assumes that the right choice of rule is always made without providing any means of finding it. The solution we will follow is the one suggested in the opening of this chapter; we will order all the choices open to the computer and try each one in the ordering, backtracking if needed to try a new choice. The backtracking we will use will backtrack to the last point at which a goal was solved and look for new ways to solve it - such a form of backtracking is called **chronological backtracking**. But first, we have to see how to represent proofs of this kind to the computer.

7.3 Representing Proof Trees

At any stage in a backwards chaining proof, the computer will be trying to prove one or more goals. A simple way of keeping track of them would be to place them all in a list. It is a matter of choice which goal we try to prove, but a sensible strategy works from the front of the list, adding subgoals to the front as they arise. Such a list is a **stack**; i.e. a series of elements in which elements are always added (pushed) or removed (popped) from the front. A stack to keep track of goals in a proof is a **goal stack**. Every proof begins with the initial problem as the only goal on the goal stack. At any stage in the proof, the only available goal is the goal at the front of the stack.

In our Proplog example, if a goal is removed using inference pattern #3, the stack gets shorter by one element. If a goal (P & Q) is removed by inference pattern #2, then the subgoals P and Q are pushed on to the goal stack in reverse order (i.e. Q first, P last). The new stack is longer by one element (P being now the first goal of the new stack). If inference pattern #2 is used, then we have a goal P and a rule in our assumptions (P ← Q). The goal P is popped from the head of the stack and the subgoal Q is pushed in its place. If the goal stack is empty then the proof is complete.

It is important to see the difference between a proof tree and a goal stack. A proof tree keeps the entire history of the proof process, including the goal that started the proof off. The goal stack only keeps those goals that remain to be solved. At the end of the proof, the proof tree is a record of the way the proof was done. At the end of proof, the goal stack is merely an empty stack. A proof tree does not correspond to a single stack, but rather a series of stacks, where each stack results from its predecessor by some act of inference. A goal stack is thus a snapshot of the leaves of a growing proof tree. This is illustrated in figure 7.3, which shows how a proof tree changes and grows in relation to the goal stack.

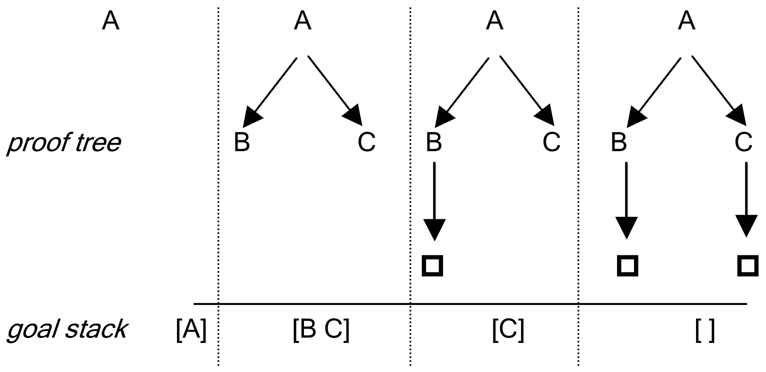


Figure 7.3 Representing the state of a proof tree using a stack

The computational advantages of a goal stack are that it is simple to represent as a list, and efficient in terms of memory usage, since it does not retain those parts of the proof tree which are irrelevant to the completion of the proof.

7.4 Chronological Backtracking in Qi

Implementing a backward chaining inference engine of the kind needed is a ten line program in Qi. This economy of expression is possible because Qi has a built-in capacity to support backtracking. We can designate a choice point by marking the rewrite rule that makes the choice for us. If this rewrite rule makes the wrong choice, we arrange for a special object to be returned called the **failure object**. When the failure object is returned, Qi unwinds the computation back to the point at which the choice was made and tries the next rewrite rule (if any) after the marked rule.

Rewrite rules that mark choice points are signalled by the use of **<-** instead of **->** within a function definition. The failure object in Qi is **#\Escape**. Using backtracking, it is easy to implement our inference engine. To make things even easier for us, we can amalgamate atoms and rules. The atom s is represented as the rule (s ←) (i.e. s can be derived from nothing) and ← is represented as **<=**. This gives economy in syntax and shortens the program (figure 7.4).🗀

🗀 Qi Programs/Chap7/proplog version 1.qi.

```
(define backchain
  Conc Assumptions  -> (backchain* [Conc] Assumptions Assumptions))

(define backchain*
  [] _ _ -> proved
  [[P & Q] | Goals] _ Assumptions
  -> (backchain* [P Q | Goals] Assumptions Assumptions)
  [P | Goals] [[P <= | Subgoals] | _] Assumptions
  <- (backchain* (append Subgoals Goals) Assumptions Assumptions)
  Goals [_ | Rest] Assumptions -> (backchain* Goals Rest Assumptions)
  _ _ _ -> #\Escape)
```

Figure 7.4 Backward chaining implemented for Proplog in Qi

The top level of the program **backchain**, receives the conclusion to be proved and a list of assumptions. Control is passed to **backchain*** which places the conclusion in a stack (**[Conc]**) and spreads the **Assumptions**; one version to process and one to remember the original.

backchain* attempts to solve the problem. First, if the goal stack is empty, **proved** is returned - the problem is solved. Next, if the goal is a conjunction, we add each part of the conjunction to the goal stack. Third, if the leading goal is the atom at the head of the first implication in the list of assumptions, (**[P | Goals] [[P <= Subgoals]**) then the first inference pattern is applied. That is, the goal **P** is removed and the subgoals **Subgoals** are added to the goal stack and the whole goal stack is passed to **backchain*** **(...(backchain* (append Subgoals Goals) Assumptions Assumptions)).** The original list of assumptions is restored so that **backchain*** returns to the top of the list of assumptions looking for new ways to solve the new goal stack.

But notice that **(backchain* (append Subgoals Goals) Assumptions Assumptions)** is preceded by **<-,** so that if this expression evaluates to the failure object **#\Escape**, then Qi will backtrack to the next line of **backchain***. This is the line **Goals [_ | Rest] Assumptions -> (backchain* Goals Rest Assumptions)**, in which **backchain*** recurses down the list of assumptions looking for another implication - which is what we want. Finally, in the last line, since no other options are open, **#\Escape** is returned. Here is an example.

```
(5-) (backchain q [[q <= p] [q <= r] [r <=]])
proved

(6-) (backchain q [[q <= p] [q <= r]])
#\Escape
```

7.5 Soundness and Completeness

Our little program is an example of an **inference engine** or **theorem-prover** for Proplog. A theorem-prover is a program that is designed to automate the process by which conclusions are drawn from assumptions. Let us write $\Delta \vdash_{Proplog}$ P to mean that from a list of assumptions Δ, the conclusion P can be derived by the rules of inference for Proplog (i.e. the inference patterns #1, #2 and #3). Let us write $\Delta \vdash_{program}$ P when P can be derived from Δ using our program. What is the relation between $\vdash_{Proplog}$ and $\vdash_{program}$? One desirable relations is

If $\Delta \vdash_{program}$ P then $\Delta \vdash_{Proplog}$ P

This should certainly be true for any Δ and P; if our program says that P follows from Δ, then, according to the rules of inference for Proplog, we should be able to derive P from Δ. A theorem-prover which has this property is said to be **sound**. The converse relation is

If $\Delta \vdash_{Proplog}$ P then $\Delta \vdash_{program}$ P

A theorem-prover with this property is said to be **complete**. Backward chaining with chronological backtracking is complete as a proof procedure provided that:-

1. there is no proof tree which has a branch which can be grown to infinity and …
2. … at any stage in a proof there are only a finite number of ways that the proof tree can be grown.

Our theorem-prover for Proplog is sound but incomplete because condition 1. is not met. For example, the problem

```
(backchain q  [[q <= p] [q <= r] [r <= q] [r <= s t]
               [r <= u] [r <= s v] [v <=] [s <=]])
```

causes an infinite loop, even though the conclusion is derivable from the assumptions according to the rules of inference of Proplog. The reason why is that assumptions #2 and #3 in this problem, namely **[q <= r] and [r <= q],** are used by our theorem-prover to backward chain from q to form a branch which can be infinitely extended.

$$\begin{tikzcd} q \arrow[r] & r \arrow[r] & q \arrow[r] & r \arrow[r] & q \arrow[r] & r \ldots\ldots \end{tikzcd}$$

The construction of a complete theorem-prover for Proplog is left as an exercise at the end of this chapter.

7.6 Writing Refinement Rules: a Preview

We have approached the idea of proof in Proplog in an informal manner. But when we wish to formally define the conditions under which a statement P can be proved, we lay down a set Γ of **refinement rules**. P is provable just when P follows from the refinement rules in Γ. Actually, when we say that P is *provable*, this is really a shorthand way of saying that P is provable from some set Δ of assumptions according to some set Γ of refinement rules. A pair composed of a set of assumptions Δ and a conclusion P is called a **sequent**, and we represent this as $\Delta >> P$.[31] We say that a sequent is **provable** just when we can prove P from Δ using the refinement rules for Γ.

We write "$\Delta \vdash_\Gamma P$" to mean "P is provable from Δ according to the refinement rules of Γ". $\vdash_\Gamma$ is our provability relation.[32] The job of a set of refinement rules is to define this relation of provability and so determine exactly what sequents are provable and what are not. Suppose we return to Proplog. Our first rule of inference was

Inference Pattern #1

Given, as an assumption, an implication of the form $(P \leftarrow Q)$, you can prove P if you can prove Q.

In other words, given any sequent $\Delta >> P$ to prove, where $(P \leftarrow Q) \in \Delta$, this sequent can be proved if the sequent $\Delta >> Q$ can be proved. We can write this as

[31] The use of sequent notation derives from **Gerhard Gentzen** (1934) who developed the **sequent calculus treatment** of first-order logic. In Gerhard's system a sequent is a pair of lists $F_1,\ldots,F_n$ and $G_1,\ldots,G_n$ called the **antecedent** and the **succeedent** respectively. The sequent is valid if the conjunction of the elements of the antecedent logically implies the disjunction of the elements of the consequent. If the succeedent is restricted by only allowing one element in the list, then the system is a **single-conclusion** sequent calculus system; which is the kind that Qi is designed to represent. See Duffy (1991) for more details.

[32] >> and |- do mean different things. Thus 'P |- P' states "From P, P is derivable" whereas "P >> P" states nothing, it is simply a representation of an object without any claims as to derivability. But where there is no danger of confusion, we will not press the point.

where (P ← Q) ∈ Δ

$$\frac{\Delta >> Q;}{\Delta >> P;}$$

The convention is that a line ________ is drawn between what is to be proved (which goes below the line) and what is needed to prove it (which goes above the line). We may also add special conditions (written in a quasi-formal English) like "where (P ← Q) ∈ Δ" to explain our refinement rule. These special conditions are known as **side conditions** to the refinement rule. The P and Q are variables that stand for any Proplog formulae. Rather than including "(P ← Q) ∈ Δ" as a side condition, it is more usual to place the "(P ← Q)" to the left of the >> that occurs below the line.

$$\frac{(P \leftarrow Q), \Delta >> Q;}{(P \leftarrow Q), \Delta >> P;}$$

The refinement rule indicates that (P ← Q) must be present in the assumptions for the refinement rule to apply. The "Δ" is simply a reminder that there may be other assumptions present. Once we are used to rendering these refinement rules, such a reminder is generally not necessary and so we can simply omit "Δ" and write

$$\frac{(P \leftarrow Q) >> Q;}{(P \leftarrow Q) >> P;}$$

Now for the second inference pattern.

Inference Pattern #2

You can prove (P & Q) if you can prove P and you can prove Q.

The kind of sequent that inference pattern #2 is used to prove is one of the form Δ >> (P & Q). Thus the form of refinement rule determined by this inference pattern is

$$\frac{\dots\dots\dots}{\Delta >> (P \,\&\, Q);}$$

where ……………. remains to be filled in. The inference pattern says Δ >> (P & Q) can be solved if Δ >> P and Δ >> Q can be solved, so ……………. is filled in by Δ >> P; Δ >> Q.

$$\frac{\Delta \gg P;\ \Delta \gg Q;}{\Delta \gg (P \,\&\, Q);}$$

Finally we can simply drop the Δ, taking it as read and simply write

$$\frac{P;\ Q;}{(P \,\&\, Q);}$$

Inference Pattern #3

Given an assumption P, you can prove P.
This is written

$$\frac{}{P, \Delta \gg\ P;}$$

There are no schemes above the line _______ in this case, since this inference pattern allows the unconditional solution of sequents of a certain pattern. Again Δ can be dropped and the refinement rule can be written as

$$\frac{}{P \gg P;}$$

Refinement rules are standard notation for expressing the concept of provability in many different proof systems and they play an important part in this book. We shall return to them again in chapter 11.

7.7 Selecting the Failure Object

By default the failure object is **#\Escape**. **fail-if** is a way of invoking backtracking which allows us to choose the nature of the failure object. **fail-if** is a higher-order function that takes

1. A 1-place function *f*.
2. An expression *E*.

and then returns

1. The failure object if (*f E*) evaluates to true.
2. The normal form of *E* if (*f E*) evaluates to false.

To illustrate the use of **fail-if**, suppose we have written two programs to solve problems in abstract algebra. The first uses intelligent techniques that mimic the way a mathematician might think; the second uses brute force and computer speed. In each case the sign of success is that the machine reduces problem to one of self-identity (so the problem ($x \circ x^{-1}$) = e can be reduced e = e). We observe that some problems that evade the first approach fall to second, and so we resolve to combine the two programs. We attempt to use intelligent techniques if possible, and if not, to backtrack to the start and using brute force. The sign of failure on the part of our "intelligent" program that it fails to return a self-identity. Thus, it is useful to change the failure object so that a non-self-identity object would cause failure and trigger backtracking. **fail-if** supplies this need. The top level of our program appears in figure 7.5.

```
(define solve-abstract-algebra
   Equation <- (fail-if (/. X (not (self-identity X)))
                              (try-intelligent-techniques Equation))
   Equation -> (try-brute-force Equation))

(define self-identity
  [X = X] -> true
  _ -> false)
```

Figure 7.5 The top level of an abstract algebra program using **fail-if**

fail-if is a powerful means of controlling backtracking which allows the elements of an infinitely large class to count as failure objects. In the Proplog program, we relied on having an explicit goal stack to hold the unsolved goals. Another way of writing the same program would be to split goals of the form P & Q into two separate processes - a process that tries to prove P and a process that tries to prove Q. It is logical to "and" these processes together so that if the attempt to prove P succeeds, **true** is returned and likewise for Q. If either attempt fails, then we return **false**. This means that failure is not a matter of returning **#\Escape**, but **false**. Therefore, we change the identity of the failure object to make **false** the failure object by using **fail-if** (figure 7.6).🗀 It is now convenient to represent the atom p as **p** instead of **[p <=].**

🗀 Qi Programs/Chap7/proplog version 2.qi

```
(define backchain
  Conc Assumptions   -> (backchain* Conc Assumptions Assumptions))

(define backchain*
  P [P | _] _ -> true
  [P & Q] _ Assumptions
  -> (and (backchain* P Assumptions Assumptions)
        (backchain* Q Assumptions Assumptions))
  P [[P <= Q] | _] Assumptions
  <- (fail-if (/. X (= X false)) (backchain* Q Assumptions Assumptions))
  P [_ | Rest] Assumptions  -> (backchain* P Rest Assumptions)
  _ _ _ -> false)
```

Figure 7.6 Backward chaining for Proplog implemented in Qi *using fail-if*

7.8 The Static Elimination of Backtracking in Qi

The backtracking constructions that use **<-** are **statically eliminable**; that is, in the absence of side-effects, they can be compiled away without losing any of the meaning of these constructions. A backtracking construction using **<-** without **fail-if** can be eliminated through the following equivalence.

$$P_1,\ldots,P_n \texttt{ <- } R \equiv P_1,\ldots,P_n \texttt{ -> } R \qquad \textbf{where (not (= R \#\textbackslash Escape))}$$

Whereas a backtracking construction that uses **fail-if** can be eliminated through this equivalence.

$$P_1,\ldots,P_n \texttt{ <- (fail-if F R)} \equiv P_1,\ldots,P_n \texttt{ -> } R \qquad \textbf{where (not (F R))}$$

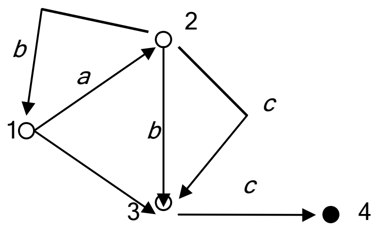
The declarative meaning of the program is not changed in this transformation, but the performance is affected. The result **R** returned from each rewrite rule is evaluated twice in the right-hand side of the equivalence, but only once on the left hand side. This means that these equivalences, as they stand, do not provide a tenable model for compiling backtracking constructions. In order to serve this role, the compiled code needs to memoise the normal form of **R** and use it again. In the internals of the compilation of Qi, this is exactly what happens.[33]

[33] The significance of these equivalences is that, because they preserve the declarative meaning of the backtracking construction, when we come to consider how Qi functions are to be type-checked in part II of this book, these equivalences will allow the elimination of backtracking constructions in favour of a more familiar construction.

Exercise 7

1. Write a complete theorem-prover for Proplog.

2. Bratko (1990) "A **non-deterministic finite state machine** (NDFSM) is a machine that reads a series of symbols and decides whether to accept or reject that series. A NDFSM has a number of states and upon receipt of a symbol will change from being in one state to being in another. There is a privileged set of states called **final states** and if the automaton is in one of these states when it finishes reading a series then it accepts that series, otherwise it rejects it. If there is one state that has to be chosen in order for the whole list to be accepted, then the NDFSM will choose that state and not one that will cause the list to be rejected." The operation of a NDFSM can be represented by a labelled directed graph in which the nodes of the graph represent the states of the NDFSM, and the labels on the arcs are the inputs required to get the NDFSM to jump from one state to the next. The figure below shows a NDFSM as a labelled directed graph.



   There is one final state in this NDFSM, state 4. Initially the NDFSM is in state 1. This automaton accepts the input *a b a b a c* in the following way. The input *a* causes the automaton to go to state 2, the *b* to state 1, the *a* to state 2, the b to state 1, the *a* to state 3 and the *c* to state 4. Write a program to simulate this NDFSM.

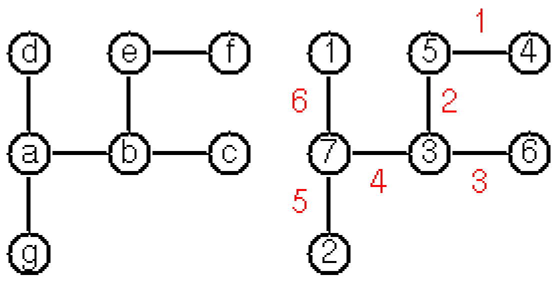
   The next problems are all taken from Werner Hett's web site of *Ninety-Nine Prolog Problems*.

3. **The eight queens problem.** This is a classical problem in computer science. The objective is to place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal. Write a program that computes all the solutions for this problem. Your program will systematically place queens and force a backtrack when a queen cannot be added to the board without attacking another queen already on it.

4. **The knight's tour.** Another famous problem is this one: how can a knight jump on a chessboard in such a way that it visits every square exactly once?

5. *__Graceful labeling problem__. This problem is taken from Werner Hett who records.

   "Several years ago I met a mathematician who was intrigued by a problem for which he didn't know a solution. His name was Von Koch, and I don't know whether the problem has been solved since.

   Anyway the puzzle goes like this: Given a tree with N nodes (and hence N-1 edges). Find a way to enumerate the nodes from 1 to N and, accordingly, the edges from 1 to N-1 in such a way, that for each edge K the difference of its node numbers (labels) equals to K. The conjecture is that this is always possible. [The diagram above shows a solution for one tree].

   For small trees the problem is easy to solve by hand. However, for larger trees, and 14 is already very large, it is difficult to find a solution. And remember, we don't know for sure whether there is always a solution!"

   Write a program that calculates a labelling scheme for a given tree.

6. *Again from Werner Hett.

   "Given a list of integer numbers, find a correct way of inserting arithmetic signs (operators) such that the result is a correct equation. Example: With the list of numbers [2,3,5,7,11] we can form the equations 2-3+5+7 = 11 or 2 = (3*5+7)/11 (and ten others!)."

   Write a program that finds the list of all possible solutions to this problem.

7. Write a function **path** to find an acyclic path P from node A to node B in the graph G. The function should return all paths via backtracking.

8. Write a function **cycles** to find a closed path (cycle) P starting at a given node A in the graph G. The function should return all cycles from A via backtracking.

9. Two graphs G_1 and G_2 are *isomorphic* if there is a 1-1 function from the set of nodes of G_1 onto the set of nodes of G_2 such that for any nodes X,Y of G_1, X

and Y are adjacent in G_1 if and only if f(X) and f(Y) are adjacent in G_2. Write a function that determines whether two graphs are isomorphic.

Further Reading

The use of backtracking in programming dates at least as far back as **Hewitt's** (1969) PLANNER and became widely used when Prolog (**Kowalski** (1979)) made its appearance. A more restrictive version of the backtracking used in Qi appeared in the MetaLisp programming language (**Lajos** (1990)) and in **Wright** (1991). **Liu** and **Staples** (1993) introduce a backtracking extension into C. **Haynes** (1987) and more recently **Sitaram** (1993) experimented with backtracking in Scheme, a functional language that is a dialect of Lisp. **Charniak** and **McDermott** (1985), discuss improvements to our implementation of basic backward chaining which involve applying rules that generate the fewest subgoals. Chapter 6 of that book contains a very readable exposition of the issues involved in using backward chaining in implementing reasoning systems. Chronological backtracking is not the only form of backtracking; **dependency-directed backtracking** aims to intelligently analyse the reasons for why a search ends in a failure node and tries to undo those decisions that are directly responsible for the failure. This area is also studied within **constraint programming**, which studies the **constraint satisfaction problem** of assigning values to each element of a set of variables in such a way that a set of constraints is satisfied. Problems which fall under this category range from DNA sequencing to scheduling.

The logic programming language **Prolog** is an extension of Proplog incorporating unification and is discussed in **Sterling** and **Shapiro** (1994) and **Bratko** (2002); **Lloyd** (1990) provides the theory to Prolog while **Hogger** (1990) is a rather gentler introduction. **Stickel** (1984, 1986) describes an extension to Prolog that is complete for first-order logic.

Web Sites

Roman Bartak (http://kti.ms.mff.cuni.cz/~bartak/constraints) maintains a site devoted to constraint programming. Free implementations of Prolog are easily obtained over the web. **Sicstus Prolog** (http://www.sics.se/sicstus) is one of the fastest versions of Prolog currently available, but is not freely available. **SWI Prolog** (http://www.swi.prolog.org) is another widely used implementation and is free. GNU Prolog (http://pauillac.inria.fr/~diaz/gnu-prolog) is a free Prolog implemented by **Daniel Diaz** that offers extra facilities for constraint satisfaction handling. **Visual Prolog** (http://www.visual-prolog.com) is available from the **Prolog Development Centre** and includes facilities for building graphical interfaces. A free download version for non-commercial purposes is available from the Web site. **Qi-Prolog** is a Prolog bundled with Qi that allows embedded function calls within terms. It is documented in Lambda Associates (www.lambdassociates.org) Werner Hett maintains a web site (http://www.hta-bi.bfh.ch/~hew/informatik3/prolog/p-99) of 99 Prolog problems, from which many of the problems in this exercise are taken.

8 Metaprograms

A metaprogram is a program that writes programs. Most students of computer science are familiar with metaprograms in the shape of compilers - programs that accept programs written in a high-level language and output object-code programs in the form of lower-level instructions – generally assembler or machine code. Advanced students of functional programming will eventually graduate to the point where they will want to do metaprogramming. At that point they may find that their chosen functional language has little or no facility to support metaprogramming. However Qi provides full support for metaprogramming and this chapter is designed to introduce the reader to metaprogramming in Qi.

8.1 Building a Parser in Qi

Languages, whether natural like English or French, or artificial, like Pascal or Qi, possess a syntax whereby certain series of symbols are grammatically acceptable and others are not. To take English as an example, “John kicks the ball” is an acceptable English sentence but “the kicks ball John” is not. One basic facility that every competent programmer needs is that of being able to design, upon request, programs that recognise whether a certain series of symbols is grammatically acceptable according to the syntax rules of a given language. Such programs are called *recognisors* for the sentences of the language in question.

A recognisor for a language may only return two values; indicating either that its input is grammatical or that it is not. Rather more useful is a *parser* which returns either the verdict that the input is not grammatical or else, if the input is grammatical, returns a description of the *grammatical structure* of the input.[34]

[34] Such programs can be a big challenge to write because of the size of the grammars for English. For instance, realistic grammars will contain several thousand rules, which pose a severe computational challenge for any parser. Happily most programmers do not set

For example, a linguist would say that "John kicks the ball" is a grammatical sentence composed of two parts - a *noun phrase* comprising the *name* "John" and a *verb phrase* "kicks the ball". The verb phrase is composed of two parts, a *transitive verb* "kicks" and another *noun phrase* "the ball". Finally the concluding noun phrase is built out of a *determiner* "the" and a *noun* "ball". This analysis is sometimes represented as a tree structure in which *sent* stands for *sentence*, *np* stands for *noun phrase*, *vp* stands for *verb phrase*, *vtrans* stands for *transitive verb*, *det* stands for *determiner* and *n* stands for *noun.* Such a tree is called a **parse tree** (figure 8.1).

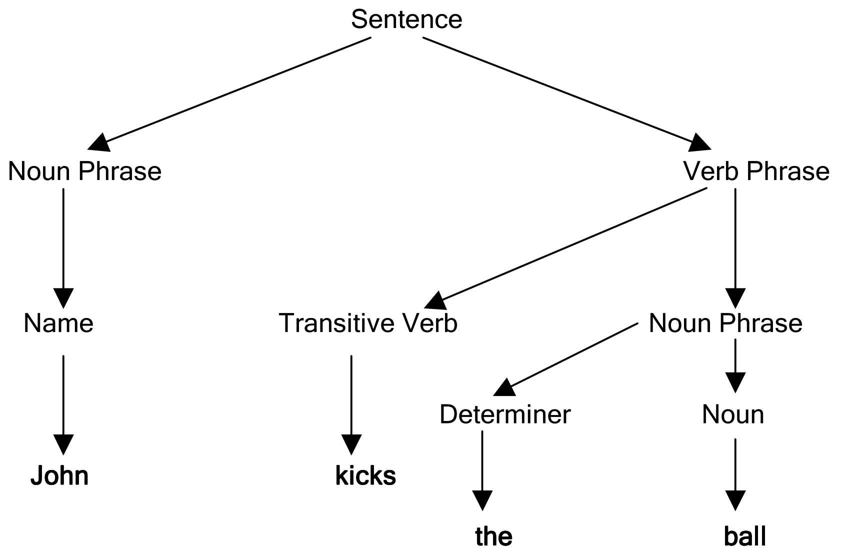


Figure 8.1 A Parse Tree Representing the Grammatical Structure of "John kicks the ball"

The choice of representation is conventional; the same information can be given as a list.

[[sent --> np vp] [np --> name] [name --> "John"] [vp --> vtrans np] [vtrans -->"kicks"] [np -->det n] [det -->"the"] [n-->"ball"]]

The grammar rules are the rules that attempt to define what counts as grammatical in the language studied. The shape and form of these

themselves such an ambitious task and the techniques taught in this chapter will suffice to solve simple parsing problems.

grammar rules vary according to the system used, but by far the commonest form is the **context-free grammar** (also called a **type 2 grammar**). A context-free grammar consists of a series of context-free grammar rules. Each such rule consists of a left-hand side and a right-hand side separated by some agreed symbol (generally →, but also sometimes ::=). The left-hand side has but one symbol and the right-hand side one or more symbols. For example, here is a simple grammar which parses our specimen sentence.

1. sent → np vp
2. np → name
3. np → det n
4. name → "John"
5. name → "Bill"
6. det → "the"
7. det → "a"
8. det → "that"
9. det → "this"
10. n → "ball"
11. n → "girl"
12. vp → vtrans np
13. vp → vintrans
14. vtrans → "kicks"
15. vtrans → "likes"
16. vintrans → "jumps"

Figure 8.2 A Grammar for a Simple Fragment of English

One symbol, called **the distinguished symbol**, occurs to the left of the arrow but never to the right; in our example this is *sent*. A series of expressions $e_1,...e_n$ are **expansions** of an expression e if there is a rule e → $e_1,...e_n$ in the grammar. The expressions that occur in the grammar but never on the left-hand side of an → are called **terminals**. A **lexical category** is a non-terminal whose expansions are always terminals.

In order to prove a sentence S is grammatical according to the grammar rules we can follow a non-deterministic procedure I call the *basic top down parsing procedure*. For our purposes a sentence will be a list (so "John kicks the ball" will be ["John" "kicks" "the" "ball"]).

Basic Top Down Parsing Procedure.

```
Let $i$ = the list containing only the distinguished symbol, let S be the
        sentence to be parsed.
Repeat.
        If $i$ consists of only terminals and $i$ = S, then halt. S is grammatical.
        Print the proof.[35]
        If $i$ consists of only terminals and $i \neq$ S, then halt. The proof has
        failed. Otherwise choose a non-terminal $i_n$ in $i$. Choose an
        expansion of $i_n$ and set the new value of $i$ to the result of replacing
        $i_n$ in the old value of $i$ by the chosen expansion.
```

To illustrate, we wish to prove ["John" "kicks" "the" "ball"] is grammatical according to *G*. The symbol $\rightarrow n$ indicates that a symbol may be expanded according to the *n*th rule in *G*. The following chain of expansions constitutes a proof that ["John" "kicks" "the" "ball"] is grammatical.

[sent] $\rightarrow 1$ [np vp] $\rightarrow 2$ [name vp] $\rightarrow 4$ ["John" vp] $\rightarrow 12$ ["John" vtrans np] $\rightarrow 14$ ["John" "kicks" np] $\rightarrow 3$ ["John" "kicks" det n] $\rightarrow 6$ ["John" "kicks" "the" n] $\rightarrow 10$ ["John" "kicks" "the" "ball"]

Though the basic top down parsing procedure is perfectly sound it requires modification for three reasons.

1. It cannot show a sentence is ungrammatical. A negative result from this procedure could be due to a faulty choice of expansions earlier in the proof. For instance if [np vp] $\rightarrow 3$ [det n vp] had been chosen instead of [np vp] $\rightarrow 2$ [name vp], then the proof attempt would have failed.
2. The basic top down parsing procedure provides no clue as to which expansions to use.
3. The basic top down parsing procedure provides no clue as to which non-terminals to expand.

The solution to the second problem is also a solution to the first. We mimic the non-determinism in the basic top down parsing procedure by arranging to backtrack if the procedure has led to a dead end. Since by this method all available expansions will be tried, if a negative result is still returned

[35] Most usually, this proof is presented as a parse tree, but there is no necessity to do so. The parse tree is only significant in rendering graphically the history of the expansions that were used to prove the sentence grammatical. The difference between a parser and a recognisor is not that the parser produces a parse tree and the recognisor does not, but that the parser produces a proof of its verdict whereas the recognisor does not.

then this must be because the input sentence is ungrammatical and so our improved procedure can return a message to this effect. The solution to the final problem is to adopt a convention as to which non-terminal will be expanded first and the most convenient choice is to work on the leftmost or leading non-terminal in *i*.

In this revised form, the top down parsing procedure is still inefficient because it has to convert all non-terminals to terminals before comparing the result with the input sentence to see if they are identical. But it is often possible to spot dead-ends before this stage. For instance, if the chain [np vp] $\rightarrow$3 [det n vp] $\rightarrow$ 6 ["the" n vp] had been followed, then the failure of "the" to match the input "John" would signify straight away that further expansion of ["the" n vp] was a waste of time.

A much improved performance arises when we arrange that whenever the leading expression i_1 of i is a terminal, that further expansion proceeds only if i_1 = S_1 (where S_1 is the first element of S). Computationally it is then convenient to remove i_1 from i and S_1 from S so that the parse finally succeeds when every word in S has been accounted for; i.e. S has been reduced to an empty list. This adaptation of the basic top down parsing procedure is called *recursive descent parsing*.

8.2 Recursive Descent Parsing in Qi

Building a recursive descent parser for *G* in Qi is not too difficult. Each non-terminal *N* in *G* is represented by a function in Qi. The nature of this representation varies according to whether *N* is a lexical category or not. Let us deal with the case where *N* is a lexical category using an example. These are the rules dealing with the lexical category n (Noun).

$$n \rightarrow \text{"ball"}$$
$$n \rightarrow \text{"girl"}$$

An initial attempt to construct a function for n would simply receive the input list and check to see if the head of the list was "girl" or "ball". If so, following recursive descent parsing, the head of the list is removed. If not, then **#\Escape** is returned showing the function has failed to find a noun.

```
(define n
  ["ball" | Words]  -> Words
  ["girl" | Words] -> Words
   _ -> #\Escape)
```

This is almost adequate except that if the head of the list is a noun, then the fact that the relevant expansion has been used must be recorded so that it can be returned at the end of the parse. A solution is to make the input a pair composed of (a) the sentence being parsed (a list of strings) and (b) a list of the rules used to parse it so far. This second list will be returned at the end of the parse as a proof that the sentence is grammatical. The revised version now reads.

```
 (define n
  [["girl" | Words] Proof]  -> [Words [[n - -> "girl"] | Proof]]
  [["ball" | Words] Proof]  -> [Words [[n - -> "ball"] | Proof]]
  _ -> #\Escape)
```

Now consider the construction of a function for a non-lexical category; for example vp. vp has two expansions vp → vtrans np and vp → vintrans. If the first expansion is used then (a) the front of the sentence will be scanned for a transitive verb and (b) if there is one, the remainder of the sentence will be scanned for a noun phrase. However either of these procedures might fail in which case the failure object will be found. In such an event we will want to try the second expansion. So the outline form of the function **vp** will be

```
(define vp
  Sentence <- (<look-for-a-noun-phrase>
          (<look-for-a-transitive-verb> Sentence)))
  Sentence -> (<look-for-a-intransitive-verb> Sentence))
```

The gaps ***<look-for-a-transitive-verb>, <look-for-a-noun-phrase>*** and ***<look-for-a-intransitive-verb>*** will be taken up by the functions that encode vtrans, np and vintrans respectively so the function will look like this.

```
(define vp
  Sentence <- (np (vtrans Sentence))
  Sentence <- (vintrans Sentence)
  _ -> #\Escape)
```

But since the input is a pair, we need again to record the expansion used. So the final form is:-

```
(define vp
  [Sentence Proof] <- (np (vtrans [Sentence [[vp - -> vtrans np] | Proof]]))
  [Sentence Proof] -> (vintrans [Sentence [[vp - -> vintrans] | Proof]])
  _ -> #\Escape)
```

Dealing with all the other expansions in the same way generates the program in figure 8.3.

```
 (define sent
   [Input Proof] <- (vp (np [Input [[sent - -> np vp] | Proof]]))
   _ -> #\Escape)

(define np
   [Input Proof] <- (n (det [Input [[np - -> det n] | Proof]]))
   [Input Proof] <- (name [Input [[np - -> name] | Proof]])
   _ -> #\Escape)

(define name
  [["John" | Input] Proof] -> [Input [[name - -> "John"] | Proof]]
  [["Bill" | Input] Proof] -> [Input [[name - -> "Bill"] | Proof]]
   _ -> #\Escape)

 (define det
  [["the" | Input] Proof] -> [Input [[det - -> "the"] | Proof]]
  [["a" | Input] Proof] -> [Input [[det - -> "a"] | Proof]]
  [["that" | Input] Proof] -> [Input [[det - -> "that"] | Proof]]
  [["this" | Input] Proof] -> [Input [[det - -> "this"] | Proof]]
  _ -> #\Escape)

(define n
  [["ball" | Input] Proof] -> [Input [[n - -> "boy"] | Proof]]
  [["girl" | Input] Proof] -> [Input [[n - -> "girl"] | Proof]]
   _ -> #\Escape)

(define vp
   [Input Proof] <- (np (vtrans [Input [[vp - -> vtrans np] | Proof]]))
   [Input Proof] <- (vp [Input [[vp - -> vintrans] | Proof]])
   _ -> #\Escape)

(define vtrans
  [["kicks" | Input] Proof] -> [Input [[vtrans - -> "kicks"] | Proof]]
  [["likes" | Input] Proof] -> [Input [[vtrans - -> "likes"] | Proof]]
   _ -> #\Escape)
```

Qi Programs/Chap8/parser.qi

```
(define vintrans
  [["jumps" | Input] Proof]
-> [Input [[vintrans - -> "jumps"] | Proof]]
  _ -> #\Escape)
```

Figure 8.3 A Parser for our Simple Fragment of English

All that is needed to complete the program is the driver function that builds the input and applies the function representing the distinguished function and tests to see if the parse is successful (figure 8.4).

```
(define parse
   Sentence -> (let Parse (sent [Sentence []])
                    (if (parsed? Parse)
                        (output_parse Parse)
                        ungrammatical)))

(define parsed?
   [[] _] -> true
    _ -> false)

(define output_parse
   [_ Parse_Rules] -> (reverse Parse_Rules))
```

Figure 8.4 The top level of the parser

The program can now be used to parse sentences according to the grammar rules (figure 8.5).

```
(22-) (parse ["the" "boy" "likes" "the" "girl"])
[[sent - -> np vp] [np - -> det n] [det - -> "the"] [n - -> "boy"] [vp - -> vtrans
np] [vtrans - -> "likes"] [np - -> det n] [det - -> "the"] [n - -> "girl"]]

(23-) (parse ["the" "cat" "likes" "the" "girl"])
ungrammatical
```

Figure 8.5 The parser in action

Careful examination of the grammar and its associated parser shows there is a computable procedure for extracting the parser given the grammar as input. So it is more sensible to program the computer to write the parser than to code it by hand. Such a program is a **parser-generator** - a metaprogram that receives a grammar as input and generates a parsing program as an output. In the next section we shall see how to write such a program in Qi.

8.3 The **eval** Function

The 1-place function **eval** exists in Qi to facilitate the production of metaprograms. It receives an input *E* and produces the output that would result from evaluating *E**, where *E** results from *E* by replacing all square brackets in *E* by round ones (figure 8.6).

```
(0-) (eval 23)
23

(1-) (eval [* 3 4])
12

(2-) (eval [3 4])
EVAL: 3 is not a function name

(3-) (eval [define f
              [cons X [cons Y Z]] -> Z])
f

(4-) (f [1 2 3])
[3]
```

Figure 8.6 Some Uses of **eval**

Figure 8.6 shows that **eval** can be used to generate function definitions, which are represented in cons form using square brackets and then evaluated by the **eval** function. This means that we can generate list structures that are associated 1-1 with function definitions, and using **eval**, generate and compile the associated definition into Qi. Thus the steps to constructing an Qi metaprogram are:-

1. Receive the input(s).
2. Generate a series of list structures which are associated with the output program.
3. Apply **eval** to these list structures to compile the output program into Qi.

To design a metaprogram[🗀] for generating parsers, we require a metaprogram that takes a list of grammar rules as input, and returns a program that will perform parsing according to those rules. Effectively this requires associating function definitions 1-1 with groups of grammar rules that are all expansions of the same non-terminal. Lets call this non-terminal the **characteristic non-terminal** of this group of rules. Thus we will collect all rules relating to (say) **vp** and from these rules we will generate an appropriate function and compile it into Qi using **eval**. To begin; the top level of our parser-generator will receive a grammar in the following form.

```
[sent - -> np vp, np - -> name, np  - -> det n, name - -> "John",
 name - -> "Bill", name - -> "Tom", det  - -> "the",  det  - -> "a", det  - ->
"that", det  - -> "this",  n - -> "girl", n - -> "ball", vp - -> vtrans np,  vp - ->
vintrans, vtrans - -> "kicks", vtrans - -> "likes",  vintrans - -> "jumps",
vintrans - -> "flies"]
```

The first task is to parenthesise each of these rules and then place them into groups corresponding to each non-terminal (figure 8.7).

```
(define generate_parser
  Grammar -> (map compile_rules
                  (group_rules (parenthesise_rules Grammar))))

(define parenthesise_rules
  [S - -> ¦ Rest] -> (parenthesise_rules1 [S -->] Rest))

(define parenthesise_rules1
  Rule [] -> [Rule]
  Rule [S - -> ¦ Rest]
         -> [Rule ¦ (parenthesise_rules1 [S -->] Rest)]
  Rule [X ¦ Y] -> (parenthesise_rules1 (append Rule [X]) Y))

(define group_rules
  Rules -> (group_rules1 Rules []))

(define group_rules1
  [] Groups -> Groups
  [Rule ¦ Rules] Groups
   -> (group_rules1 Rules (place_in_group Rule Groups)))
```

[🗀] Qi Programs/Chap8/metaprog.qi.

```
(define place_in_group
  Rule [] -> [[Rule]]
  Rule [Group ¦ Groups]
            -> [(append Group [Rule]) ¦ Groups]
                        where (belongs-in? Rule Group)
  Rule [Group ¦ Groups] -> [Group ¦ (place_in_group Rule Groups)])

(define belongs-in?
  [S ¦ _] [[S ¦ _] ¦ _] -> true
  _ _ -> false)
```

Figure 8.7 Placing the grammar in a canonical form

The **compile_rules** function takes each group of rules and determines if the characteristic non-terminal is a lexical category or not.   Depending on the result, the appropriate code generation procedure is invoked (figure 8.8).

```
(define compile_rules
  Rules -> (if (lex? Rules)
                (generate_code_for_lex Rules)
                (generate_code_for_nonlex Rules)))

(define lex?
  [[S - -> Terminal] ¦ _] -> (string? Terminal)
  _ -> false)

(define generate_code_for_nonlex
  Rules -> (eval
    (append
          [define (get_characteristic_non_terminal Rules)
                  |  (mapapp gcfn_help Rules)]
      [X -> #\Escape])))

(define mapapp
    _ [] -> []
    F [X | Y] -> (append (F X) (mapapp F Y)))

(define get_characteristic_non_terminal
  [[CNT | _] | _] -> CNT)
```

```
(define gcfn_help
  Rule  -> [Parameter <- (apply_expansion Rule [list [head Parameter]
                   [cons [list | Rule] [head [tail Parameter]]]])])

(define apply_expansion
   [CNT -> | Expansion] Parameter
     -> (ae_help Expansion Parameter))

(define ae_help
   [] Code -> Code
   [NT | Expansion] Code -> (ae_help Expansion [NT Code]))

(define generate_code_for_lex
  Rules -> (eval (append
  [define (get_characteristic_non_terminal Rules)
           #\Escape -> #\Escape |  (mapapp gcfl_help Rules)]
              [X -> #\Escape])))

(define gcfl_help
   [CNT -> Terminal]
   -> [[list [cons Terminal P] Parse]
        -> [list P [cons [list CNT -> Terminal] Parse]]])
```

Figure 8.8 Metaprogram component of the parser generator

Thus applied to the rules **[[np - -> det n] [np - -> name]]** our metaprogram generates the following code.[36]

```
(define np
  #\Escape -> #\Escape
  Parameter <- (n (det (list (head Parameter)
              (cons (list np - -> det n)
                                (head (tail Parameter))))))
  Parameter <- (name (list (head Parameter)
                 (cons (list np - -> name) (head (tail Parameter)))
   X -> #\Escape)
```

The only significant lack is that the **parse** function is not generated. Instead of a **parse** function dedicated to a specific grammar, we instead create a higher-order version of the same function that receives the distinguished symbol of any grammar as a input (figure 8.9).

[36] The code contains an optimisation for avoiding unnecessary function calls; if the failure object is received then the function immediately returns failure.

```
(define parse
  \D is the distinguished symbol of the grammar\
  D Sentence -> (let Parse (D [Sentence []])
                (if (parsed? Parse)
                    (output_parse Parse)
                    ungrammatical)))

(define parsed?
  [[] _] -> true
  _ -> false)

(define output_parse
  [_ Output] -> (reverse Output))
```

Figure 8.9 The top level of our parser generator

It is now simple to take a grammar and to generate a parser for it. Our new parse function is designed to work with any suitable grammar. To invoke our parser for the grammar above on the sentence **["the" "girl" "likes" "the" "ball"]**, we enter **(parse sent ["the" "girl" "likes" "the" "ball"])** (figure 8.10).

```
(67-) (generate_parser
       [sent - -> np vp  np - -> name  np - -> det n
            name - -> "John"  name - -> "Bill"  name - -> "Tom"
      det - -> "the"  det  - -> "a"  det - -> "that"
      det - -> "this"  n - -> "girl"  n - -> "ball"
      vp - -> vtrans np  vp - -> vintrans  vtrans - -> "kicks"
      vtrans - -> "likes"  vintrans - -> "jumps"
      vintrans - -> "flies"]
[sent np name det n vp vtrans vintrans]

(68-) (parse sent ["the" "girl" "likes" "the" "ball"])
[[sent - -> np vp] [np - -> det n] [det - -> "the"]
 [n - -> "girl"]  [vp - -> vtrans np] [vtrans - -> "likes"]
 [np - -> det n] [det - -> "the"]  [n - -> "ball"]]

(69-) (parse sent ["the" "girl" "likes" "the" "cat"])
ungrammatical
```

Figure 8.10 Generating a Parser for a Context Free Grammar

Exercise 8

1. *Given the grammar rules

```
sent - -> sent connective sent
connective - -> "and"
connective - -> "or"
```

   our parsing procedure goes into an infinite loop when given an input. How would you cure this fault? Alter the metaprogram so that it generates parsers that are free of this deficiency.

2. **Our metaprogram generates only parsers. A compiler would not only parse the input, but translate it into some target language. Upgrade our parser-generator so that it becomes a compiler-compiler, generating compilers instead of parsers. You may want to do some background reading on this. Tarver (1990) is one place to begin; this model uses lazy evaluation. Lajos (1990) describes a developed version of the model that uses eager evaluation.

3. *Read Bundy and Duffy on the Knuth-Bendix procedure and implement the procedure in Qi, so that the rewrite rules generated by the Knuth-Bendix procedure are compiled directly into Qi function definitions.

4. *Melvin Micro notices that the rules for the Proplog inference engine in chapter 7 can be coded into efficient Qi; for instance the assumptions **p <= [q & s], p <= r, r** can be expressed in three functions.

```
(define p
  <- (fail-if not (and (q) (s)))
  <- (fail-if not (r))
  -> false)

(define q
 -> false)

(define r
 -> true)

(define s
 -> false)
```

   Querying the computer if **[p & r]** is provable can be done with the call **(and (p) (r)).** Write a metaprogram that takes a set of Proplog assumptions as input and generates such a program.

4. Amend your program in 3., so that at the end of a successful proof, the proof itself is printed off.

5. **Read Wikstrom or Paulson on Standard ML and design a **cross-compiler** from SML into core Qi. Your program should read in an ML program from a file as a list of characters (see **read-file-as-charlist** in appendix A) and either output the corresponding Qi program into a different file (see **write-to-file** in appendix A) or evaluate the output on the fly so as to compile the Qi code into the current image. You can ignore ML type definitions, but you need to decide what to do with ML constructor functions. The easiest course is to represent them as simple list constructions. Ambitious readers can try the type secure version of this problem posed at the end of chapter 11.

6. **Follow up the references on *partial evaluation* in the further reading and build a program **mix** that partially evaluates Qi functions. Your program should be able to take a function like

```
(define expt
   _ 0 -> 1
  X Y -> (* X (expt X (- Y 1))))
```

and partially evaluate it with the second (2) parameter fixed at 3, so that it outputs the function

```
(define expt2.3
  X ->  (* X (* X X))))
```

If you succeed in this, then you may want to try more ambitious examples, such as partially evaluating the unification function described in chapter 13.

Further Reading

Metaprogramming was very much a feature of the first functional programming language Lisp, but introductory Lisp texts tend to skip over metaprogramming. Other functional languages like SML avoid the **eval** function, sacrificing metaprogramming ability to type security. Consequently there is very little reading on metaprogramming in functional languages; perhaps the biggest area of related research has been **partial evaluation**, which uses metaprogramming techniques to optimise the performance of programs. The principle references for this are **Futamura** (1983), **Jones, Sestoft** and **Sondergaard** (1985) and **Sestoft** (1986). **Diller** (1988) provides an overview in chapter 11. **Tarver** (1990) gives the Lisp source code for a compiler-compiler based on the technology in this chapter.

The Qi **eval** is derived from the Common Lisp EVAL that coexists with Qi; see any introductory Lisp text for the use of EVAL. **McCarthy** (1978) records that the invention of the EVAL function gave rise to the first interpreter for Lisp.

The construction of parsers and compilers has an honourable place in the history of computing, right back to the construction of the first FORTRAN compiler. **Grune** and **Jacobs** (1991) offers an excellent summary of parsing techniques with a full bibliography. **Aho** and **Ullman** (1972) is the classic text for compiler design.

Web Sites

Programmar (http://www.programmar.com/) a visual environment for building parsers that are platform-independent, programming language-independent and reusable.

Qi-YACC is a compiler-compiler built into Qi that uses a more sophisticated encoding than the one used in this chapter. It is bundled inside Qi with documentation from Lambda Associates (www.lambdassociates.org).

9 Writing Good Programs

The previous chapter covered the last elements of what may be called the *core* Qi language. Core Qi is a very powerful programming language and it is possible to write a great number of good and useful programs in it. Many textbooks say little about how to write good programs, preferring to teach by example, rather than through precept. But because we have now covered enough to know how powerful and elegant functional programming can be, it is worth pausing to try and distil what we have learnt about how to write functional programs.

First it must be said that functional programming is a *craft.* Like any craft, it takes practice and practice is the best teacher. The more exercises you do, the better you become, and hopefully you should outgrow the exercises in this book, and go on to pursue ideas of your own. In course of working on programs, you will develop an intuitive appreciation of how to tackle problems and, with luck, you will become a craftsman. Here are 9 precepts, designed to help you improve your technique, which tell you what to aim at and what to avoid.

1. *Employ top down programming methodology to tackle difficult problems.*

Most significant programming tasks are too difficult for you to know exactly how they are to be solved. Functional programmers respond by a top down methodology, whereby the top level functions of the program are coded first, then the functions immediately below the top level, and so on down to the system functions. At each step they postpone the consideration of exactly how the given functions they cite in a definition are themselves implemented until they come to define them. Use the same technique and don't be afraid of being banal. For example, suppose you are given the task of getting a computer to make a move in chess and you have decided to employ some method whereby the computer generates a list of likely moves and analyses them. However you are still a little hazy about how the best moves are selected. No matter; you can make an appropriate beginning by writing.

```
(define play-move
  Position -> (select-best-move (generate-likely-moves Position)))
```

2. *Make a program clear before you try to make it efficient.*

Inexperienced programmers are sometimes tempted into trying to optimise before they have fully understood what they are doing. Even if the program then works, the programmer is not quite sure what is going on, and debugging the program becomes very difficult. Aim to be clear initially and forget about being clever until later. Often clear programs tend to be efficient anyway.

3. Get used to throwing away code.

Declarative programmers talk about **throwaway designs** - programs that are written in order to help the understanding, but which are thrown away and replaced by something better once that understanding has been achieved. Don't be afraid to throw things away instead of patching them up.

4. Use significant variable and function names.

If a variable is supposed to stand for a list of towns, then use **Towns** rather than **X**. Similarly if a function is supposed to sort towns by population call it **sort-by-population** rather than **sbp**. You may think you know what **sbp** means now, but returning after 3 weeks holiday it may mean nothing to you. Suitable variable and function names remove the need to attach copious comments.

5. Look for a common pattern in different processes, and try to capture it with higher-order functions.

We covered this in the chapter on higher-order programming. Acquire the habit of looking for common patterns in diverse processes and of using higher-order functions to capture these patterns. Apart from being an excellent mental training, forming this habit will lead you to simpler and often faster programs.

6. If a procedure is used more than once, then it should be defined within its own function.

If there is a procedure, which is called again and again within your program, hive it off into a separate function and give it a name. Not only does this lead to a shorter program, since you can just invoke the procedure through the name of the function instead of typing in the whole procedure, but also the program becomes easier to maintain. If the procedure has to be changed, then only its definition need be altered, rather than everywhere it is called.

7. Avoid writing large function definitions.

Students who have attained some experience in functional programming, but are not yet expert, often write large unwieldy functions into which a large amount of activity is squeezed. Keep functions small. It is better to have a lot of short simple functions than a few large complex ones.

8. Try to avoid using assignments to hold information unless the information is of a permanent non-changing kind.

A source of "spaghetti code", where the thread of control is lost, is due to assignments over-playing their role. Ideally global values should be used to represent information which remains almost unchanged throughout the execution of program. In this role, they reduce the need to create extra inputs to functions, since a function can consult the global value if it needs to access this information. However, if a global value changes throughout a program, then the program is in danger of being a "spaghetti program". It is very difficult to correct mistakes in the program since the source of the error involving the global value cannot be easily localised.

9 . If you find yourself feeling discouraged, tired or annoyed then don't continue to hack away at your program. Give it up and do something else and return to it later.

Writers of computer science texts often neglect the truism that programming is a human activity and that human beings get upset. Being upset and writing good code just do not go together. Anything you do while you tired or unsettled is liable at best to be worthless and at worst to be counterproductive. Most experienced programmers can recall making mistakes when feeling like this (a favourite is deleting some vital file). Save your work, log off, and walk away.

Part II: Working with Types

10 Types

Socrates: *Well then, since we have agreed that kinds of things can similarly mix with each other, is there not some sort of knowledge that one needs to take one's progress through the realm of statements, if one is going to point correctly which kinds will harmonise with which, and which are mutually discordant?*

Theaetetus: *Of course one needs knowledge, and possibly the greatest there is.*

Plato *The Theaetetus*

10.1 Types and Type Security

The observations that Plato made over 2,300 years ago also hold true in computing. Programmers manipulate objects of many types, and for each type there is a set of specific legitimate operations that can be performed on the objects which belong to or **inhabit** that type. It is sensible to multiply two numbers or to take the head of a list. It is not sensible to try to take the head of a number or multiply two lists. The ability to recognise and avoid discordant **type errors** of this kind is one important form of knowledge that the programmer needs to possess if his programs are to work.

Since type errors must be avoided in order to have a properly working program, computer languages provide varying degrees of support to ensure that written in them are free from type errors, or **type-secure**. In ANSI C, this support is virtually non-existent. Type errors are found when programs crash and the computer core dumps. Other languages, like Lisp, try to provide a message to point the programmer to the source of the error. Both these languages support **dynamic type checking;** only when programs are run, are the type errors in them detected.

In the '60s, **static type checking** was invented. In statically typed languages, type errors are detected in programs without having to run them. Statically typed languages are almost invariably **strongly typed languages** - i.e. every object of computation and every function must belong to a type, and the set of recognised types is rigorously laid down. This is a condition of being able to analyse programs to determine their type-security. The **type discipline** of the language is given by the rules, which determine how objects and types are associated. When functional programmers talk of typed programming languages, they mean languages that are statically and strongly typed.

10.2 Modifying the Read-Evaluate-Print Loop

In statically typed functional programming language, the Read-Evaluate-Print loop is modified to become a Read-Check-Evaluate-Print loop. The extra steps in the loop are italicised.

The Read-Check-Evaluate-Print Loop

Do until the user exits.

1. Read in the user's input.
2. *Type check it.*
3. *If there is a type error, output an error message and go to 1.*
4. If there is no type error, evaluate it to give the normal form.
5. Print the normal form together with its type.

If **(tc +)** is typed to the Qi top level, the Qi switches over to this loop.[37] After evaluation, the normal form and its type are printed separated by a colon, so **6 : number** means **6** *belongs to the type number*. **6 : number** is an example of a **typing** (i.e. a functional expression together with a type). Typings can be entered to the evaluator. If a typing E : A is typed to the top level, the Qi interpreter performs two operations.

1. If E : A is not provable, then a type error is signalled.
2. If E : A is provable, then E is evaluated. If E has a normal form E*, then E*: A is printed.

A sample session is shown in figure 10.1.

[37] **(tc -)** switches back to dynamic type checking.

```
(0-) (tc +)
true

(1+) (* 7 8)
56 : number

(2+) (* 7 a)
type error

(3+) (* 6 6) : string
type error
```

Figure 10.1 Enabling type checking in Qi

Qi is equipped with nearly 100 system functions, many of which have types. A complete type system for Qi would include a rule for every such function, which would be prohibitively large to state on paper and very inefficient if implemented directly onto the computer. Instead we suppose that there is a function τ which maps every function symbol to its type in Qi by reference to a lookup table for types or a **type environment**. Using the type environment, Qi can deduce the types of the expressions entered to its top level.

10.3 Base Types and Operators

Every typed functional language has a set of predefined types called **base types**, and operators called **type operators** for creating new types out of old ones. Qi has 5 base types.

1. **symbol**; e.g. a21, computer_studies, myname.
2. **string**; e.g. "my name is".
3. **boolean**; true, false.
4. **number**; e.g. ...,-3,-2,-1,0,1,2,3,... , 3/4, 6/7, 9/2, ... 3.1462.
5. **character**; e.g. ...,#\r, #\9,....

A **type operator** is a means of building new types out of old. Qi has three principle type operators, **list**, $\rightarrow$ and *****.

10.4 The list Type Operator

We explain the list operator by giving the conditions under which something inhabits a list type.

Rule for the Type Operator list

Where A is any type, list l : (list A) just when for each x in l, x : A.

[3 4] inhabits the type **(list integer)** which is the type of all lists whose elements are all integers; a list of strings inhabits the type **(list string);** a list of lists of strings inhabits the type **(list (list string))**. What if we submit a list of symbols and integers to Qi?

```
(7+) [76 trombones]
type error
```

There is no type to which each element of the list belongs, the list does not inhabit a list type and consequently is treated as badly typed by Qi. Strongly typed functional languages do not naturally tolerate lists containing objects of different types. However we can retain strong typing and handle mixed sequences too through the use of **product types**, as we shall see later.

10.5 The Function Space Type Operator →

The next rule gives the conditions under which a function inhabits a type constructed by the **function space** operator →.

Rule for the Type Operator →

For any types A and B, *f*: (A → B) just when for all *x*, if *x* : A then (*f x*) : B.

Here is an example entered to Qi. The symbol - -> is used for the type operator →.

```
(1+) (define double
     {number - -> number}
     X -> (* X 2))
double : (number - -> number)
```

Figure 10.2 Typechecking a function in Qi

double has the type **number** → **number**, because if *x* is a number then (**double** *x*) is a number. Qi will not verify the type of any function unless this type is specified in curly brackets immediately after the name of the function; failure to do this will result in a **missing { in <name of function>** error message.

The rule for type operators applies only to 1-place functions; but from chapter 5 we recall that this is no limitation on the expressive power of a functional notation because currying allows us express *n*-place functions in terms of 1-place ones. We consider the function + for instance, as being a 1-place function that when applied to an number *m* returns a function that adds *m* to whatever number *n* is supplied to it. So if we apply + to 1, we return a closure that adds 1 to an object. If we apply that abstraction to another number, we derive the successor of that number (figure 10.3)

```
(1+) (+ 1)
#<CLOSURE :LAMBDA [Z1414] [+ 1 Z1414]> : (number - -> number)

(2+) ((+ 1) 2)
3 : number
```

Figure 10.3 Typechecking and using a partial application

By the rule for the type operator →, the function + has the type number → (number → number). If + is defined in the manner of chapter 5, this is the type that Qi allocates to the function.

```
(3+) (define plus
        {number - -> (number - -> number)}
        0 X -> X
        X Y -> (+ 1 (plus (- X 1) Y)))
plus : (number - -> (number - -> number))
```

In many textbooks, the internal parenthesises for the type operator →, are omitted, so that number → (number → number) is written as number → number → number. Qi permits this shorthand and automatically inserts the missing parentheses. This notational shorthand is used throughout this book.

10.6 The Product Type Operator *

The rule for product types is simple.

Rule for the Product Type

<x,y> : (A * B) just when x : A and y : B.

The function **@p** builds a tuple of elements from a pair of inputs x and y of types A and B respectively. The result is a tuple which inhabits the product type (A * B) . The function **fst** isolates the first element of a tuple and the function **snd** isolates the second element (figure 10.4).

```
(1+) (@p 1 a)
(@p 1 a) : (number * symbol)

(2+)  (fst (@p 1 a))
1 : number

(3+) (snd (@p 1 a))
a : symbol
```

Figure 10.4 Tuples in Qi

Qi allows pattern matching over tuples (using =). This function defines **fst**.

```
(6+) (define my_fst
      {(A * B) - -> A}
      (@p X Y) -> X)
my_fst : ((A * B) - -> A)
```

10.7 Polytypes

What is the type of the identity function?

```
(define identity
  X -> X)
```

We can say that **identity** has the type **string** → **string**, but with equal reason we could also say that it has the type **number** → **number** or **(list symbol)** → **(list symbol).** There is no reason to identify the type of **identity** with any particular one of these types. Instead we say **identity** inhabits the type (**A** → **A**) for any type **A** that one cares to mention. The type **A** → **A** is the **most general type** of all these types; and all the previous types are substitution instances of it. The presence of a variable in the type of **identity** shows that the type of this function is a **polytype.** The inhabitation rule for a polytype is as follows:-

Rule for Polytypes

If A is a polytype, *x* : A just when, for any type B which results from the uniform substitution of types for variables in A, *x* : B .

Functions whose types are polytypes are **polymorphic** functions. There is also one object, which is not a function, that inhabits a polytype - the empty list [ ]. Typing [ ] reveals the type of [ ] to be **(list A).**

```
(2+) [ ]
[ ] : (list A)
```

A very important polymorphic function is the equality function **=**. Objects can be compared using = only when they are of the same type.

```
(3+) (= 6 6)
true : boolean

(4+) (= a 6)
type error
```

Qi also provides a more inclusive equality function **==** which operates exactly as does **=**, but allows comparison of objects of different types. The type of **==** is **A → B → boolean.**[38]

10.8 Using Type Checking

Some of the programs of chapters 2, 3 and 4 can be typechecked using the information in this chapter, provided the constituent definitions are given the appropriate types. In this section, we review some programs from these chapters, seeing how typechecking can be applied to them.

The simple example of I/O in chapter 1 was **(+ 7 (input))** which returned 14 if the user typed 7 after entering it. This expression does not typecheck for the simple reason that it is impossible to anticipate what the user might type to **input**. **input+** is a version of **input** that receives a monomorphic type and places the constraint that whatever the user submits to **input+** must be of that type. Thus the expression **(cons (input+ : number) [])** will typecheck as having the type **(list number)**. Any attempt to enter a non-number to **input+** will get a warning and a request to re-enter (figure 10.5).

[38] In most statically typed languages, **==** is not supported. The reasoning is that since objects of different types can never be equal, there is no point comparing objects of different types to see if they are the same. We can think of an analogy in a world where every person is the citizen of only one country. In such a world the question "Is Mr Jules, French citizen of Paris, one and the same as Mr Jules, resident of London and British citizen" must always be answered in the negative. In the real world, where dual citizenship abounds, such questions are sensible and are sometimes answered positively. The world of programming is closer to the real world in allowing objects to have "dual citizenship" and so Qi caters for this.

```
(4+) (cons (input+ : number) [])
hello
hello is not a number
Reinput:  8
[8] : (list number)
```

Figure 10.5 Using **input+**

Figure 10.6 reproduces the Cartesian product program of chapter 3 with the appropriate types.

```
(define cartesian-product
  {(list A) - -> (list A) - -> (list (list A))}
  [ ] _ -> [ ]
  [X | Y] Z  -> (append (all-pairs-using-X X Z) (cartesian-product Y Z)))

(define all-pairs-using-X
  {A - -> (list A) - -> (list (list A))}
  _ [ ] -> [ ]
  X [Y | Z] -> [[X Y] | (all-pairs-using-X X Z)])
```

Figure 10.6 A type-secure version of the Cartesian product program

The old program computed the Cartesian product of, for example, a list of symbols and a list of numbers forming a list of lists containing symbols and numbers. The type checked version is more restrictive; both input lists must contain elements of the same type. A way of recapturing some of the functionality of the old program is to generate tuples rather than two element lists. Figure 10.7 shows the revised program that uses **@p** in place of the previous list constructor. The types are revised in line with the new approach.

```
(define cartesian-product
  {(list A) - -> (list B) - -> (list (A * B))}
  [ ] _ -> [ ]
  [X | Y] Z  -> (append (all-pairs-using-X X Z) (cartesian-product Y Z)))

(define all-pairs-using-X
  {A - -> (list B) - -> (list (A * B))}
  _ [ ] -> [ ]
  X [Y | Z] -> [(@p X Y) | (all-pairs-using-X X Z)])
```

Figure 10.7 A revised version using tuples instead of lists

Qi allows **[A]** as shorthand for **(list A)** so that the functions in figure 10.7 can be written.

```
(define cartesian-product
 {[A] - -> [B] - -> [(A * B)]}
 [ ] _ -> [ ]
 [X | Y] Z -> (append (all-pairs-using-X X Z) (cartesian-product Y Z)))

(define all-pairs-using-X
  {A - -> [B] - -> [(A * B)]}
  _ [ ] -> [ ]
  X [Y | Z] -> [(@p X Y) | (all-pairs-using-X X Z)])
```

In the backward chaining program of chapter 7, **backchain*** either returns **#\Escape** or the symbol **proved**. This function will not type-check because there is no single type to which **proved** and **#\Escape** both belong. There are several ways around this. One way is to adopt a different output than **proved**, say **#\P**, which gives **backchain*** a character output. A neater alternative is to use **fail-if** so as to create one's own failure object, and nominate a symbol to trigger failure - such as **unproved**. The program in figure 10.8 uses this technique.🗀

```
(define backchain
  {symbol - -> [[symbol]] - -> symbol}
   Conc Assumptions -> (backchain* [Conc] Assumptions Assumptions))

(define backchain*
  {[symbol] - -> [[symbol]] - -> [[symbol]] - -> symbol}
  [ ] _ _ -> proved
  [Goal | Goals] [[Goal <= | Subgoals] | _] Assumptions
  <- (fail-if (/. X (= X unproved))
       (backchain* (append Subgoals Goals) Assumptions Assumptions))
  Goals [_ | Rest] Assumptions -> (backchain* Goals Rest Assumptions)
  _ _ _ -> unproved)
```

Figure 10.8 Using backtracking in a type-secure manner

🗀 Qi Programs/Chap10/proplog version 3.qi

10.9 Loading Files

When static typing is switched on and we load a file in Qi, every expression in that file will be typechecked. The **load** function permits several files to be loaded in chronological order (e.g. **(load "a" "b" "c")** loads files **a**, **b** and **c** in that order). The success of type checking is independent of both the order in which files are loaded and the order of function definitions in a file.

When a file is loaded with type checking enabled, then a type error message may arise. For example, this function placed in a file **prog**.

```
(define factorial
  {number - -> number}
    0 -> 1
    X -> (cons X (factorial (- X 1))))
```

will generate a type error message. In this case, none of the code loaded will be admitted into the Qi type system. After a type secure definition is compiled into Qi; *it can be reloaded only if the type attached to the definition has not changed.* The message **<function name> already has a type** will be generated if you try to change a type.

Exercise 10

1. Define the following functions with types (problems inspired by Werner Hett).
   a. a function **last** that returns the last element of a list;
   b. a function **mirror** which tests to see if a list is of even length and if its first half is a mirror image of its second; thus **[a b c c b a]** iwould pass this test;
   c. a function **remdup** that removes duplicates of a list;
   d. a function **partition** that takes an equivalence relation and a list and partitions the list by the relation. **Thus (partition == [1 1 3 4 2 7 8 5 8 9]) = [[1 1] [3] [4] [2] [7] [8 8] [5] [9]]**. The order of the elements does not matter.
   e. **run-length**; which takes a list as an input and returns a list of pairs giving the frequency of each element. Thus **(run-length [1 1 3 4 2 7 8 5 8]) = [(@p 1 2) (@p 3 1) (@p 4 1) (@p 2 1) (@p 7 1) (@p 8 2) (@p 5 1) (@p 9 1)].**
   f. **duplicate**; which duplicates the elements of a list; **(duplicate [a b c] 3) = [a a a b b b c c c].**
   g. **drop**; which drops every nth element from a list. **(drop [a b c d e f g h i k] 3) = [a b d e g h k]**

   h. **split**; which splits a list into two parts. So **(split [a b c d e f g h i k] 3) = [[a b c] [d e f g h i k]].**

2. Provide correct types to all the functions contained in the following programs.
   a. the prime program of chapter 3.12;
   b. the change counting program of chapter 3.14;
   c. the Goldbach conjecture program of figure 3.15;
   d. the powerset program in figure 4.13;
   e. the Goldbach conjecture program of chapter 4.14;
   f. the bubble sort program in figure 5.10.
   g. the Newton's method program of figure 5.11.

3. Attach to each of the following functions, its most general type.

```
(define prim
  0 X F -> X
  N X F -> (F (- N 1) (prim (- N 1) X F)))

(define mapf
  _ _ [] -> []
  C F [X | Y] -> (C (F X) (mapf C F Y)))

(define prependX
   X -> (append X))


(define has?
  Test [ ] -> false
  Test [X | Y] -> (or (Test X) (has? Test Y)))

(define b
   F Test States -> true       where (has? Test States)
   F Test States
   -> (let NewStates (mapf append F States)
            (if (empty? NewStates)
               false
               (b F Test NewStates))))

(define s*
   _ [ ] -> [ ]
   _ [X] -> [X]
   R [X Y | Z] -> [Y | (s* R [X | Z])]    where (R Y X)
   R [X Y | Z] -> [X | (s* R [Y | Z])])
```

```
(define s
  R X -> (fix (/. Y (s* R Y)) X))

(define a
 F -> (/. X (/. Y (F X Y))))
```

4. Reimplement the semantic net program in figure 6.4 so that a semantic net appears as a list of pairs; each pair being composed of a symbol, and another pair composed of a pointer (is_a or type_of) and a list of properties. Thus asserting that Mark Tarver is both male and a teacher would be represented by the nested pair **(@p Mark_Tarver (@p is_a [male teacher]))**. Since you will not be using a property list, your program will contain a top level function that maintains the semantic net as bound to a formal parameter. This same top level function will query the user as to whether she wishes to assert or query the system. Your program must pass the type-checker.

Further Reading

Types that are not polytypes, are **monotypes** and languages that allow polytypes are **polymorphic languages**. Languages, which are strongly typed but do not admit polytypes are called **monomorphic languages**. An example of a monomorphic procedural language is **Nicholas Wirth**'s **Pascal** to which **Holmes** (1993) provides a good introduction. Edinburgh ML (**Gordon, Milner** and **Wadsworth** (1979)) was the first statically typed polymorphic functional programming language.

Discussions of the relative advantages of typed languages (like SML) and untyped languages (like Lisp) have appeared in user groups. The introduction to **Abelson** and **Sussman** (1996) contains some interesting observations on this topic as does **Backhouse** (1990).

11 Defining Types

For flexibility, any strongly typed language has to allow the user to define types. These types are called **user-defined types**. One way of defining a type α is to give the precise conditions under which an object can be proved to inhabit the type α. In chapter 7, we saw that *refinement rules* could be used to define this relation of provability. In Qi, to define a new type α is to give a set of refinement rules Γ that define the conditions under which a sequent of the form $\Delta >> E : \alpha$ is provable. Languages that borrow on this method of defining types I have referred to elsewhere as **deductively typed** languages, and Qi is, currently, the only language of this kind

11.1 Enumeration Types

The simplest type that may be defined by a series of datatype refinement rules is an **enumeration type**. An enumeration type is defined by simply listing all its inhabitants. Figure 11.1 defines a type **fruit** that is coextensive with the fruits of the fruit-machine of chapter 2. A notational convenience is that Qi does not require typings (i.e. expressions of the form X : A) to be entered with the outer brackets in datatype definitions. The underlines in figure 11.1 are represented in Qi by 1 or more underscores _____.🗀

The effect of this datatype definition is to allow a series of objects to be of more than one type: cherry is both of type **symbol** and of type **fruit**. The identity function of the previous chapter inhabited an infinity of types, but there was a most general type to which this function belonged ($A \rightarrow A$) of which all the others were instances. The latter property is not a feature of the object cherry, which belongs to two types, which are not instances of some more general type. An object is **overloaded** when it inhabits least two of these types, which are not substitution instances of a more general type.

🗀 Qi Programs/Chap11/fruit.qi.

```
(datatype fruit
  ___________
  cherry : fruit;
  __________
  pear : fruit;
  ____________
  orange : fruit;
  ______________
  pineapple : fruit;
  ___________
  lemon : fruit;)
```

Figure 11.1 The datatype **fruit** *that enumerates the fruits of the fruit machine*

Simply entering **cherry** to the Qi top level returns the verdict that **cherry : symbol**. Given that **cherry** is overloaded, Qi always defaults to the base type if possible. To get Qi to agree that **cherry** is now also of type **fruit**, a typing must be entered (figure 11.2).

```
(1+) cherry
cherry : symbol

(2+) cherry : fruit
cherry : fruit

(3+) apple : fruit
type error
```

Figure 11.2 Recognising fruits

11.2 Side Conditions

A side condition to a refinement rule is a condition that must be satisfied for the refinement rule to be correctly applied, but which is not conveniently expressible in sequent notation. In Qi these are expressed by the **if** keyword, placed before the refinement rule, the keyword itself to be followed by some boolean expression *E*. If *E* evaluates to **false** when the refinement rule is applied to a proof, then the refinement rule fails in its application (the proof is unchanged). Side conditions can be used to shorten a definition. Figure 11.3 shows the use of a side condition to shorten the definition of the datatype **fruit**.

```
(datatype fruit

  if (element? Fruit [cherry pear orange pineapple lemon])
  __________
  Fruit : fruit;)
```

Figure 11.3 Using a side condition to simplify a definition

11.3 Synonyms

If representing a type takes a lot of space, then it makes sense to define it using some shorthand symbol. Suppose we have a program in which tuples of the type **(number * number)** are used as Cartesian co-ordinates. Rather than continuously write **(number * number)** we may wish to define a type **coor** to mean the same as **(number * number).** The synonyms declaration allows this to be done simply. Here is a declaration that **coor** is short for **(number * number)** and **plot** is short for **(list coor).**[39]

```
(synonyms

  coor (number * number)
  plot [coor])
```

Figure 11.4 Using the synonyms declaration to create synonyms

11.4 Left and Right Rules

Suppose we are defining a type **details**, where **details** is a two element list composed of a string (representing a person's name) and a number (representing a telephone number). Obviously it is sensible to assert.

```
           \ refinement rule details_right \
         Name : string; Telephone : number;
              [Name Telephone] : details;
```

Notice that **;** follows **Name : string** and **Telephone : number**, indicating that these are *two* separate goals that must be solved if **[Name Telephone] : details** is to be proved. Let us also assume an obviously true refinement rule.

[39] Inbuilt types and inbuilt type operators cannot be made synonyms. These are **-->**, *****, **list, array**, **goals** and **goal-object**.

\ *sequents refinement rule* \

$$\frac{}{\text{X : A >> X : A;}}$$

We can now prove *a* : **string,** *b* : **number >> [***a b***] : details.**

a : **string,** *b* : **number >> [***a b***] : details** *if*

a : **string,** *b* : **number >>** *a* : **string**
and *a* : **string,** *b* : **number >>** *b* : **number** (by ***details_right***)

which are both proved by ***the sequents refinement rule***

However ***details_right*** alone is not enough. Suppose the problem to solve is **[***a b***]** : **details** >> *a* : **string. T**he ***details_right*** rule supplied does not enable the sequent to be proved, since the assumption, and not the conclusion is of the form **[***a b***]** : **details**. To complete the identification of the two types a second refinement rule is needed.

\ *refinement rule details _left* \
Name : string, Telephone : number >> P;
[Name Telephone] : details >> P;

Notice that **Name : string** and **Telephone : number**, are to the left of >> indicating that these are *two* separate assumptions. The proof is now easy.

[*a b***]** : **details** >> *a* : **string** *if*
a : **string,** *b* : **number >>** *a* : **string** (by ***details_left***)

which is itself proved by ***the sequents refinement rule***

The two refinement rules, ***details_right*** and ***details_left***, are the **right** and **left** refinement rules for the type **details**. In defining a set of refinement rules for a datatype D, often we must allow for objects of type D to occur in the *assumptions* of a sequent to be proved, (i.e. to the *left* of the >>) or within the *conclusion* of the sequent (i.e. to the *right* of the >>). The left and right refinement rules for D describe what may be done in such cases. The left refinement rules say what we may prove *from* assumptions involving objects of type D, and the right refinement rules what is needed *to* prove conclusions that objects are of type D. In Qi, the sequents refinement rule is part of its inbuilt type theory (of which more in chapter 13), hence we do not have to include this refinement rule in our definition of **details**.

Qi permits a shorthand for entering left and right rules. Instead of writing two rules.

```
          \ refinement rule details_right \
       Name : string; Telephone : number;
       __________________________________
           [Name Telephone] : details;

          \ refinement rule details _left \
    Name : string, Telephone : number >> P;
    _______________________________________
        [Name Telephone] : details >> P;
```

an accepted abbreviation is

```
       Name : string; Telephone : number;
       ==================================
           [Name Telephone] : details;
```

11.5 Defining Recursive Types

The type binary is the type to which all binary numbers belong. A binary number will be identified with a list of 1s and 0s with [0] and [1] as the simplest binary numbers. How might binary be defined as a type? [0] and [1] are binary numbers. Given any list (cons X (cons Y Z)), this list is binary if and only if X is 1 or 0 and (cons Y Z) is binary. If (cons X (cons Y Z)) is written as [X Y | Z], then the definition of this datatype appears in figure 11.5.

The datatype of binary numbers is introduced by stating the refinement rules used to manage proofs involving that datatype. The base refinement rule gives the simplest case; a list of one element (0 or 1) is a binary number.

```
(datatype binary

        if (element? X [0 1])
        _____________________
        X : zero-or-one;

        X : zero-or-one;
        ________________
        [X] : binary;
```

Qi Programs/Chap11/binary.qi

```
\ binary_right \
X : zero-or-one; Y : binary;
[X | Y] : binary;

\ binary_left \
X : zero-or-one, [Y | Z] : binary >> P;
[X Y | Z] : binary >> P;)
```

Figure 11.5 Defining the datatype of binary numbers

binary_right says how to prove that objects are binary; provided that **Y** is binary and **X** is either 0 or 1, then **[X | Y]** is binary. ***binary_left*** says **[X Y | Z] : binary** can be replaced as an assumption **by X : zero-or-one** and **[Y | Z] : binary**. Functions using this datatype can now be defined (figure 11.6).

```
(27+) (define complement
        \calculates the complement of a binary number\
        {binary - -> binary}
        [0] -> [1]
        [1] -> [0]
        [1 N | X] -> [0 | (complement [N | X])]
        [0 N | X] -> [1 | (complement [N | X])])
complement : (binary - -> binary)
```

Figure 11.6 Operating with the binary number datatype

11.6 Simulating a Calculator

To demonstrate the utility of recursive types, we will consider the simulation of a simple calculator. Our calculator program takes numbers or lists and evaluates them to a number result (figure 11.7). Having loaded the program and set it running, we can calculate the answers to problems like **[88.9 + 8.7]** and **[56.8 * [45.3 - 21.7]].**

```
(define do-calculation
   [X + Y] -> (+ (do-calculation X) (do-calculation Y))
   [X - Y] -> (- (do-calculation X) (do-calculation Y))
   [X * Y] -> (* (do-calculation X) (do-calculation Y))
   [X / Y] -> (/ (do-calculation X) (do-calculation Y))
   X -> X)
```

Figure 11.7 A calculator program written for core Qi

However, as soon as type checking is switched on, entering the program produces a type error, because **do-calculation** is designed to deal with mixed lists and strong typing does not allow such lists. Recursive types provide a way of tackling this problem (figure 11.8).

```
(datatype arith-expr

   X : number;
   X : arith-expr;

   if (element? Op [+ - * /])
   X : arith-expr; Y : arith-expr;
   ====================
   [X Op Y] : arith-expr;)
```

Figure 11.8 Our first attempt at defining a datatype of arithmetic expressions

Armed with these datatype rules, we attempt to define our calculator (figure 11.9).

```
(2+) (define do-calculation
       {arith-expr - -> arith-expr}
       [X + Y] -> (+ (do-calculation X)
                             (do-calculation Y))
       [X - Y] -> (- (do-calculation X) (do-calculation Y))
       [X * Y] -> (* (do-calculation X) (do-calculation Y))
       [X / Y] -> (/ (do-calculation X) (do-calculation Y))
       X -> X)
Correctness Check Failure: rule 1 of do-calculation
```

Figure 11.9 An erroneous attempt to produce a type secure calculator

The **do-calculation** program fails to type check because + produces numbers and not **arith-exprs** as required by our type assignment. Suppose we change the type of do-calculation to **arith-expr** → **number**.

```
(3+) (define do-calculation
      {arith-expr - -> number}
      [X + Y] -> (+ (do-calculation X) (do-calculation Y))
      [X - Y] -> (- (do-calculation X) (do-calculation Y))
      [X * Y] -> (* (do-calculation X) (do-calculation Y))
      [X / Y] -> (/ (do-calculation X) (do-calculation Y))
      X -> X)
Correctness Check Failure: rule 5 of do-calculation
```

Figure 11.10 Another erroneous attempt to produce a type secure calculator

Now the final rewrite rule fails to typecheck because Qi finds that it cannot prove that the input X is a number from the assumption that X is an **arith-expr** (remember that our rules say that all numbers are **arith-exprs**; not that all **arith-exprs** are numbers).

Our solution is used within the SML family of functional programming languages, which is to use a **label** (sometimes confusingly called a **constructor function**) to mark out the numbers within an **arith-expr**. Labels act as markers for the type-checker by unambiguously signalling the types of the objects they label. Figure 11.11 introduces the label **num** as a way of marking out numbers.

```
(datatype  arith-expr

   X : number >> Y : A;
   [num X] : arith-expr >> Y : A;

   X : number;
   [num X] : arith-expr;

   if (element? Op [+ - * /])
   X : arith-expr; Y : arith-expr;
   =====================
   [X Op Y] : arith-expr;)
```

Figure 11.11 Using labels

This solution works with the following definition of **do-calculation**. 🗀

🗀 Qi Programs/Chap11/calculator.qi

```
(define do-calculation
  {arith-expr - -> number}
   [X + Y] -> (+ (do-calculation X) (do-calculation Y))
   [X - Y] -> (- (do-calculation X) (do-calculation Y))
   [X * Y] -> (* (do-calculation X) (do-calculation Y))
   [X / Y] -> (/ (do-calculation X) (do-calculation Y))
   [num X] -> X)
```

Figure 11.12 A type secure version of do-calculation

11.7 Verified Objects

The use of labels in our **do-calculation** program makes for confusing reading - typing **in [[num 2] + [num 3]]** is a counterintuitive way of adding 2 and 3. **Verified objects** make life easier. A verified object is an object that belongs to the type **verified**. The inhabitation rule for this type is:

X : **verified** just when the normal form of X is **true**.

Unlike the other types so far encountered, the type verified is non-decidable; that is, there is no decision procedure for deciding whether an object belongs to this type or not. Consequently none of the self-evaluating objects of Qi will be recognised as belonging to this type - not even **true**!

At this point you will wonder what the point of such a type is: actually, this type is very useful. It allows us to write a nicer **do-calculation** program.

When Qi typechecks definitions that include guards, it assumes that if the rule fires, then the guard must evaluate to **true** and hence Qi assumes that, in that circumstance, the guard is of the type **verified**. For instance, we can assume that if an input **X** passes the guard **(number? X)** (which tests for numberhood) that **X** is a number so, we can write:

```
_____________________________
(number? X) : verified >> X : number;
```

If we insert this rule in the datatype rules for **arith-expr**, and place a guard in the **do-calculation** function, then the whole thing typechecks.

```
(datatype  arith-expr

   ______________________________
   (number? X) : verified >> X : number;

   X : number;
   ___________
   X : arith-expr;

   if (element? Op [+ - * /])
   X : arith-expr; Y : arith-expr;
   ====================
   [X Op Y] : arith-expr;)

(define do-calculation
  {arith-expr - -> number}
   [X + Y] -> (+ (do-calculation X) (do-calculation Y))
   [X - Y] -> (- (do-calculation X) (do-calculation Y))
   [X * Y] -> (* (do-calculation X) (do-calculation Y))
   [X / Y] -> (/ (do-calculation X) (do-calculation Y))
   X -> X          where (number? X))
```

Figure 11.13 A type secure version of do-calculation using verified objects

11.8 Defining Type Operators

The operators list, → and * are the base type operators of Qi, but datatype refinement rules allow the introduction of new type operators. As an example, we shall consider **streams**. Streams are a device used by functional programmers for manipulating infinitely large collections of objects without having to generate each element of them. A stream requires an ordering of the elements of the collection such that

1. There is a first element of this ordering
2. For element in the ordering there is a a computable method for finding the unique next element in the ordering.

Thus suppose we wish to work with a stream that represents the set of natural numbers. There are an infinite number of natural numbers and so it is impossible to place them in a finite list. However they may be represented as a stream in which the first element is the number 0 and *f* is the function (+ 1) that adds 1 to any natural number. The stream of natural numbers can represented as the tuple <0, (+ 1)>.

By means of the appropriate functions, this representation can be made to behave like an infinitely long list of natural numbers. In chapter 4 we saw that a list could be decomposed through the head and tail functions. If streams can act as proxies for lists, then the corresponding versions **stream-head** and **stream-tail** should be definable. Intuitively, the tail of the stream of the natural numbers beginning with 0 will be the stream of natural numbers beginning with 1 (i.e. stream-tail(<0, (+ 1)>) = <1, (+ 1)>) and the **stream-head** of the stream of natural numbers beginning with 0 is just 0 (i.e. stream-head(<0, (+ 1)>) = 0). This operation can be generalised; the tail of a stream <X, *f*> is the stream <(*f* X), *f*> formed by pairing the result of applying the stream function to the initial element of the stream with the stream function itself. The head of a stream <X, *f*> is just X.

In Qi streams are plausibly identified as tuples of elements and functions of the right type. Since it is possible to construct streams for natural numbers, characters etc., we can meaningfully talk of streams of natural numbers, strings and characters. Streams are then a polytype operation over types. The functions **stream-head** and **stream-tail** can then be defined as well (figure 11.14).

```
(datatype stream

   X : (A * (A - -> A));
   ==============
   X : (stream A);)

(define stream-tail
  {(stream A) - -> (stream A)}
   (@p X F) -> (@p (F X) F))

(define stream-head
   {(stream A) - -> A}
    (@p X F) -> X)
```

Figure 11.14 Streams as a datatype in Qi

11.9 Working with Functions Other than Cons

Using the square brackets [and] to write refinement rules is only a notational convenience; we could choose to write **(cons X (cons Y Z))** instead of **[X Y | Z]** in the definition of **binary** and the refinement rules would work in exactly the same way. Moreover the function **cons** is only one particular constructor, and there is no reason why a datatype definition should be restricted to working with just that one constructor.

As a case study we will consider developing a datatype definition for the natural numbers. We could use a successor notation, whereby 3 is defined as **[succ [succ [succ 0]]]**, but this is a clumsy representation when even moderately sized numbers are involved. Instead of this, we will use the conventional decimal representation of the natural numbers. Our datatype definition contains three rules; the first states that anything is a natural number if it is an integer greater than -1. The second says that the result of adding 1 to a natural number is also a natural number. The last says that subtracting 1 from a natural number is also a natural number if that number is not 0.

```
(datatype natnum

   if (integer? X)
   if (> X -1)
   ____________________
   X : natnum;

   X : natnum;
   ______________
   (+ X 1) : natnum;

   (not (= X 0)) : verified; X : natnum;
   ______________
   (- X 1) : natnum;)
```

Figure 11.15 Defining the datatype of natural numbers

The definition of addition over natural numbers is now straightforward.

```
(define plus
 {natnum --> natnum --> natnum}
  0 X -> X
  X Y -> (plus (- X 1) (+ Y 1) )    where (not (= X 0))       )
```

Figure 11.16 Defining natural number addition

11.10 Subtypes

One problem with the addition program of the previous section is that it does not work in hand with our other arithmetical operations. This is a disadvantage, because the operations that return natural numbers should be able to return objects to functions that accept numbers. This is because natural numbers are a **subtype** or special case of the type of all numbers.

We want to be able to enter **(- (plus 34 5) 7.5)** and get it past the typechecker without disturbing our datatype rules. To this we define the concept of a subtype and add that natural numbers are a subtype of numbers.

```
(datatype subtype

 (subtype B A); X : B;
 _________________
  X : A;
______________________
(subtype natnum number);        )
```

If we now enter **(- (plus 34 5) 7.5)** then we get **31.5 : number**.

11.11 Type Checking Assignments

If a global variable is used in conjunction with strong typing in Qi then it requires a refinement rule that states what type of object the global identifier holds. The expression **(set *my-name* mark)** will not typecheck unless it is stated within a datatype definition what is the type of **(value *my-name*)**. Arrays are declared in this way. So to declare the array of chapter 6, we declare the type of the expression **(value *ordinance-survey*)** to be an array of symbols (figure 11.16).[40]

```
(datatype some_global_objects

________________________
(value *my-name*) : symbol;

_____________________________________
(value *ordinance-survey*) : (array symbol);)
```

Figure 11.17 Declaring the types of some global objects

[40] Property lists cannot be typechecked in Qi.

11.12 Structures

Suppose we want to define a ship as a type with the following component parts; the name - which is a string of some kind; the nationality - which is a symbol; the tonnage - which is a number and the length - which is also a number. The type **ship** could be defined as a datatype, but it is better to define it as a **structure** (figure 11.18).

```
(structure ship
  name string
  nationality symbol
  tonnage number
  length number)
```

Figure 11.18 Defining a ship as a structure in Qi

What does **structure** do? It creates a type **ship** with a **constructor** function **make-ship** of type **string** → (**symbol** → (**number** → (**number** → **ship)))** and the following **selector** functions.

1. **ship-name** of type **ship** → **string**,
2. **ship-nationality** of type **ship** → **symbol**,
3. **ship-tonnage** of type **ship** → **number**,
4. **ship-length** of type **ship** → **number**, ….
5. …. and a **recognisor** function **ship?** of type **A** → **boolean** that recognises ships

Figure 11.19 shows an example of their use.[41]

```
(1+) (structure ship
        name string
        nationality symbol
        tonnage number
        length number)
ship
```

[41] The expression **#<ship :name Sea Venus :nationality greek :tonnage 1055.9 :length 568.8>** cannot itself be evaluated. The normal form representation of structures depends on Lisp and is not readable as such by Qi. Hence **make-ship is** used to refer to the structure **#<ship :name Sea Venus :nationality greek :tonnage 1055.9 :length 568.8>.**

```
(2+) (make-ship "Sea Venus" Greek 1055.9 568.8)
#<ship :name Sea Venus :nationality Greek :tonnage 1055.9 :length 568.8> :
ship

(3+) (ship-tonnage (make-ship "Sea Venus" Greek 1055.9 568.8))
1055.9 : number
```

Figure 11.19 Defining and using a structure

Structures are useful for several reasons.

1. They automatically generate for us a batch of constructor, selector and recognisor functions that we might otherwise have to define ourselves.
2. They allow efficient processing, instead of having to rely on functions that contain pattern-matching, we can access the appropriate parts of the type using selector functions.
3. They allow us to redefine types without problems. If later we decide that the type **ship** is to include an extra slot called **horsepower** of type number, then we simply add it without disturbing our other selector functions which behave in the same way as they did before.
4. Unlike datatypes, the speed of typechecking is invariant of the number of structures declared. More structures do not slow typechecking.

One glitch with structures (inherited from Lisp) is that **=** and **==** always return **false** for structures. To compare two structures, you need to define a function that compares structures on the basis of the equality of their components or use the **congruent?** function (see appendix A).

11.13 Abstract Data Types

Structures are powerful because they allow us to hide away inessential information about how objects are represented and make changes easily; a technique called **information hiding**. Their disadvantage is that they only cater for objects with a fixed number of components. But if we want to use the techniques of information hiding with respect to objects that are recursively defined, and that can be of any size, then we have to use **abstract data types**.

A good example is a stack. A stack is a structure into which objects are placed or **pushed** one by one. Objects can also be removed or **popped** one by one. The basic rule governing stacks is that the object popped is the last object that was pushed onto the stack. A concrete example is a the magazine of a rifle, in which the last bullet pushed into the magazine is the first to be fired. The stack can be empty or it can contain any number of elements. This last observation means that we cannot use structures, which rely on knowing that the stack will only have a fixed finite number of components.

However we can approach the problem of defining stacks using the same philosophy as we found in structures; that is, we define stacks by (a) defining the appropriate constructor, selector and recognisor functions and stating how they behave and (b) defining their types.

The simplest stack is the empty stack; a stack with no elements. Let us define a 1-place function **empty-stack** that returns the empty-stack.

```
(define empty-stack
    _   -> "e!")
```

We have chosen **"e!"** as the empty stack; the choice is arbitrary. The function **push** pushes an element onto a stack.

```
(define push
  X S -> [X | S])
```

pop removes the element at the top of the stack and returns the rest of the stack.

```
(define pop
  X -> (error "empty stack!")      where (= X (empty stack))
  [X | S] -> S)
```

Finally **top** returns the top of the stack.

```
(define top
 X -> (error "empty stack!") where (= X (empty stack))
 [X | S] -> S)
```

We need now to enter the types we want to associate with these functions. Since stacks, like streams, can contain elements of any kind, we need to define a stack as an operator over types. The empty stack is a polymorphic object of type (stack A). **push** pushes an object of type A onto a stack of type (stack A). **top** returns an element of type A from a stack of type A. Finally **pop** returns a stack of type A given a stack of type A.

Rather than scattering all this material over the program, it is better to draw it together into one place. The **abstype** declaration does precisely this (figure 11.20) .

```
(abstype stack

(:types  empty-stack  :  (A --> (stack B))
         push : (A --> (stack A) --> (stack A))
         top : ((stack A) --> A)
         pop : ((stack A) --> (stack A))                    )

(:defs   (define empty-stack
                 _ -> "e!")

         (define push
                 X S -> [X | S])

         (define top
                 X -> (error "empty stack!") where (=  X (empty-stack ok))
                  [X | S] -> X)

         (define pop
                  X -> (error "empty stack!") where (= X (empty-stack ok))
                 [X | S] -> S)      )          )
```

Figure 11.20 Defining a stack as an abstract datatype

```
(1+) (empty-stack ok)
"e!" : (stack A)

(2+) (push 0 (empty-stack ok))
[0  . "e!"] : (stack number)

(3+) (top [0  | "e!"])
error: type error
```

The final input fails, because the type checker has no understanding of what a stack is apart from the operations used to build it. We try again

```
(4+) (top (push 0 (empty-stack ok)))
0 : number
```

The example shows that the actual concrete representation for an abstract datatype like stack is not used in the type checker. So although our empty stack is a string, the command **(output (empty-stack ok))** will not type check (though **(output "~A" (empty-stack ok))** will). Our stacks must be handled by means of the constructor, accessor and recognisor functions that we have laid down.

11.14 Changing the Type Environment

Every time a datatype is declared, the type environment of Qi changes. The natural progression is that the type environment grows larger. As this process proceeds, the performance of the typechecker degrades in a roughly linear relation to the number of type rules in the system. If the number of rules approaches several hundred, then the type checker can become very slow, particularly on large function definitions.

Generally in such large systems, with a great many types in use, the system is generally the product of more than one hand. In such a case, the programmer working on one part of the system might actually use only a fragment of the type discipline of the whole system. Nevertheless, the type checker, unless instructed, will slog through the program armed with the entire type system.

The commands **preclude**, **include**, **preclude-all-but** and **include-all-but** allow the programmer to optimise the type checker to the program at hand. Thus it is possible to have a massive type system in Qi and programs of many thousands of lines, and yet type check them in reasonable speed by configuring the type checker to ignore those types whose rules are not needed.

The command **preclude** accepts a list of symbols which are names of datatypes and sets them aside from use in type checking. Placing the command **(preclude [streams])** at the head of a file will cause the type checker to ignore the rules for streams. The command **(include [streams])** works the opposite way, placing the rules back into the type environment. The command **(preclude-all-but [streams])** will preclude all user defined datatypes apart from streams, whereas **(include-all-but [streams])** will include all user defined datatypes apart from streams. Reasonably enough, **(preclude-all-but [ ])** will set aside all user defined datatypes and **(include-all-but [ ])** will include all of them.

Overwriting old datatype definitions or synonyms (as opposed to simply setting them aside) with new ones can make previously type secure programs type insecure and is not encouraged. Qi warns over such attempts at redefinition. Similarly redefining functions and changing their types will raise warning messages for the same reason. In both cases, if the user really wants to to be sure of type security, the proper course is to exit and start from scratch. Entering **(strong-warning +)** will cause Qi to make all warnings into error messages.

11.15 Optimising Search

The default Qi search strategy is to use every user-defined rule and, on failure, to backtrack to the last rule used to see if it can be legally applied in a different way and if not, to try the next one. This is pretty much the chronological backtracking approach described in chapter 7. It is possible to change this strategy by use of the keyword **commit!** placed before a rule (e.g.).

```
(datatype l_formula

  commit!
  X : l_formula; Y : l_formula;
  ====================
  [@p X Y] : l_formula;
...... ...... ...... ......)
```

The effect of this keyword is that if the rule is successfully applied, the type checker does not, on later failure, backtrack to that choicepoint or any other rules within that datatype. **commit!** has to be used with some caution as it can, if wrongly applied, block the verification of a type secure program.

Exercise 11

1. Define the following enumeration types.
   a. **coin** for British coinage and type-check the **next-denomination** function in chapter 3 using this type (count 0 as of type **coin** for this question).
   b. **fruit** and **winnings** for the fruit-machine of chapter 2 so that winnings are numbers which represent potential wins from the machine.

2. Define a type **formula** for Proplog formulae so that **p**, **[p <= [q & r]]** etc. are all of type **formula.** Using your answer, build an interactive Proplog system that enables the user to enter a goal to your system and applies backward chaining to solve it.

3. *Define **set** as a type operator such that **X : (set A)** just when **X** is a list of **A**s with no duplicated elements.

   a. Define a function **coerce-to-set** of type **[A]** → **(set A)** that acts as an identity function if it receives an object of type **(set A)** and if it does not returns an error message**.**
   b. Define a function **acons** of type **A** → **(set A)** → **(set A)** that receives an

object **O** and a list of objects **L** and conses **O** to **L** provided **O** is not an element of **L**. **If O** is an element of **L**, **acons** returns **L.**

c. Define functions **hd** and **tl** that act as **head** and **tail**, but have the types **(set A)** → **A** and **(set A)** → **(set A)** respectively.

d. Define **set-union** of type **(set A)** → **(set A)** → **(set A)** that unions sets together.

e. Define **set-intersection** of type **(set A)** → **(set A)** → **(set A)** that gives the intersection of two sets.

f. Define **set-difference** of type **(set A)** → **(set A)** → **(set A)** that gives the difference of two sets.

4. *Rewrite the cribbage program set in exercise 4.8, so that it is type-checked. A card like J♥ can be represented as a pair (@p 11 hearts) composed of a number indicating rank and the symbol showing the suit. Define **rank** and **suit** as enumeration types and **card** as a synonym for pairs composed of a rank and a suit. Use the answer to 11.3 to ensure that players are not dealt duplicate cards.

5. What effect does **commit!** have in the following?

```
(datatype foo

  commit!
  X : number;
  X : foo;

  X : symbol;
  X : foo;)
```

6. *The function **read-file-as-charlist** (see appendix A) reads the contents of a text file as a list of characters, returning that list as an output. Given the following simple grammar.

<expr> ::= <number> | (<expr> <op> <expr>)
<op> ::= + | - | * | /
<number> ::= <digit> | <digit>^<number> (^ shows there is no space between digits)
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

devise a function **read-exprs** of type **string** → **(list expr)** that reads in characters from a file and outputs expressions. If the file cannot be constructed as having only expressions in it, then **read-exprs** should return an error message.

7. **Read Wikstrom or Paulson on Standard ML and design a *type-secure* **cross-compiler** from SML into core Qi. Your program should read in an ML program

from a file as a list of characters (see read-file-as-charlist in appendix A) and either output the corresponding Qi program into a different file (see write-to-file in appendix A) or evaluate the output on the fly so as to compile the Qi code into the current image. You can ignore ML type definitions, but you need to decide what to do with ML constructor functions. The easiest course is to represent them as simple list constructions.

Further Reading

The sequent based notation used to define types in Qi was introduced by **Gentzen** (1934) and **Diller** (1990) and **Duffy** (1991) provide good introductions to the use of this notation in modern logic. Structures are part of Common Lisp, and Qi structures are a species of Lisp structure; see **Steele** (1990) 468-489 for more details. Other languages like SML provide abstract datatypes for hiding implementation-dependent representations of data - see **Wikstrom** (1988) and **Paulson** (1996) for more on this. **Abelson** and **Sussman** (1996) provide an excellent discussion on the use of streams in programming.

12 Compiling Qi

In this chapter and the next, we are going to delve more deeply into some of the theory behind Qi. In particular we are going to examine how Qi functions are compiled into the lambda calculus mentioned in chapter 1. This compilation process is a two-stage procedure whose first stage compiles Qi into an enriched version $\mathcal{L}$ of the lambda calculus. The second stage compiles $\mathcal{L}$ into lambda calculus. The significance of $\mathcal{L}$ becomes apparent in the next chapter, because $\mathcal{L}$ expressions can be type checked. We shall see that this process of type checking is actually a process of proof, and can be described using the same refinement rule notation used to define types in Qi. To begin, we look at the lambda calculus.

12.1 The Lambda Calculus

The lambda calculus, invented by **Alonzo Church** (1940), provides a notation for talking about functions. Just as in Qi, when we wish to apply a function *f* to an input *i*, the lambda calculus requires us to write (*f i*). The function square is said to be **applied** to 3, and the whole expression (square 3) is called an **application**.

Strictly speaking, the syntax of the pure lambda calculus makes no explicit allowance for referring to numbers, so (square 3) is not an expression of the pure lambda calculus. Many authors allow the pure lambda calculus to be enriched by a set of constants (**Hindley** and **Seldin** (1986) call this an **applied lambda calculus**), and they allow these constants to count as lambda calculus expressions. We will follow this example, and for the moment admit numbers and names of functions into the lambda calculus. Later in this chapter, we shall see how they can be reduced to expressions of the pure lambda calculus. The syntax rules for the pure lambda calculus are given next.

Syntax Rules for the Pure Lambda Calculus

1. x,y,z,x′,y′,z′,.... are all variables.
2. Any variable on its own is an expression of the lambda calculus.
3. If e_1 and e_2 are lambda expressions then the **application** (e_1 e_2) of e_1 to e_2 is an expression of the lambda calculus.
4. If e is an expression of the lambda calculus and v is a variable then (λ v e) (called an **abstraction**) is an expression of the lambda calculus.
5. Nothing else is an expression of the lambda calculus.

Abstractions are used within the lambda calculus to reveal the internal structure of a function. An abstraction has the form (λ v e), where v is a variable and e is an expression of lambda calculus. "(λ v e)" signifies a function that given some input v, returns the output e. The simplest example is the abstraction (λ x x), which denotes the **identity function** that receives any input x and returns x. Here are some more examples.

Abstraction	Interpretation
(λ x 3)	receives any input x and returns 3.
(λ x y)	receives any input x and returns y.
(λ x (x 5))	receives any input x and returns the result of applying x to 5.
(λ x (x x))	receives any input x and returns the result of applying x to x.
(λ x (λ y (x y)))	receives any input x and returns a function (λ y (x y)) which receives any input y and applies x to y.

Figure 12.1 Some abstractions and their interpretations

An occurrence of a variable v is **bound** in an abstraction if v occurs within an expression of the form (λ v e), and if it is not bound it is **free**. It is important to distinguish between *occurrences* of variables and variables themselves. The same type of variable may occur several times in one lambda expression and, in relation to that expression, some of the occurrences can be free and some can be bound. For example, in the application ((λ x (y x)) x) there are three occurrences of the variable x. Reading from left to right, the first two are bound, and the last is free. A lambda expression where all variables are bound is said to be a **closed lambda expression** or **combinator**.

We perform a **substitution for the free variable occurrences** within a lambda expression when we uniformly replace all free occurrences of that variable throughout the lambda expression. For example, substituting z for free occurrences of x in ((λ x (y x)) x) gives ((λ x (y x)) z)); the first two occurrences of x are not replaced because they are bound. A substitution of an expression y for the free occurrences of a variable x in a lambda expression e is written $[e]_{y/x}$. Here are some more examples.

Expression	Result
$[(\lambda\ x\ x)]_{y/x}$	$(\lambda\ x\ x)$
$[(\lambda\ y\ x)]_{z/x}$	$(\lambda\ y\ z)$
$[(\lambda\ x\ (x\ y))]_{y/z}$	$(\lambda\ x\ (x\ y))$
$[(\lambda\ x\ (y\ (\lambda\ y\ (x\ y))))]_{z/y}$	$(\lambda\ x\ (z\ (\lambda\ y\ (x\ y))))$

Figure 12.2 Performing Substitutions

12.2 Reasoning with the Lambda Calculus

The lambda calculus was originally designed as a formal system for proving the equality of functions expressed in lambda calculus notation. We write $e_1 = e_2$, where e_1 and e_2 are lambda calculus expressions, to state that e_1 and e_2 denote the same function. Obviously if e_1 and e_2 are syntactically identical then $e_1 = e_2$ holds. This is the first rule of the lambda calculus.

$e = e$

The next rules deal with the obvious properties of =; symmetry and transitivity.

From $e_1 = e_2$ derive $e_2 = e_1$.
From $e_1 = e_2$ and $e_2 = e_3$ derive $e_1 = e_3$.

The next three rules state the principle that equals may be substituted for each other.

From $e_1 = e_2$ derive $(e_3\ e_1) = (e_3\ e_2)$
From $e_1 = e_2$ derive $(e_1\ e_3) = (e_2\ e_3)$
From $e_1 = e_2$ derive $(\lambda\ v\ e_1) = (\lambda\ v\ e_2)$

In addition, there are three more rules for proving lambda expressions equal; the rules of **α conversion, β reduction** and **η conversion**.

$(\lambda\ x\ x)$, $(\lambda\ y\ y)$ and $(\lambda\ z\ z)$ are all ways of referring to the identity function. Consequently the equations $(\lambda\ x\ x) = (\lambda\ y\ y)$ and $(\lambda\ y\ y) = (\lambda\ z\ z)$ are both true, and should be provable in the lambda calculus. α conversion allows this proof by legitimising the uniform replacement of a bound variable within an abstraction by some other variable. We write $(\lambda\ x\ x) \rightarrow_{\alpha} (\lambda\ y\ y)$ to show that $(\lambda\ x\ x)$ is convertible to $(\lambda\ y\ y)$ by α conversion. It does not matter what replacement variable is used, provided that the replacement is performed uniformly and there is no **variable capture**.

A variable capture occurs when the effect of inserting the replacement variable is to bind a hitherto free variable. For instance the function $(\lambda\ x\ (x\ y))$ can be α converted to $(\lambda\ z\ (z\ y))$, but it cannot be α converted to $(\lambda\ y\ (y\ y))$ without capturing or binding the previously free y. We formulate the rule for α conversion so that variable captures are not allowed.

The Rule of α Conversion

Any expression $(\lambda\ v\ e)$ may be α converted to $(\lambda\ v'\ [e]_{v'/v})$ provided v' does not occur free in e.

If e_1 and e_2 can be made identical by α conversions alone then they are **congruent** to each other and this is written $e_1 \approx e_2$.

The application of the function $(\lambda\ x\ (\text{square}\ x))$ to 3 produces (square 3). We can write this as (square 3) or $((\lambda\ x\ (\text{square}\ x))\ 3)$. In any event, the following equation must be true.

$$((\lambda\ x\ (\text{square}\ x))\ 3) = (\text{square}\ 3)$$

Deriving (square 3) from $((\lambda\ x\ (\text{square}\ x))\ 3)$ requires replacing each free occurrence of x within (square x) by 3 and eliminating the λ x; an operation called **β reduction**. This operation is written $((\lambda\ x\ (\text{square}\ x))\ 3) \rightarrow_{\beta} (\text{square}\ 3)$. Here is an example where β reduction is applied twice.

$$((\lambda\ x\ ((\lambda\ y\ (x\ y))\ z))\ z) \rightarrow_{\beta} ((\lambda\ x\ (x\ z))\ z) \rightarrow_{\beta} (z\ z)$$

As with α conversion, there is a proviso to β reduction. We must take care with expressions where a variable has both free and bound occurrences. Thus suppose we attempt to β reduce $((\lambda\ x\ (\lambda\ y\ (x\ y)))\ y)$ without regard to the fact that y occurs both free and bound in different occurrences; we derive

$$((\lambda\ x\ (\lambda\ y\ (x\ y)))\ y) \rightarrow_{\beta} (\lambda\ y\ (y\ y)) \quad ???$$

However if we should first do α conversion on ((λ x (λ y (x y))) y) to get ((λ x (λ z (x z))) y), and then perform β reduction we derive (λ z (y z)); but plainly (λ z (y z)) $\neq$ (λ y (y y)).[42]

The reduction of ((λ x (λ y (x y))) y) to (λ y (y y)) is wrong because the second occurrence of y in ((λ x (λ y (x y))) y) (which is free) becomes bound if placed in the scope of λ y. In this case, to perform β reduction, it is first necessary to rename the bound occurrences of y using α conversion, and only then can β reduction be performed. The rule of β reduction, like the rule for α conversion, has a clause within it stipulating that variables may not be captured.

The Rule of β Reduction

((λ v e_1) e_2) may be transformed by β reduction to $[e_1]_{e_2/v}$ provided no free occurrence of a variable in e_2 becomes bound.

In some cases, the result of performing all possible β reductions is an expression free of λs. This need not always be so. The expression ((λ x ((λ y ((z x) y))) 3) β reduces to (λ y ((z 3) y)) and the λ is retained after the β reduction. If the result of a β reduction is an abstraction, then the operation of β reduction is said to be a **partial application**.

The η conversion rule is the least intuitive of the lambda calculus rules.

The Rule of η Conversion

Any abstraction (λ v (e v)) may be transformed by η conversion to e if v does not occur free in e.

An example of η conversion is (λ x ((y z) x)) $\rightarrow_\eta$ (y z). The principle of **Functional Extensionality** justifies η conversion**.** This principle says that two functions are identical if they produce the same outputs given the same inputs. That is:-

If for all x, (f x) = (g x) then f = g.

[42] In case there is doubt on this issue, reflect that (λ z (y z)) designates a function that receives an input and applies y to it, and (λ y (y y)) designates a function that receives an input and applies it to itself.

We can argue from this principle in defence of η conversion as follows. η conversion is justified if we can show that, where e is any lambda calculus expression and v is a variable which does not occur free in e, $(\lambda\ v\ (e\ v)) = e$. By the Principle of Functional Extensionality, the proof is secured if it can be proved that, for any a, $((\lambda\ v\ (e\ v))\ a) = (e\ a)$. Performing β reduction on $((\lambda\ v\ (e\ v))\ a)$ gives $(e\ a)$ (since e contains no free occurrences of v, β reduction leaves e unchanged), so the equation is proved.

12.3 The Church-Rosser Theorems

When working with the lambda calculus, the goal is often to reduce a lambda expression to its simplest form. For example, the simplest form of $(((\lambda\ x\ (\lambda\ y\ ((+\ x)\ y)))\ 3)\ 5)$ is $((+\ 5)\ 3)$ (or 8 if the rules of arithmetic are added to the lambda calculus). The expression $((+\ 5)\ 3)$ is a **normal form** of $(((\lambda\ x\ (\lambda\ y\ ((+\ x)\ y)))\ 3)\ 5)$. We say that e is **lambda convertible** to e' when e can be converted to e' by α, β or η conversion rules. In the context of the lambda calculus, a normal form of an expression e is any expression e' where

(a) e is lambda convertible to e' and
(b) at most only α conversion can be applied to change e'.

An important theorem called the **first Church-Rosser theorem** has a bearing on the search for normal forms. Let $e_1 \geq e_2$ obtain when e_1 is reducible to e_2 by zero or more α conversions and/or β reductions. The first Church-Rosser theorem states that if $e_1 \geq e_2$ and $e_1 \geq e_3$ and e_2 and e_3 are not syntactically identical, then there is an e_4 such that $e_2 \geq e_4$ and $e_3 \geq e_4$.

An immediate corollary of the first Church-Rosser theorem is that all normal forms are congruent to each other.

Proof: suppose e_1 reduces to two non-identical normal forms e_2 and e_3, then by the first Church-Rosser theorem there exists an expression e_4 to which e_3 and e_4 are reducible. However, since e_2 and e_3 are normal forms, e_2 and e_3 are convertible to e_4 by α conversion alone. Since $e_2 \approx e_4$ and $e_3 \approx e_4$, then $e_2 \approx e_3$.

This theorem is a basis for the lambda calculus as a foundation for functional programming. If the theorem did not obtain, then two equally acceptable computations of the same lambda expression could terminate with significantly different results. This would make computation with lambda calculus quite hopeless.

A **reduction strategy** for the lambda calculus is a strategy that determines how reductions are performed. Two reduction strategies are **normal order** and **applicative order evaluation**.

The rule of normal order evaluation is that β-redexes must always be reduced from the outside inwards, and where there is a choice, working from the leftmost such redex. The rule of applicative order evaluation is that the leftmost most deeply embedded β-redex must always be reduced first. Another way of expressing this is to say that, given an expression (e1 e2) to evaluate, applicative order evaluation first performs β reduction on e1 and then e2 before attempting β reduction on the whole expression.

For example, reducing ((λ x (x x)) ((λ y y) z)) to a normal form by normal order evaluation involves three steps. For clarity, the chosen β-redexes are underlined.

$$\underline{((\lambda\ x\ (x\ x))\ ((\lambda\ y\ y)\ z))} \rightarrow_\beta (\underline{((\lambda\ y\ y)\ z)}\ ((\lambda\ y\ y)\ z)) \rightarrow_\beta (z\ \underline{((\lambda\ y\ y)\ z)}) \rightarrow_\beta (z\ z)$$

Applicative order evaluation derives the same result in two steps.

$$((\lambda\ x\ (x\ x))\ \underline{((\lambda\ y\ y)\ z)}) \rightarrow_\beta \underline{((\lambda\ x\ (x\ x))\ z)} \rightarrow_\beta (z\ z)$$

When bound variables are repeated, applicative order evaluation is often more efficient than normal order evaluation. However, sometimes, normal order evaluation will find a normal form where applicative order evaluation fails.

For instance ((λ x z) ((λ x (x x)) (λ x (x x)))) has the normal form z, but using applicative order evaluation, this normal form is not computed since ((λ x (x x)) (λ x (x x))) $\rightarrow_\beta$ ((λ x (x x)) (λ x (x x)))! Applicative order evaluation repeats this step endlessly. The **second Church-Rosser theorem** states if a lambda expression has a normal form, then normal order evaluation will find it.

The expression ((λ x (x x)) (λ x (x x))) itself has no normal form. An important theorem of the lambda calculus is that there is no mechanical procedure for accurately determining, in all cases, whether a lambda expression has a normal form. This theorem is a corollary of another important theorem about the lambda calculus that was proved by Church, namely, that it is **Turing-equivalent**. This means that the lambda calculus is powerful enough to serve as a programming language. If there was an algorithm *𝒜* that could determine if the normal order evaluation of any lambda expression terminated, we would also have a decision procedure for finding if *any* program terminated (i.e. translate it into lambda calculus and then apply *𝒜*). However, Turing's proof of the **Unsolvability of the Halting Problem**, states that there is no such procedure to be found.[43]

12.4 Representing Things in Lambda Calculus

It is surprising that a formal system as basic as the pure lambda calculus should have the expressive power of a programming language. The usual control features of conventional programming languages are not in evidence; there are no commands like **if.... then ....else** or **and**. There are no commands like **do ... while** or **for ... until**. There are not even integers or booleans in the lambda calculus.

The basic technique for dealing with this problem is to identify a set of equations that describe the behaviour of the wanted features and then to choose representations in the lambda calculus which reflect those equations. Thus in the case of if …then …else, true and false, we have two essential equations.

(if true then X else Y) = X
(if false then X else Y) = Y

The boolean true can be identified with the lambda expression (λ x (λ y x)) and the boolean false with (λ x (λ y y)). The conditional if....then....else is identified with (λ z (λ x (λ y ((z x) y)))). If these identifications are correct, then the lambda equivalent of (if true then a else b) should evaluate to a. Let us check; we write (if true then a else b) as (((if true) a) b) (figure 12.3).

$(((\text{if true})\ a)\ b) = (((\text{if } (\lambda x\ (\lambda y\ x)))\ a)\ b)$
$= ((((\lambda z\ (\lambda x\ \underline{(\lambda y\ ((z\ x)\ y))}))\ (\lambda x\ (\lambda y\ x)))\ a)\ b)$
$\to_\alpha ((\underline{((\lambda z\ (\lambda x\ (\lambda y'\ ((z\ x)\ y'))))\ (\lambda x\ (\lambda y\ x)))}\ a)\ b)$
$\to_\beta (((\lambda x\ (\lambda y'\ (\underline{((\lambda x\ (\lambda y\ x))\ x)}\ y')))\ a)\ b)$
$\to_\beta (\underline{((\lambda x\ (\lambda y'\ ((\lambda y\ x)\ y')))\ a)}\ b) \to_\beta (\underline{((\lambda x\ (\lambda y'\ x))\ a)}\ b)$
$\to_\beta \underline{((\lambda y'\ a)\ b)} \to_\beta a$

Figure 12.3 Proving that (((if true) a) b) = a in lambda calculus

[43] Turing's result basically shows that there is no algorithm which can determine for any arbitrary computer program (Turing machine) whether it will halt for any arbitrary input. This is one of the most famous proofs in computer science and is intimately related to the proof of the undecidability of first-order logic. See Boolos and Jefferies (1977) for more on this.

It is left as an exercise at the end of this chapter, to show (((if false) a) b) = b also holds.

A basic operation that we need later is forming the pair <x,y> of two elements x and y. We represent this operation by the curried expression ((tuple x) y). Together with this pairing operation are two operations fst and snd that take first and second elements of a pair. These operations must satisfy the equations.

(fst ((tuple X) Y)) = X
(snd ((tuple X) Y)) = Y

We define these concepts as follows. Our $=_{df}$ notation means "is defined as".

tuple $=_{df}$ (λ x (λ y (λ z ((z x) y))))
fst $=_{df}$ (λ x (x true))
snd $=_{df}$ (λ x (x false))

Again, we leave it as an exercise for the reader to verify that these definitions do actually satisfy the two equations. Lambda calculus lacks any natural numbers, but there are many ways of representing them. Each representation treats natural numbers as built up by compositions of the successor function to 0.

i.e. 1 = (succ 0), 2 = (succ (succ 0)),.... etc.

The differences lie in how succ and 0 are defined. We use a definition from **Barendregt** (1984) that defines the natural numbers in terms of the progression.

0 = (λ x x)
1 = ((tuple false) 0)
2 = ((tuple false) 1) …..

We introduce succ as a function that adds 1 to a natural number, a zero test function zero? which, when applied to 0, evaluates to the lambda calculus representation of true. We also introduce the predecessor function pred that subtracts 1. We introduce ⊥ as a constant indicating the error condition of attempting to find the predecessor of 0. The following definitions meet those constraints.

succ $=_{df}$ (λ x ((tuple false) x))
zero? $=_{df}$ (λ x (x true))
pred $=_{df}$ (λ x (((if (zero? x)) ⊥) (snd x)))

From these equations we can prove that (pred (succ x)) = x, (zero? 0) = true etc. The reader is invited to establish these facts.

12.5 Recursion and the Y combinator

The lambda calculus is rich enough to provide surrogates for conditionals, booleans, tuples, natural numbers and the basic arithmetical operations of adding and subtracting 1 from a number. What is still missing is some construction that will do the job of a **do … while** or **repeat .. until** of a conventional procedural programming language. To illustrate how lambda calculus copes with this challenge, we will consider the definition of addition in lambda calculus. Two equations give the properties of addition.

$$0 + Y = Y$$
$$X + Y = (\text{succ } ((\text{pred } X) + Y))$$

To represent these equations in lambda calculus, we use an expression called the **fixpoint combinator** ***Y*** or **Y-combinator** whose behaviour is described by the following equation.

$$(\boldsymbol{Y}\ f) = (f\ (\boldsymbol{Y}\ f))$$

It is possible to define ***Y*** in pure lambda calculus, and there are several ways of doing so; we leave it as an exercise to show that the following definition satisfies $(\boldsymbol{Y}\ f) = (f\ (\boldsymbol{Y}\ f))$.

$$\boldsymbol{Y} = (\lambda\, f\ (V\ V)) \text{ where } V = (\lambda\, x\ (f\ (x\ x)))$$

We can use the Y-combinator to represent addition (here written add) over the natural numbers. We first transcribe the equations for addition into lambda calculus. For clarity we omit currying and use the usual if ... then .. else notation.

$$\text{add} =_{df} (\lambda\, x\ (\lambda\, y\ (\text{if } (\text{zero? } x) \text{ then } y \text{ else } (\text{succ } (\text{add } (\text{pred } x)\ y)))))$$

add mentions itself within its own definition; from our studies we know that this is a recursive definition of add. We could compute additions with this definition if we added a rule that said that we were allowed to replace any occurrence of 'add' in a lambda expression by the right-hand side of this definition. In doing so, we would be enriching the rules of the lambda calculus by allowing replacement according to a definition as an extra rule in the lambda calculus. This is precisely the step taken in the next section, but in this section we want to show that, in principle, the pure lambda calculus can represent addition without needing this extra rule.

In our next step, we replace the occurrence of "add" by a new bound variable. We abbreviate this expression as ADD.

ADD $=_{df}$ (λ f (λ x (λ y (if (zero? x) then y else (succ (f (pred x) y))))))

Now we define + as (***Y*** ADD). This new definition of + will add numbers. As an example, consider the evaluation of ((+ 1) 2) in figure 12.4.

Expression	Justification
((+ 1) 2)	
(((***Y*** ADD) 1) 2)	definition of +
(((ADD (***Y*** ADD)) 1) 2)	definition of ***Y***
(((λ f (λ x (λ y (if (zero? x) then y else (succ (f (pred x) y))) (***Y*** ADD))))) 1) 2)	definition of ADD
((λ x (λ y (if (zero? x) then y else (succ ((***Y*** ADD) (pred x) y))))) 1) 2)	β reduction
((λ y (if (zero? 1) then y else (succ ((***Y*** ADD) (pred 1) y))) 2)	β reduction
(if (zero? 1) then 2 else (succ ((***Y*** ADD) (pred 1) 2)))	β reduction
(succ (((***Y*** ADD) 0) 2))	(zero? 1) is false
(succ (((ADD (***Y*** ADD)) 0) 2))	definition of ***Y***
(succ (((λ f (λ x (λ y (if (zero? x) then y else (succ (f (pred x) y))) (***Y*** ADD))))) 0) 2))	definition of ADD
(succ ((λ x (λ y (if (zero? x) then y else (succ ((***Y*** ADD) (pred x) y))))) 0) 2))	β reduction
(succ ((λ y (if (zero? 0) then y else (succ ((***Y*** ADD) (pred 0) y))) 2))	β reduction
(succ (if (zero? 0) then 2 else (succ ((***Y*** ADD) (pred 0) 2))))	β reduction
(succ 2)	(zero? 0) is true

Figure 12.4 Evaluation using Y-combinators

12.6 From Lambda Calculus to Functional Programming

The preceding section showed that it is possible to program in the pure lambda calculus by using the definitions to rewrite numbers etc. as lambda expressions. Programming at this level of detail is not attractive; for example the number 2 in our pure lambda calculus is:-

$$((\lambda\ x\ (\lambda\ y\ (\lambda\ z\ (((z\ x)\ y))))\ (\lambda\ x\ (\lambda\ y\ y))\ (\lambda\ x\ (\lambda\ y\ (\lambda\ z\ (((z\ x)\ y))))\ (\lambda\ x\ (\lambda\ y\ y)))$$

Practical programming with the lambda calculus requires a **barrier of abstraction** between the programmer and the complexity of these expressions. A barrier of abstraction enables the programmer to work these representations, without having to hold in mind all the details of his representation. A very simple mechanism provides the answer to this problem. We allow lambda calculus expressions to be *named*, so that instead of writing (λ x (λ y x)) we can write **true**.

Adopting this mechanism requires us to adopt a convention for treating names. We introduce names for lambda calculus expressions by $=_{df}$ (meaning "is defined as"), so (true $=_{df}$ (λ x (λ y x))). A definition of this kind takes the form of some constant to the left-hand side of $=_{df}$ and a expression to right-hand side of $=_{df}$.

This convention allows lambda expressions to be named, but we maintain the proviso that the same constant cannot be defined in two different ways. This requires a record (called an **environment**) of such definitions to be maintained. Formally we identify an environment with a set Σ of pairs <c, e> where c is a constant and e is its lambda calculus definition. Σ has the **uniqueness property** that no constant is defined in two different ways; i.e.

$$\forall c \forall e \forall e^*(<c, e> \in \Sigma \ \&\ <c, e^*> \in \Sigma) \rightarrow e = e^*$$

In our extension of the lambda calculus, evaluation operates not on a lambda expression alone, but on a pair <Σ, e> where Σ is an environment and e is a lambda expression. We write <Σ, e> $\Rightarrow$ <Σ^*, e*> to signify that <Σ, e> can be evaluated to <Σ^*, e*>. There are three rules for evaluation in our system (figure 12.5)

1. <Σ, e> $\Rightarrow$ <Σ, e*> if e is lambda convertible to e*
2. <Σ, e> $\Rightarrow$ <Σ, e*> if <e, e*> $\in \Sigma$
3. <Σ, (c $=_{df}$ e)> $\Rightarrow$ <$\Sigma^* \cup$ {<c, e>}, c>, where $\Sigma^* = \Sigma$ - {x | <c, x>}

Figure 12.5 Rules for evaluation in the context of an environment

Rule 1 states that a lambda expression can be evaluated to another lambda expression e* in the context of an environment provided e is lambda reducible to e*. The second rule says that e may be evaluated to e* if e is defined as e* in S. The final rule says that given a definition (c $=_{df}$ e) and an environment Σ, that the effect of evaluating that definition is to add the definition of c to Σ; **overwriting** any old definition of c in the environment.[44]

The variable capture problem arises in rule 2; let <c, (λ x (x y))> ∈ Σ. Unfolding within (λ y (c y)) gives (λ y (λ x (x y))) in which the free y in (λ x (x y)) becomes bound. It is possible, with respect to unfolding, to adopt the same sort of restrictions that were applied to α and β conversion and insist the c cannot be unfolded to e if a free variable y in e becomes captured as a result. Nevertheless, a simpler and more efficient computational model is obtained if we allow only closed terms in Σ.

λ+ is a simple imaginary functional programming language based directly on the rules in figure 12.5. On first entering the read-evaluate-print loop, the environment Σ in λ+ is empty. So initially, λ+ is a lambda calculus evaluator driven by a reduction strategy which we assume to be normal order evaluation. Here is an imaginary interaction with λ+.

λ+; x
⇒ **x**

The prompt λ+; shows that the read-evaluate-print loop is running. Typing in **x** and then carriage return causes the λ+ interpreter to return the normal form of **x**. The **x** evaluates to itself; such an expression is a self-evaluating expression. Here are some more inputs.

[44]This rule raises an intriguing possibility; what if we choose to evaluate a lambda expression which contains a definition; for example (λ x (c x)) (c $=_{df}$ (λ x (λ y (x y)))) ? In this case, the evaluation of the lambda expression will change any definition of c already in the environment. If we think of the program driving the evaluation as residing in the definitions of the constants within Σ, then we have the prospect of allowing **self-modifying** programs. A program is self-modifying if running that program can cause the program itself to change. The original functional programming language, Lisp, did allow self-modifying programs and this feature is a fascinating and under-exploited feature of the language. More recent arrivals to the functional scene have avoided self-modifying programs. SML for instance avoids self-modifying programs. The case against self-modifying programs is that it is very difficult to debug or prove the correctness of a program that changes during its own execution. Self-modification is almost an inevitable feature of having the capacity to perform metaprogramming.

λ+; **((λ x x) (λ x x))**
⇒ **(λ x x)**

λ+; **(λ x (x x))** (λ **x (x x)))**

The last expression has no normal form so λ+ enters an infinite loop. Starting again, we introduce truth values and boolean operations.

λ+; **(true** $=_{df}$ **(λ x (λ y x)))**
⇒ **(λ x (λ y x))**

λ+; **(false** $=_{df}$ **(λ x (λ y y)))**
⇒ **(λ x (λ y y))**

λ+; **(if** $=_{df}$ **(λ z (λ x (λ y ((z x) y)))))**
⇒ **(λ z (λ x (λ y ((z x) y))))**

λ+; **(((if true) false) true)**
⇒ (λ **x (λ y y))**

λ+ is not a practical functional programming language. Practical functional programming languages do not represent numbers as lambda expressions, but make use of arithmetic operations that come with the instruction set in the resident hardware, treating numbers, strings and booleans as primitive objects. Like Qi, functional programming languages also facilitate the definition of functions by not requiring function applications to be curried. With these changes λ+ could serve as a practical programming language. But here we leave lambda calculus for the moment, where it merges into functional programming, to return to the compilation of Qi.

12.7 The Compilation of Qi to $\mathscr{L}$

The language $\mathscr{L}$ is an extension of the lambda calculus designed to support the compilation and type checking of pattern-directed function definitions. It contains one significant addition to lambda calculus, which is the appearance of pattern-matching abstractions. We shall explain how Qi is compiled to $\mathscr{L}$ by reference to the compilation of the Qi function in figure 12.6.

```
(define member
   _ [ ] -> false
  X [X | _] -> true
  X [_ | Y] -> (member X Y))
```

Figure 12.6 The membership function in Qi

To begin compilation to $\mathcal{L}$, all lists are translated to cons form, and all wildcards are replaced by unique variables (figure 12.7).

```
(define member
  V [] -> false
  X (cons X W) -> true
  X (cons Z Y) -> (member X Y))
```

Figure 12.7 Eliminating Wild Cards and Converting to cons form

The further step is to ensure that each rule is **left linear**, where a left linear rule is one that does not contain the same variable twice to the left of the arrow. The second rule of member is not left linear, so we make it so by renaming the repeated occurrences of the variable and by placing a guard on the rule (figure 12.8).

```
(define member
  V [] -> false
  X (cons U W) -> true  where (= U X)
  X (cons Z Y) -> (member X Y))
```

Figure 12.8 Eliminating non-left linearity

The next step is to determine the arity *n* of the function by counting the number *n* of patterns to the left of each arrow in each rule. If there is a different number in any two rules, then an error is raised. If not, then the arity *n* is used to build an abstraction. The function member is defined to mean the same as this abstraction and we use the $=_{df}$ notation to signify this (figure 12.9).

```
member =df (λ A (λ B (V [ ] -> false
             X (cons U W) -> true          where (= U X)
             X (cons Z Y) -> (member X Y))))
```

Figure 12.9 Building an abstraction

The next step is to replace each individual rule by an expression that uses a **pattern-matching lambda abstraction**. A conventional abstraction is part of lambda calculus, but pattern-matching abstractions are not part of lambda calculus; they are, however, part of $\mathscr{L}$.

The syntactic difference between a conventional abstraction and a pattern-matching abstraction is that a conventional abstraction may contain only a variable after the λ, whereas a pattern-matching abstraction may have a pattern. For our purposes, a pattern is either a base expression (i.e. a symbol, string, number, boolean or character) or the empty list, a tuple of patterns or a cons expression built of patterns. Thus the expression, (λ x x) is a conventional lambda abstraction, but the expression (λ (cons 1 [ ]) 2) is a pattern-matching abstraction. Here are some more examples with their attendant interpretations.

Expression	Interpretation
(λ (cons 1 [ ]) 2)	A function that if it receives an input (cons 1 [ ]) returns 2.
(λ 1 2)	A function that if it receives an input 1 returns 2.
(λ [ ] [ ])	A function that if it receives the input [ ] returns [ ].
(λ (cons x [ ]) x)	A function that if it receives a one-element list returns its first element.

What happens if (λ (cons 1 [ ]) 2) is applied to, say, the number 3? Intuitively the input 3 does not match the pattern (cons 1 [ ]) and so the application fails. In such a case, we designate the object ⊗ as being the result of this application - indicating a match failure. Now consider the first rule of the member function

V [ ] -> false

Using a pattern-matching abstraction, this becomes **(λ V (λ [ ] false))**

The second rule is **X (cons U W) -> true where (= U X)**

which becomes **(λ X (λ (cons U W) (where (= U X) true))**

and the final rule **X (cons Z Y) -> (member X Y)** becomes

(λ X (λ (cons Z Y) (member X Y)))

Each rule is now translated into a pattern-matching abstraction; it remains to link these abstractions together by (a) applying each abstraction to the relevant formal parameters of the original function and (b) placing them in an ordering that indicates the order in which they are used.

For example the first abstraction was **(λ V (λ [] false))** which is designed to be applied to the variables A and B of the abstraction in figure 12.9. Doing this results in the expression **(((λ V (λ [] false)) A) B)**. Repeating this for both of the other two pattern-matching abstractions gives:-

```
1.  (((λ V (λ [ ] false)) A) B)
2.  (((λ X (λ (cons U W) (where (= U X)  true)) A) B)
3.  (((λ X (λ (cons Z Y)  ((M X) Y))) A) B)
```

Now part (b) is performed; we combine these expressions into a single case expression whose ordering shows the order in which each expression is evaluated.

```
(cases  (((λ V (λ [ ] false)) A) B)
        (((λ X (λ (cons U W) (where (= U X) true)) A) B)
        (((λ X (λ (cons Z Y)  ((M X) Y))) A) B))
```

The semantics of **cases** is to evaluate each expression in its scope in turn, and to return the result of the first evaluation that does not return ⊗. If all evaluations return ⊗, then ⊗ is returned as an error message, indicating a match failure.

It only remains to replace the rules in 12.9 by this case expression and insert the Y-combinator to derive an expression in $\mathcal{L}$ (figure 12.10).

```
(Y  (λ M (λ A (λ B (cases (((λ V (λ [ ] false)) A) B)
                (((λ X (λ (cons U W) (where (= U X) true)) A) B)
                 (((λ X (λ (cons Z Y) ((M X) Y))) A) B))))))
```

Figure 12.10 The member function reconstructed in $\mathcal{L}$

In the next chapter, we shall see that it is $\mathcal{L}$ expressions that are actually typechecked. We have not so far mentioned function definitions that use backtracking; for the purposes of type checking, these are compiled out during the compilation to $\mathcal{L}$. Thus the construction $p_1 \ldots p_n$ **<-** (fail-if *f x*) (where $p_1,\ldots,p_n$ are patterns) is compiled into (λ p_1 …(λ p_n (where (not (*f x*)) *x*))…).[45]

12.8 The Compilation of $\mathcal{L}$ to Lambda Calculus

The compilation of $\mathcal{L}$ to lambda calculus involves eliminating the abstractions generated from compiling patterns. These abstractions have one of three forms;

(a) they are of the form (λ c x) where c is a base expression
(b) they are of the form (λ (cons x y) z) or (λ (@p x y) z)
(c) they are conventional abstractions of the lambda calculus.

The first form can be eliminated by the equivalence:

E1. (λ c x) = (λ y (where (= y c) x))

The second form can be eliminated in the context of the applications where it occurs by repeatedly applying the equivalence.

E2a. ((λ (cons x y) z) w)
= ((λ x ((λ y (where (cons? w) z**) (tail w)) (head w))
where z** = $[z^*]_{(\text{tail w})/y}$ and z* = $z_{(\text{head w})/x}$
E2b. ((λ (@p x y) z) w) = ((λ x ((λ y (where (tuple? w) z**) (snd w)) (fst w))
where z** = $[z^*]_{(\text{snd w})/y}$ and z* = $z_{(\text{fst w})/x}$

The final form is eliminated straightforwardly by β reduction

E3. ((λ x y) z) = $[x]_{z/y}$

To illustrate on the expression **(((λ V (λ [ ] false)) A) B)**. The innermost application is **((λ V (λ [ ] false)) A)**. The outer abstraction is a simple abstraction of the lambda calculus, so E3 is applied leaving **(λ [ ] false)**. In the context of the entire expression, the result is **((λ [ ] false) B)**. This is simplified by E1 to **(where (= B [ ]) false)**.

The next expression is **(((λ X (λ (cons U W) (where (= U X) true)) A) B)** in which the innermost application is **((λ X (λ (cons U W) (where (= U X) true)) A)**. The outer

[45] This serves for the purposes of typechecking, although the expression ***X*** is repeated which , from an operational point of view, is not efficient. The Qi compiler locally binds the x in the subsequent reduction of $\mathcal{L}$ to Lisp.

abstraction is a simple abstraction of the lambda calculus so E3 is applied leaving **(λ (cons U W) (where (= U A) true))**.

Putting this expression into its original context, we get **((λ (cons U W) (where (= U A) true)) B).** Applying E2. gives us **((λ U ((λ W (where (cons? B) z**) (tail B)) (head B))** where **z**** is defined as

$$z^{**} = [z^*]_{(tail\ B)/W} \text{ and } z^* = [(where\ (=\ U\ A)\ true)]_{(head\ B)/U}$$

i.e. z** = **(where (= (head B) A) true).** Substituting according to this definition we derive

((λ U ((λ W (where (cons? B) (where (= (head B) A) true)) (tail B)) (head B))

Two applications of E3. then derive

(where (cons? B) (where (= (head B) A) true)).

The final case was **(((λ X (λ(cons Z Y) ((M X) Y)) A) B)**. Reducing from the outside this time, the outer expression is a simple abstraction, so that E3. delivers **((λ (cons Z Y) ((M X) Y)) B)**. Applying E2 and then E3 twice gives **(where (cons? B) ((M A) (tail B)))**. Combining these results within the original case statement gives the expression below.

```
(Y (λ M (λ A (λ B (cases (where (= B []) false)
          (where (cons? B) (where (= (head B) A) true))
              (where (cons? B) ((M A) (tail B))))))))
```

Factoring out the **where**s by the equivalence

E4. (**where** P (**where** Q R)) = (**where** (and P Q) R)

derives

```
(Y (λ M (λ A (λ B
  (cases (where (= B []) false)
       (where (and (cons? B) (= (head B) A)) true)
       (where (cons? B) ((M A) (tail B))))))))
```

The case statement is now compiled into a nested if...then...else statement according to the equivalence.

E5. (**cases** (**where** P Q) ….) = (if P Q (cases …..))
E6. (**cases** P …) = P if P is an unguarded expression (one not prefaced by **where**)
E7. (**cases**) = ⊥ (⊥ is the error condition)

The resulting expression appears below.

```
(Y (λ M (λ A (λ B (if (= B []) false
                 (if (and (cons? B) (= (head B) A))
                          true
                   (if (cons? B) ((M A) (tail B))
                           ⊥))))))))
```

In actual application Qi compiles into a labelled lambda calculus with functions being explicitly named. The functional programming language Lisp is used as a close approximation of a labelled lambda calculus.[46]

Exercise 12

1. Which variable occurrences are free and which are bound in the following expressions?
   ((y x) y), ((λ y (y x)) y), (x x), (y (λ y (x (y z))))
2. Perform the necessary substitutions on the following.
   $[(y\ x)\ y]_{z/y}$, $[(y\ x)\ y]_{z/x}$, $[(\lambda\ y\ ((y\ x)\ y))]_{z/y}$, $[(x\ x)]_{z/x}$, $[(y\ (\lambda\ y\ (x\ (y\ z))))]_{z/y}$
3. Reduce the following to a normal form using α, β and η conversion.
   ((y x) y), ((λ x (λ y (x x))) y), ((λ x (λ y (x y))) (λ x (x y)))
4. Show the following equations hold using the definitions provided in this chapter.
   (((if false) x) y) = y,
   (fst ((tuple x) y)) = x,
   ((snd ((tuple x) y)) = y,
   (pred (succ x)) = x,
   (zero? 0) = true,
   (*Y* f) = (f (*Y* f)) where *Y* = (λ f (V V)) and V = (λ x (f (x x)))
5. Hand compile the join and reverse functions of chapter 4 into *ℒ* and then into lambda calculus.
6. *Implement an interpreter for the pure lambda calculus that reduces lambda expressions to a normal form using normal order evaluation. You will define lambda expressions as a recursive type.
7. *Implement an interpreter for the pure lambda calculus that reduces lambda expressions to a normal form using applicative order evaluation. You will define lambda expressions as a recursive type.
8. Using your answer to 6., implement λ+. Your program will define an environment as a list of pairs composed of a name and a lambda expression.

[46] For more on the relations between Qi and Lisp, see appendix B.

9. *Define the expressions of $\mathcal{L}$ as a recursive type and build a compiler from $\mathcal{L}$ into lambda calculus.
10. *Implement a compiler from Qi into $\mathcal{L}$.

Further Reading

Gordon (1988) (chapters 4-7) gives a nice introduction to the lambda calculus. For the encyclopaedic accounts, **Barendregt** (1984) and **Hindley** and **Seldin** (1986), but **Barendregt** (1992) (pp 118-147) for a quick look. The compilation of pattern-matching code into extended abstractions and then into lambda calculus is discussed in **Peyton-Jones** (1988). The generation of lambda notation represents the end of this chapter, but in the complete compilation of a functional programming language, the production of this sort of code is only an interim in the compilation of a functional program into executable object code. For accounts of various strategies for compiling and interpreting lambda expressions see **Landin** (1964), **Diller** (1988), **Turner** (1978) and **Peyton-Jones** (1987).

Web Sites

(http://www.santafe.edu/~walter/AlChemy/software.html) provides a standalone lambda calculus interpreter written in C. **Colin Taylor** provides a version (http://drcjt.freeyellow.com), including source code) which will run on a Palm Pilot.

13 Type Checking as Proof

13.1 The Qi Type System

This chapter presents Qi's type-checker as a proof procedure $\mathcal{T}$ for proving the types of expressions in $\mathcal{L}$. We begin at the top level of the type-checker, which is a loop used to load and typecheck a Qi source file F. The top level is very simple; Qi compiles the expressions of F into a list *l* of $\mathcal{L}$ expressions. Qi then cycles through the elements of *l*, type checking each element *e* by a type checking procedure $\mathcal{T}$ which attempts to prove that *e* is well-typed.

The application of $\mathcal{T}$ to *e* will give rise to one of three cases.

1. $\mathcal{T}$ may terminate, having failed to prove that *e* is well typed.[47]
2. $\mathcal{T}$ may terminate, having proved that *e* has a type. This result will be added to the type environment and *e* will be compiled into the Qi image and taken out of *l*.
3. $\mathcal{T}$ may fail to terminate.[48]

[47] If expressions are entered to the top level, then a type error will be raised when such a proof cannot be found.

[48] $\mathcal{T}$ is guaranteed to terminate in the absence of any user-defined types. There are type rules that can cause infinite loops (see the final section of this chapter). In practice Qi sets a limit on the number of inferences employed in any proof (1,000,000 is the default). Hence termination always occurs in time. However, this maximum can be reset by **maxinferences**, so that it is possible for Qi to fail to terminate typechecking within any reasonable period of time (such as a year or two!).

At the end of a successful load, */* will be reduced to the empty list. If the load is unsuccessful, either the load will not terminate, or it will terminate and */* will not be empty. In the last case, Qi will print error messages detailing where the proofs failed. To define $\mathcal{T}$, we need to state what rules are employed by $\mathcal{T}$ and what proof procedure $\mathcal{T}$ uses. We shall explore the Qi type system by examining the rules that $\mathcal{T}$ uses.

13.2 Type Checking Applications

Our first set of rules (figure 13.1) enables applications to be typechecked.

Sequents	*Application*	*Primitive*
$\dfrac{}{X : A >> X : A;}$	$\dfrac{F : (A \rightarrow B); X : A;}{(F\ X) : B;}$	$\dfrac{\text{if } X \text{ is base and } A \in \tau(X)}{X : A;}$

Figure 13.1 The rules for typechecking applications

The *Sequents Rule* is obvious. The *Rule of Application* states that an application (F X) has the type B provided F has the type A → B for some A and X has the type A. The *Primitive Rule* assumes that there is a function τ from the set of all base expressions of Qi (i.e. numbers, symbols, function symbols and the rest) into the powerset of all types such that A ∈ τ(*e*) when A is a type of *e*; thus τ(1) = {number}, τ("1") = {string}, τ(+) = {number → (number → number), symbol}. The *Primitive Rule* states that given a typing *e* : A to prove, we may prove it if *e* is a base object where A ∈ τ(*e*).

From these rules, we can establish the types of applications. Here is an e example; the goal is to prove that (* 1 0) : number. To adopt the curried form required by the *Rule of Application* we write (* 1 0) as ((* 1) 0).

Proof: by the *Rule of Application* ((* 1) 0) : number is proved if there is a type A such that (* 1) : A → number and 0 : A. Let A be the type number. Then we need to prove (* 1) : number → number and 0 : number. 0 : number follows by the *Primitive Rule*. By the *Rule of Application* (* 1) : number if there is some type B such that * : B → (number → number) and 1 : B. Let B be the type number; * : number → (number → number) and 1 : number are proved by the *Primitive Rule.*

13.3 Choosing Values by Unification

The preceding proof could be easily mechanised on a computer except for the problem of choosing values for variables. Consider the case in the preceding proof where it was necessary to decide a value for 'B' such that '* : B → (number → number)' was solvable. The *Primitive Rule* can show that * : number → (number → number). The problem is to match '* : number → (number → number)' to the conclusion '* : B → (number → number)'. These two typings match completely except where 'B' is matched to 'number'.

In order to agree that the two typings match, an extended notion of "match" must be used and **unification** provides this notion. Two expressions **unify** when there is a uniform substitution for the variables in each expression that makes the two expressions the same. 'B' unifies with 'number' because the association B |→ number makes the two identical. A set of associations that unifies two expressions X and Y is a **unifier** of X and Y. In the case of 'B' and 'number', the unifier is the single substitution of 'number' for 'B'. 'number' is said to be the **value** of 'B' under this unification. The elements of a unifier are called **bindings**, and process of substituting the variables in an expression by their values under a unifier is called **dereferencing**.

For example, under the association A |→ boolean, the expression '(list (list A))' dereferences to '(list (list boolean))'. Under the associations {A |→ (list B), B |→ number} the expression '(list (list A))' dereferences to '(list (list (list number)))'. Conventionally if σ is a unifier of two expressions then the result of dereferencing an expression *E* by σ is written σ(*E*). The **most general unifier** or **MGU** of two expressions is the *smallest* unifier needed to unify the two. For example, A → B unifies with number → B via the unifier {A |→ number, B |→ number}. However this is not the MGU of the two expressions which is {A |→ number}.

Unification is used within type checking to perform the needed substitutions. The typings * : B → (number → number) and * : number → (number → number) are unified to find their MGU σ and the instantiation is given by σ(B). Here σ(B) = number.

Finding the MGU of two expressions is a decidable problem and the most commonly used method for finding it was suggested by Robinson (1965). Robinson's algorithm can be specified in a set of recursive equations for a function *mgu* that returns $\perp$ if its inputs do not unify and a set of bindings otherwise. The encoding in Qi is quite straightforward.[49]

1. *mgu*(x, y) = *mgu'*(x, y, {})
2. *mgu'*(x, x, σ) = σ
3. *mgu'*(x, y, σ) = {x $\mapsto$ y} $\cup$ σ if x is a variable and x does not occur in y
4. *mgu'*(x, y, σ) = {y $\mapsto$ x} $\cup$ σ if y is a variable and y does not occur in x
5. *mgu'*((x_1,..x_n), (y_1,..y_n), σ) = $\perp$ if *mgu'*(x_1, y_1, σ) = $\perp$
   = *mgu'* (*deref*((x_2,...,x_n), σ'), *deref*((y_2,..,.y_n), σ') ,σ') otherwise, where σ'= *mgu'*(x_1,y_1, σ)
6. *mgu'*(x, y, σ) = $\perp$ in all cases not covered by 2-5.
7. *deref*(x, σ) = *deref*(y, σ) if x $\mapsto$ y $\in$ σ
8. *deref*((x_1,…,x_n), σ) = ((*deref*(x_1, σ)),..., *deref*(x_n, σ))
9. *deref*(x, σ) = x in all cases not covered by 7 or 8.

Figure 13.2 Computing the MGU by a Set of Recursive Equations

Figure 13.2 gives some specimen inputs to the unification function and the outputs. Notice that in case 6 the unification fails since a variable cannot be associated with an expression containing that variable (this is called an **occurs-check failure**).

Expression 1	Expression 2	MGU
a	a	{}
X	b	{X $\mapsto$ b}
Y	X	{X $\mapsto$ Y} or {Y $\mapsto$ X}
(f Y)	X	{X $\mapsto$ (f Y)}
a	b	$\perp$
X	(f X)	$\perp$
(f (f X Y) Y)	(f (f a b) b)	{X $\mapsto$ a, Y $\mapsto$ b}

In Qi, all the rules of its type system (including those entered by the user) are interpreted *modulo* unification with respect to types. We shall adopt this approach in all the subsequent proofs in this chapter.

[49] The encoding in Qi is set as an exercise at the end of this chapter.

13.4 Type Checking Abstractions

The *Rule of Abstraction* says that, where X is a variable, (λ V X) has the type A → B if, where c is arbitrary object of type A, $[X]_{c/V}$ (the result of replacing all free occurrences of V by c throughout X) is of the type B (figure 13.3).

Abstraction

$$\frac{\textit{where } c \textit{ is arbitrary} \quad c : A \gg [X]_{c/V} : B;}{(\lambda\ V\ X) : (A \rightarrow B);}$$

Figure 13.3 The rule for typechecking abstractions

The meaning of "arbitrary" is a little subtle. In conventional usage to select something arbitrarily is to select it at random. This is not the sense of "arbitrary" needed here. For instance, one could "prove" that all numbers are odd by arbitrarily selecting 3 and stating that since 3 is odd and selected arbitrarily, all numbers are odd. This is obviously wrong.

The problem then arises of how we show that our choice of object is genuinely arbitrary, or arbitrary in sense that allows us to claim that the object chosen stands proxy for all cases. The philosopher's answer is that the object chosen must have no special conditions attached to it. This is correct, but changes the problem into - how are we supposed to recognise that no special assumptions are attached? The logician supplies a syntactic criterion; the object supplied should be given a unique name, which does not occur anywhere else in the proof; such a name is **fresh**. This shows up immediately what is wrong with the 'proof' that all numbers are odd. We begin with the sequent

odd(3) >> ∀x odd(x)

We select our so-called arbitrary case, 3.

odd(3) >> odd(3)

However, here the proof fails, since "3" is not a fresh symbol and so is not arbitrary in the required sense. Here is a proof of (λ x ((* 3) x)) : (list number) → (list number), demonstrating the use of the *Abstraction* and *Sequents Rules*

Proof: by the *Abstraction Rule*, (λ x ((* 3) x)) : (list number) → (list number) is provable if *a* : number >> ((* 3) *a*) : number is provable (*a* is our fresh symbol). By the *Rule of Application*, *a* : number >> ((* 3) *a*) : number is provable if both

a : number >> (* 3) : A → number
a : number >> *a* : A

are provable for some type A. The second sequent is solved using the *Sequents Rule* with unifier {A |→ number}. The first sequent now becomes

a : number >> (* 3) : number → number.

By the *Rule of Application* again, this splits into two subproblems

a : (list number) >> * : (B → (number → number).
a : (list number) >> 3 : B

Both these problems are solved by the *Primitive Rule* using the unifier {B |→ number}.

There is one extra wrinkle to using unification in type checking, which is worth mentioning here, and it is called **standardising apart**. Consider the problem of showing (λ x x) has a type; this can be rephrased as "Find the (most general) value for A such that (λ x x) : A." However if we try to unify (λ x x) : A with the conclusion (λ V X) : (A → B) of the Abstraction Rule then an occurs-check failure arises (A cannot be unified with (A → B)). The problem arises from the accidental use of A in both rule and goal. To avoid this, we uniformly rename or standardise apart the variables of any rule before applying it, so that all variables in the rule are fresh.

13.5 Polymorphic Functions

When we say that the polymorphic function (λ x x) has the type A → A, this is a shorthand for asserting that for *all* values of A, (λ x x) inhabits A → A, or in logical notation (∀A (A → A)). The *Rule of Generalisation* enables proofs of the types of polymorphic functions. To prove that X : (∀A B), we must prove that, where C is any arbitrary type, X : $B_{C/A}$ (where $B_{C/A}$ is the result of substituting all occurrences of A by C). The concept of arbitrary is defined in the same way as previously - by the introduction of a fresh symbol.

Generalisation *where* C *is fresh*

$\underline{X : B_{C/A};}$
X : (∀A B);

Specialisation

$\underline{X : B_{C/A}, X : (\forall A\ B)\ >>\ P;}$
X : (∀A B) >> P;

Figure 13.4 The rules for typechecking with polytypes

Here is a proof that (λ x x) : (∀A (A → A)).

Proof: Applying the *Rule of Generalisation* to the problem derives (λ x x) : *a* → *a*, (*a* is our fresh symbol). Applying the *Rule of Abstraction* to (λ x x) : *a* → *a* derives *b* : *a* >> *b* : *a* (*b* is our fresh symbol) which is solved by the *Sequents Rule.*

The Rule of Specialisation is obvious; if we have shown that X : (∀A B), then we can conclude that X : $B_{C/A}$ for any type C we care to choose. Here is a proof of that [ ] : (∀A (list A)), remove : (∀A (A → ((list A) → (list A))) >> ((remove 1) [ ]) : (list number).

Proof: By the *Rule of Specialisation* applied twice, we derive

[ ] : ∀A (list A), [ ] : (list number),
remove : (number → ((list number) → (list number))),
remove : ∀A(A → ((list A) → (list A))) >> ((remove 1) [ ]) : (list number).

Let Δ to be the assumptions in this sequent. We have to prove Δ >> ((remove 1) [ ]) : (list number).

By the *Rule of Applications* we have 2 sequents to prove; Δ >> (remove 1) : (B → (list number)) and [ ] : B. Applying the *Rule of Applications* to the first sequent we derive a total of three sequents;

1. Δ >> remove : (C → (B → (list number)))
2. Δ >> 1 : C.
3. Δ >> [ ] : B

The first is solved by unification using the *Sequents Rule* ({C |→ number, B |→ (list number}). The second (1 : C, with C |→ number) is solved by *Primitive* and [ ] : B is solved by the *Sequents Rule* with B |→ (list number). In the implementation of Qi, the *Specialisation Rule* is not used, since its purpose is only to allow us to specialise the ∀-bound symbols. Instead, unification is built into the system so that the appropriate specialisation is made automatically.

Finally, the error objects ⊥ and ⊗ have the polymorphic type (∀ A A).

$$\frac{}{\bot : (\forall\ A\ A)} \qquad \frac{}{\otimes : (\forall\ A\ A)}$$

Figure 13.5 The rules for error objects

13.6 Typing Special Forms

Certain expressions of Qi are special forms - they cannot be curried and they have their own typing rules.[50]

Cons Rule (left)

$$\frac{\text{X : A, Y : (list A) >> P;}}{\text{(}\textbf{cons}\text{ X Y) : (list A) >> P;}}$$

Cons Rule (right)

$$\frac{\text{X : A; Y : (list A);}}{\text{(}\textbf{cons}\text{ X Y) : (list A)}}$$

@p Rule (left)

$$\frac{\text{X : A, Y : B >> P;}}{\text{(}\textbf{@p}\text{ X Y) : (A * B) >> P;}}$$

@p Rule (right)

$$\frac{\text{X : A; Y : B;}}{\text{(}\textbf{@p}\text{ X Y) : (A * B)}}$$

Equality Rule

$$\frac{\text{X : A; Y : A;}}{\text{(}\textbf{=}\text{ X Y) : boolean}}$$

Conditional Rule

$$\frac{\text{X : boolean; Y : A; Z : A;}}{\text{(}\textbf{if}\text{ X Y Z) : A;}}$$

Local Rule

where c *is fresh*

$$\frac{\text{Y : B;} \quad \text{c : B >> } [\text{Z}]_{\text{c/X}} \text{ : A;}}{\text{(}\textbf{let}\text{ X Y Z) : A;}}$$

Figure 13.6 The rules for special forms

Here is a proof that (let X 6 (if (= X 5) 0 1)) : number. By the *Local Rule*, (let X 6 (if (= X 5) 0 1)) : number if both 6 : A and *x* : A >> (if (= *x* 5) 0 1) : number. The first problem is solved by the *Primitive Rule* with A |→ number. The second problem dereferences to

[50] Other expressions which have their own typing rules are do, error, input+ and output. However these are relatively unimportant and for reasons of space are not discussed here. Their representation in Qi can be viewed in the file **type theory 3.0.qi**, which is loaded during installation and which contains the type rules for Qi.

x : number >> (if (= x 5) 0 1) : number

By the *Conditional Rule*, this problem is solvable if the following are solvable.

1. x : number >> (= x 5) : boolean;
2. x : number >> 0 : number
3. x : number >> 1 : number

2. and 3. are solvable by the *Primitive Rule* and 1. decomposes by the *Equality Rule* to

4. x : number >> x : B
5. x : number >> 5 : B

4. is solved by the *Sequents Rule* with B |→ number and 5. by the *Primitive Rule*.

13.7 Internal Forms

The rules in this section deal with constructions that are generated from Qi definitions. They are referred to as **internal forms**. Internal forms are expressions that are generated internally through the Qi compiler and are compiled out by the procedures of the previous chapter before the Lisp object code is generated. The first such rule deals with the ***Y*** combinator.

Combinator Rule
where c *is fresh*

$$\frac{c : A >> [Y]_{c/X} : A}{(\boldsymbol{Y}\ (\lambda\ X\ Y)) : A}$$

Figure 13.7 The rule for typechecking recursive functions

In chapter 12 we saw that that the combinator ***Y*** could be used to define recursive functions. The combinator rule allows us to typecheck the types of such functions. For example we define factorial as (λ x (if (= x 0) 1) ((* x) (factorial ((- x) 1))))). Using the ***Y*** combinator this becomes (***Y*** (λ y (λ x (if (= x 0) 1 ((* x) (y ((- x) 1))))))). We prove that this expression has the type number → number.

Proof Sketch: ($\boldsymbol{Y}$ (λ y (λ x (if (= x 0) 1 ((* x) (y ((- x) 1))))))) : number → number.

By the *Combinator Rule*, this is provable if the following sequent is provable

f : number → number >> (λ x (if (= x 0) 1 ((* x) (*f* ((- x) 1))))) : number → number.

The reader can complete the proof.

The next two rules deal with case statements and guards.

Cases Rule
For each *i*, $1 \leq i \leq n$
$\underline{\text{Case}_i : \text{A;}}$
(**cases** $\text{Case}_1 \ldots \text{Case}_n$) : A

Guard Rule
Guard : boolean;
$\underline{\text{Guard : verified >> X : A;}}$
(**where** Guard X) : A

Figure 13.8 The rules for typechecking cases and guards

The first rule says that the body of a function definition (consisting of a series of cases) has the type A if every case in the body of the function has the type A. The second rule applies to guards and says that a guarded expression X has the type A if the guard has the type boolean and X has the type A under the assumption that the guard is verified. These rules are fairly straightforward.

The Patterns Rule is used to typecheck extended abstractions. We assume that $V_1 \ldots V_n$ are all the variables occurring in P, and $\phi_1, \ldots, \phi_n$ are fresh symbols, and $C_1, \ldots C_n$ are fresh variables. For any κ, the expression $[\kappa]_{<\phi_1 \ldots \phi_n>/<V_1,\ldots,V_n>}$ indicates the replacement of the free occurrences of the variables $V_1 \ldots V_n$ in κ by $\phi_1, \ldots, \phi_n$.

Patterns Rule

$$\begin{array}{c} \phi_1 : C_1, \ldots, \phi_n : C_n >> [P]_{<\phi_1 \ldots \phi_n>/<v_1,\ldots,v_n>} : A; \\ \underline{[P]_{<\phi_1 \ldots \phi_n>/<v_1,\ldots,v_n>} : A >> [X]_{<\phi_1 \ldots \phi_n>/<v_1,\ldots,v_n>} : B;} \\ (\lambda\ P\ X) : (A \rightarrow B); \end{array}$$

Figure 13.9 The rule for typechecking extended abstractions

The *Patterns Rule* subsumes the *Abstractions Rule* as a special case. Thus given P is a variable then the sequent $\phi_1 : C_1, \ldots, \phi_n : C_n >> [P]_{<\phi_1 \ldots \phi_n>/<v_1,\ldots,v_n>} : A$; collapses into a problem equivalent to the trivial P : C >> P : A (where C is a fresh variable). The second sequent is just the same as the problem posed by the *Abstractions Rule*.

We illustrate the action of these rules by reference to the example of the last chapter, which is reproduced here.

```
(Y (λ M (λ A (λ B (cases (((λ V (λ [ ] false)) A) B)
           (((λ X (λ (cons U W) (where (= U X) true)) A) B)
           (((λ X (λ (cons Z Y) ((M X) Y))) A) B))))))
```

Here is a proof that the function defined has the type (∀A (A → ((list A) → boolean))).

Proof:

```
(Y (λ M (λ A (λ B (cases (((λ V (λ [ ] false)) A) B)
        (((λ X (λ (cons U W) (where (= U X) true)) A) B)
        (((λ X (λ (cons Z Y) ((M X) Y))) A) B))))))
                           : (∀A (A → ((list A) → boolean)))
```

By the *Generalisation Rule* this is provable if

```
(Y (λ M (λ A (λ B (cases (((λ V (λ [ ] false)) A) B)
        (((λ X (λ (cons U W) (where (= U X) true)) A) B)
        (((λ X (λ (cons Z Y) ((M X) Y))) A) B))))))
                                    : (c → ((list c) → boolean))
```

By the *Combinator Rule* this is provable if

M : (c → ((list c) → boolean))
>> (λ A (λ B (cases (((λ V (λ [ ] false)) A) B)
(((λ X (λ (cons U W) (where (= U X) true)) A) B)
(((λ X (λ (cons Z Y) ((M X) Y))) A) B))))
: (c → ((list c) → boolean))

By the *Abstractions Rule* this is provable if

a : c, M : (c → ((list c) → boolean))
>> (λ B (cases (((λ V (λ [ ] false)) a) B)
(((λ X (λ (cons U W) (where (= U X) true)) a) B)
(((λ X (λ (cons Z Y) ((M X) Y))) a) B)))
: ((list c) → boolean)

By the *Abstractions Rule* this is provable if

b : (list c), a : c, M : (c → ((list c) → boolean))
>> (cases (((λ V (λ [ ] false)) a) b)
(((λ X (λ (cons U W) (where (= U X) true)) a) B)
(((λ X (λ (cons Z Y) ((M X) Y))) a) b)) : boolean

Let Δ = {**b : (list c), a : c, M : (c → ((list c) → boolean))**}, then by the *Cases Rule*, three cases remain to be proved.

1. Δ >> **(((λ V (λ [ ] false)) a) b) : boolean**
2. Δ >> **(((λ X (λ (cons U W) (where (= U X) true)) a) b) : boolean**
3. Δ >> **(((λ X (λ (cons Z Y) ((M X) Y))) a) b) : boolean**

Case 1: Δ >> **(((λ V (λ [] false)) a) b) : boolean**

Δ >> **(((λ V (λ [ ] false)) a) b) : boolean**

By the *Applications Rule* this is provable if

Δ >> **((λ V (λ [ ] false)) a) : (D → boolean)**
Δ >> **b : D**

The second problem is solved by *Sequents* with **D** |→ **(list c)**. We are left with

Δ >> **((λ V (λ [ ] false)) a) : ((list c) → boolean)**

By the *Applications Rule* this is provable if

Δ >> **(λ V (λ [ ] false)) : (E → ((list c) → boolean))**
Δ >> **a : E**

The second problem is solved by *Sequents* with **E** |→ **c**. We are left with

Δ >> **(λ V (λ [ ] false)) : (c → ((list c) → boolean))**

By the *Abstractions Rule* this is provable if

v : c, Δ >> **(λ [ ] false) : ((list c) → boolean)**

By the *Patterns Rule* this is provable if

v : c, Δ >> **[] : (list c)**
v : c, [] : (list c), Δ **>> false : boolean**

which are both solved by the *Primitive Rule*. The proof of the remaining cases is left to the reader.

Internal forms are all ephemeral objects that exist during the compilation of Qi functions and do not appear in the resulting object code. Consequently it is not possible to execute internal forms as actual procedures. Thus an attempt to evaluate the expression **(cases 1 2 3)** will return the message that **cases** is not defined.

13.8 Mutual Recursion

The ***Y*** combinator can also handle mutual recursion. Thus

```
(define even?
   {number - -> boolean}
   1 -> false
   X -> (odd? ((- X) 1)))

(define odd?
   {number --> boolean}
   1 -> true
   X -> (even? ((- X) 1)))
```

is represented first by individually representing each function in $\mathcal{L}$.

```
(λ A (cases ((λ 1 false) A)
             ((λ X (odd? ((- X) 1))) A)))

(λ A (cases ((λ 1 true) A)
             ((λ X (even? ((- X) 1))) A)))
```

The constituent functions of the mutual recursion are paired to form a tuple.

```
(@p (λ A (cases ((λ 1 false) A) ((λ X (odd? ((- X) 1))) A)))
   (λ A (cases ((λ 1 true) A)   ((λ X (even? ((- X) 1))) A))))
```

Each recursive call to a function in the tuple is replaced by an index to the position of the function in the tuple.

```
(@p (λ A (cases ((λ 1 false) A)
                   ((λ X ((snd T) ((- X) 1))) A)))
      (λ A (cases ((λ 1 true) A)
                   ((λ X ((fst T) ((- X) 1))) A))))
```

Finally, the parameter **T** is lambda-bound in the scope of a ***Y*** - combinator.[51]

```
(Y (λ T (@p (λ A (cases ((λ 1 false) A)
                   ((λ X ((snd T) ((- X) 1))) A)))
           (λ A (cases ((λ 1 true) A)
                   ((λ X ((fst T) ((- X) 1))) A))))))
                 : ((number → boolean) * (number → boolean))
```

We abbreviate **(@p …))))** by **P**, we can prove that **(*Y* (λ T P)) : ((number → boolean) * (number → boolean)).**

Proof Sketch: (*Y* (λ T P)) : ((number → boolean) * (number → boolean)) if **T : ((number → boolean) * (number → boolean)) >> P : ((number → boolean) * (number → boolean)).**

We leave the rest of the proof to the reader.

[51] See Field and Harrison (1988) for an exposition of this treatment of mutual recursion. Appendix G contains an example of the use of this technique to determine if 5 is even - not for the faint-hearted.

13.9 Global Variables

The *Global Rule* states that an assignment of Y to X has the type A just when X is a symbol and it is provable that both the current value of X has the type A and so does Y. Since Qi contains no rules that enable conclusions of the form (value X) : A to be proved, it is left to the user to enrich the type system by a rule stating the kind of object assigned to X in the way explained in chapter 11. **set** is also a special form.

Global Rule
where X is a symbol
(**value** X) : A; Y : A;
(**set** X Y) : A;

Figure 13.10 The rule for typechecking global assignments

13.10 Procedure $\mathcal{T}$*

In the absence of any user axioms, the proof procedure $\mathcal{T}$ for typechecking is depth-first search with chronological backtracking, and $\mathcal{T}$ is sound and terminating in respect of the type rules we have stated.[52] Another piece of good news is $\mathcal{T}$ is accurate, in the sense that if $\mathcal{T}$ says that an expression has a type, then it really does have that type.[53]

The downside is that $\mathcal{T}$ is not particularly efficient. The *Cases Rule*, for instance, splits a proof of (cases Case$_1$ … Case$_n$) : A into a number of subproofs that work on each Case$_i$. Since the type A is known and is variable-free, each Case$_i$ can be treated independently and there is really no need for backtracking between these subproofs of Case$_i$: A. Other rules offer opportunities for the elimination of choice points. The *Combinator Rule* is the only rule for checking combinators and so is a **committed choice** in the sense that if it can be applied, it should be and there should be no backtracking to reconsider this decision.

Another problem with $\mathcal{T}$ is that the procedure targets $\mathcal{L}$ expressions rather than Qi expressions, and this creates two problems. First, $\mathcal{L}$ expressions are generally larger and more deeply nested than Qi expressions, so the time spent on inferencing is proportionately greater.

[52] See appendix F. The proof procedure itself can be viewed by the **spy** command, which is used for debugging. See appendix H.
[53] See appendix E.

Second, any type errors detected are detected in $\mathcal{L}$ expressions and not in the Qi source, which makes it hard to raise a clear type error report.

The $\mathcal{T}^*$ procedure eliminates these disadvantages by type checking Qi source, and procedurally implements the role of many of the type rules used in $\mathcal{T}$. The Y-combinator, Pattern, Cases, Generalisation and Guard rules are redundant in $\mathcal{T}^*$. The bogus choicepoints are omitted. The $\mathcal{T}^*$ procedure operates on a typed Qi function definition. Such a definition generally has the form.

(define <function>
 $\{A_1$ - -> ... A_n - -> B}
 ${}_a p_1$ ... ${}_a p_n$ -> r_a
 ....
 ${}_m p_1$ ... ${}_m p_n$ -> r_m)

Each rule **${}_i p_1$ ... ${}_i p_n$ -> r_i** can be thought of as defining a sort of mini-function whose type is intended to be identical with the host function. Therefore to typecheck the entire function, it is enough to show that each rewrite rule obeys the type associated with its host function.

Under $\mathcal{T}^*$, the process of type checking a rewrite rule has two parts, corresponding to the two structural components of a rule. The first structural component is a sequence of patterns and the second is the result returned if the patterns match the inputs. The first job of the type checker is to show that each pattern **p** fits the type assigned to it. This we call **the integrity condition**. The second job is to show that, assuming the integrity conditions are met, that the result has the type expected of it. This we call the **correctness condition**. The two principles are defined below.

The Integrity Condition

An assignment of a type **B** to pattern **p** meets the integrity condition just when the sequent Δ >> **p** : **B** is provable; where Δ is the set of typings **{V, A | V : A}** where **V** is a variable in **p** and **A** is a fresh type variable.

The Correctness Condition

Assume a rule **p_1 ... p_n -> r** is assigned the type **$A_1 \rightarrow \ldots A_n \rightarrow B$**. Then this assignment meets the correctness condition just when the sequent **p_1 : A_1 ... p_n : A_n** >> **r** : **B** is provable.

Ƭ* verifies the type security of a function through establishing these two properties. Lets see what this means in concrete terms through an example. Here is a definition of a datatype called **details** which represents the details about a person – her name, address and telephone number.

```
(datatype details

  \details right\
 Name : string; Address : string; Telephone : number;
 _____________________________________________________
  [Name Address Telephone] : details;

 \details left\
 Name : string, Address : string, Telephone : number >> P;
 _____________________________________________________
  [Name Address Telephone] : details >> P;)
```

The function **address** returns the address of a person given their details.

```
(define address
  {details - -> string}
   [Name Address Telephone] -> Address)
```

Represented in cons form; this definition appears as:

```
(define address
  {details - -> string}
  (cons Name (cons Address (cons Telephone []))) -> Address)
```

To establish the type **details - -> string,** we have to satisfy the Integrity and Correctness Principles. The Integrity Principle requires that the following sequent be proved.

```
Name : A, Address : B, Telephone : C
  >> (cons Name (cons Address (cons Telephone []))) : details
```

The details right rule maps this sequent to three subgoals.

```
Name : A, Address : B, Telephone : C  >> Name : string;
Name : A, Address : B, Telephone : C  >> Address : string;
Name : A, Address : B, Telephone : C  >> Telephone : number;
```

These sequents are solvable under the assignments A $\mapsto$ string, B $\mapsto$ string, and C $\mapsto$ number. The Correctness Condition then requires that that the following sequent be proved.

(cons Name (cons Address (cons Telephone []))) : details
>> Address : string

The details left rule maps this sequent to an immediately soluble goal.

Name : string, Address : string, Telephone : number
>> Address : string

The Correctness Condition in its current form is adequate only for non-recursive functions. We define the expression '**r** is recursive w.r.t. *f* to mean that **r** occurs in a recursive definition of *f* and a recursive call to *f* occurs within **r**. The first revision to the Correctness Condition caters for such recursive calls.

The Correctness Condition (Revised)

Assume a rule $\mathbf{p}_1 \ldots \mathbf{p}_n$ **-> r is** is assigned the type $\mathbf{A}_1 \rightarrow \ldots. \mathbf{A}_n \rightarrow \mathbf{B}$ and **r** is not recursive w.r.t. *f*. Then this assignment meets the correctness condition just when the sequent $\mathbf{p}_1 : \mathbf{A}_1 \ldots \mathbf{p}_n : \mathbf{A}_n$ >> **r** : **B** is provable. Assume a rule $\mathbf{p}_1 \ldots \mathbf{p}_n$ **-> r is** is assigned the type $\mathbf{A}_1 \rightarrow \ldots. \mathbf{A}_n \rightarrow \mathbf{B}$ and **r** is recursive w.r.t. *f*. Then this assignment meets the correctness condition just when the sequent $f : \mathbf{A}_1 \rightarrow \ldots. \mathbf{A}_n \rightarrow \mathbf{B}$, $\mathbf{p}_1 : \mathbf{A}_1 \ldots \mathbf{p}_n : \mathbf{A}_n$ >> **r** : **B** is provable.

Here is an illustration using the linear recursive definition of the factorial function.

```
(define factorial
  {number - -> number}
  0 -> 1
  X -> (* X (factorial (- X 1))))
```

After currying, the proof obligations that the typechecker generates are:

1. **>> 0 : number;**
2. **0 : number >> 1 : number;**
3. **X : A >> X : number;**
4. **X : number, factorial : (number - -> number)**
   >> ((* X) (factorial ((- X) 1)))) : number

Sequent 1. is generated by applying the Integrity Condition to the first rewrite rule and sequent 3. by applying the same condition to the second rewrite rule. Sequents 2. and 3. come from applying the Correctness Condition to the first and second rewrite rules respectively. Since the expression **(* X (factorial (- X 1)))** is recursive with respect to **factorial**, an extra assumption **factorial : (number - -> number)** is required.

The Correctness Condition does not deal with guards or constructions using **<-**. The latter do not pose any special problem because they can be eliminated in favour of the forward arrow using the equivalences of chapter 7. The use of guards is also straightforward to typecheck. The rule

$$p_1 \ldots p_n \text{ -> } r \qquad \text{where } G$$

has the type $A_1 \rightarrow \ldots A_n \rightarrow B$ just when the Integrity Condition and Correctness Conditions are met and **G** can be proved to be a boolean under much the same assumptions as required in the proof of correctness. In other words that is, all of the following conditions are met; for each p_i, let Δ_i be the set of typings {**V, A | V : A**} where **V** is a variable in p_i and **A** is a fresh type variable.

1. $\Delta_i >> p_i : A_i$ is provable for each **i** of **1,…,n**. (Integrity)
2. **G : verified**, $p_1 : A_1 \ldots p_n : A_n >> r : B$ is provable. (Correctness)
3. The sequent $p_1 : A_1 \ldots p_n : A_n >> G :$ **boolean** is provable. (Guard)

Here is an example using these conditions.

```
(define find_address
  {string - -> [details] - -> string}
   Name [[Name Address Telephone] | _] -> Address
   Name [_ |  Details] -> (find_address Name Details))
```

The function contains no explicit guard, but it also contains a non-left linear rewrite rule in the first position. To render it left-linear for the purposes of type-checking, we need to insert a guard. After doing this, the definition appears as below.

```
(define find_address
  {string - -> [details] - -> string}
   Name1 [[Name2 Address Telephone] | _] -> Address
                                            where (= Name_1 Name_2)
   Name [_ |  Details] -> (find_address Name Details))
```

Seven proof obligations are needed to prove the function is well-typed (figure 13.11).

	Rewrite Rule 1	Rewrite Rule 2
Integrity	**Name1 : A >> Name1 : string; Name2 : B, Address : C, Telephone : D, Whatever : E >> [[Name2 Address Telephone] | Whatever] : (list details)**	**Name : F >> Name : string Detail : G, Details : H >> [Detail | Details] : (list details)**
Correctness	**Name1 : string, [[Name2 Address Telephone] | Whatever] : (list details), (= Name1 Name2) : verified >> Address : string**	**Name : string, [Detail | Details] : (list details), find_address : string - -> ((list details) - -> string) >> ((find_address Name) Details) : string**
Guard	**Name : string, [[Name2 Address Telephone] | Whatever] : (list details) >> (= Name1 Name2) : Boolean**	

Figure 13.11 The proof obligations from the **find_address** *function*

The proof obligation **Name_1 : A >> Name_1 : string** is trivially solvable. In fact problems of this kind are always generated from patterns that are simply variables, and so $\mathcal{T}^*$ omits these proof obligations because they are always solvable.

Finally only mutual recursion remains to be dealt with. The technique of handling a mutual recursion like

```
(define even?
  {number - -> boolean}
  1 -> false
  X -> (odd? (- X 1)))
```

```
(define odd?
   {number --> boolean}
   1 -> true
   X -> (even? (- X 1)))
```

is to assume the typings **odd? : number** → **boolean** and **even? : number** → **boolean** in typechecking the recursive calls. There are eight proof obligations generated by these two functions, but two are trivial and so we omit them. The remaining six are shown in figure 13.12.

	even?	odd?
Integrity	>> 1 : number;	>> 1 : number;
Correctness	1 : number >> true : boolean; X : number, odd? : (number --> boolean) >> (odd? ((- X) 1)) : boolean	1 : number >> false : boolean; X : number, even? : (number --> boolean) >> (even? ((- X) 1)) : boolean

Figure 13.12 The proof obligations from the **odd?** *and* **even?** *functions*

The technique of $\mathcal{T}^*$ is to prove Integrity and Correctness for each rule of the function. There is no need for the Y-combinator, Pattern, Cases and Guard rules and these rules are no longer retained. Type variables in the type of a function are replaced by fresh terms at the start of the proof and so the Generalisation Rule is not required. $\mathcal{T}^*$ trades declarative purity for a reduction in the size of the type theory, and the elimination of bogus choicepoints, but delivers the same result in less time.[54]

We shall prove the procedural equivalence[55] of $\mathcal{T}$ and $\mathcal{T}^*$ by showing how the rules not used by $\mathcal{T}$ are implemented procedurally within $\mathcal{T}^*$. The procedural equivalence is quite simple to show, except in the case of the Patterns Rule. Figure 13.13 shows the rules used by $\mathcal{T}$ and $\mathcal{T}^*$.

[54] The improvement in performance is significant; using $\mathcal{T}$ to typecheck and compile one program took 77,453 inferences. Under $\mathcal{T}^*$, the same program was checked and compiled using 10,187 inferences. In cases where there are type errors, $\mathcal{T}^*$ is less likely than $\mathcal{T}$ to lose itself in combinatorial complexities.

[55] I.e.; that they both produce the same result for the same input.

The efficiency gain in $\mathcal{T}^*$ comes from three sources.

1. The elimination of rules that are used only once in the proof. Instead these rules are procedurally invoked early in the procedure and then not used again.
2. The elimination of bogus choice points.
3. The elimination of long stretches of proof that are purely mechanical and add to the search space without changing the result.

The easiest way to demonstrate the procedural equivalence of $\mathcal{T}$ and $\mathcal{T}^*$ is to work through the rules that $\mathcal{T}^*$ does not use and see why $\mathcal{T}^*$ does not use them.

First, the Generalisation Rule needs only to be used once in $\mathcal{T}$, at the beginning of a proof to remove the universal quantifiers binding the type variables. Since all types are shallow types[56], the Generalisation Rule is not needed after these quantifiers are removed. This is done automatically in $\mathcal{T}^*$, so the Generalisation Rule is not needed.

The Cases Rule requires that each case in the body of a definition obeys the type constraints of the whole function. The type constraints are free of shared variables, so that in effect, any proof using Cases can be split into *n* independent subproofs. In $\mathcal{T}^*$, this is exactly what happens, each rewrite rule is checked separately to ensure that it meets the type constraints. This means that the Cases Rule is not needed in $\mathcal{T}^*$.

The Y-combinator Rule is used in $\mathcal{T}$ to typecheck recursive functions. Effectively any abstraction (λ x y) which amounts to the recursive definition of a function *f* is written as ($\boldsymbol{Y}$ *f* (λ x y)). The Y-combinator Rule, when applied to a problem of the form ($\boldsymbol{Y}$ *f* (λ x y)) : A, gives *f* : A >> (λ x y) : A. Thereafter the Y-combinator Rule plays no further part in the proof. An obvious procedural optimisation is to generate the problem *f* : A >> (λ x y) : A directly from the recursive definition, and dispense with the Y-combinator Rule. This means that the Y-combinator Rule is not needed in $\mathcal{T}^*$.

56 A shallow type is one where all the universal quantifiers binding the type variables occur to the far left of the type.

Principle Rules used by $\mathcal{T}$	Principle Rules used by $\mathcal{T}^*$	Rules not used by $\mathcal{T}^*$
Sequents	Sequents	Generalisation
Primitive	Primitive	Cases
Applications	Applications	Patterns
Generalisation	Global	Guard
Cases	@p Rules (left and right)	Y-combinator
Patterns	Cons Rules (left and right)	
Guard	Equality Rule	
Global	Conditional Rule	
Y-combinator	Abstraction	
@p Rules (left and right)		
Cons Rules (left and right)		
Equality Rule		
Conditional Rule		

Figure 13.13 The rules used by $\mathcal{T}$ and $\mathcal{T}^*$.

The optimisation of dispensing with the Patterns and Guard Rules is more delicate. Recall that in $\mathcal{T}$, the task of typechecking a function begins with a problem of the form $(\lambda\ x_1 \ldots(\lambda\ x_n\ \ y)) : A_1 \rightarrow \ldots A_n \rightarrow A_{n+1}$. Each x_i is a variable, and by repeated applications of the Abstractions Rule, we generate a problem $x_1 : A_1, \ldots, x_n : A_n >> y : A_{n+1}$. The body y of the former abstraction will consist of a series of cases $c_1, \ldots, c_m$ each of which must be proved to be of type A_{n+1}. That is, our proof obligation breaks down into the following series.

$x_1 : A_1, \ldots, x_n : A_n >> c_1 : A_{n+1}$

....

$x_1 : A_1, \ldots, x_n : A_n >> c_m : A_{n+1}$

Each c_i is an application composed of an n-place abstraction $(\lambda\ p_1 \ldots (\lambda\ p_n\ z))$ applied to $x_1, \ldots, x_n$ taking the form $(((\lambda\ p_1 \ldots (\lambda\ p_n\ z))\ x_1)\ldots\ x_n)$. Each p_i is a pattern. Thus the proof obligation for c_i is:

$x_1 : A_1, \ldots, x_n : A_n >> (((\lambda\ p_1 \ldots (\lambda\ p_n\ z))\ x_1)\ldots\ x_n) : A_{n+1}$

Repeated use of the Applications Rule gives the following proof obligations.

where $B_1 \dots B_n$ are free type variables

$x_1 : A_1,\dots, x_n : A_n$ >> $x_1 : B_1$
....
$x_1 : A_1,\dots, x_n : A_n$ >> $x_n : B_n$
$x_1 : A_1,\dots, x_n : A_n$ >> $(\lambda\ p_1 \dots (\lambda\ p_n\ \ z)) : B_1 \rightarrow \dots B_n \rightarrow A_{n+1}$

The proof obligations

$x_1 : A_1,\dots, x_n : A_n$ >> $x_1 : B_1$
....
$x_1 : A_1,\dots, x_n : A_n$ >> $x_n : B_n$

are trivially solvable by unifying each $x_i : B_i$ with the assumption $x_i : A_i$.

The final proof obligation is therefore

$x_1 : A_1,\dots, x_n : A_n$ >> $(\lambda\ p_1 \dots (\lambda\ p_n\ \ z)) : A_1 \rightarrow \dots A_n \rightarrow A_{n+1}$

The expressions x_1 ,..., x_n do not occur anywhere within the conclusion of this sequent. Consequently they play no further part in the proof and may be thinned away. The proof obligation can now be simplified to

>> $(\lambda\ p_1 \dots (\lambda\ p_n\ \ z)) : A_1 \rightarrow \dots A_n \rightarrow A_{n+1}$

n repeated applications of the Patterns Rule gives the following proof obligations. Each Δ_i is a series of typings of the form v : C, where v is a variable in p_i and C is a fresh type variable.

Δ_1 >> $p_1 : A_1$
....
Δ_n >> $p_n : A_n$
$p_1 : A_1 \dots p_n : A_n$ >> $z : A_{n+1}$

Let us pause at this point and observe that the proof process so far is not only entirely mechanical, but is guaranteed to succeed merely by virtue of the syntactic structure of the language $\mathcal{L}$. In other words, type failure can only occur past this point, and reconsideration of the previous steps is a waste of time.

The second point to observe is that the proof obligations now facing us correspond precisely to the integrity and correctness checks required by $\mathcal{T}$*. The integrity checks are contained exactly by the proof obligations

$\Delta_1 >> p_1 : A_1$
……………..
$\Delta_n >> p_n : A_n$

while the correctness check is contained by the proof obligation

$p_1 : A_1 \ldots p_n : A_n >> z : A_{n+1}$

The advantage of $\mathcal{T}^*$ is that the mechanical and time-wasting steps required in $\mathcal{T}$ to get to this point are eliminated.

13.11 From Type Checking to Automated Deduction

Qi gives the user enormous power to modify the type system. With enormous power comes the responsibility of saying what you mean. Poorly thought out type rules can cause the integrity of the system to fail.[57] Probably the shortest and most destructive type rule that can be added to Qi is given below.

$$\frac{}{X : A;}$$

This single rule collapses Qi into an untyped language. Everything becomes well-typed.

However we don't have to search for no-brainers to find rules that will upset the type checker. The wrong rules can cause type checking to go into a loop.[58] Here is an example. A student programmer Melvin considers that the list [1 a] should be well-typed since all its members inhabit the type of things that are either numbers or symbols. Accordingly he defines a type operator **or** with the following proof properties.

[57] In this respect programming in Qi is not different from any formal methods approach - if the specification is wrong then the proof of the correctness of the program is worthless.
[58] Qi provides a tool for tracing proofs of type correctness and its use is discussed in appendix H.

```
X : A >> P; X : B >> P;
X : (A or B) >> P;

X : A;
X : (A or B);

X : B;
X : (A or B);
```

Figure 13.14 A type definition that causes problems.

He enters **[1 a]** to Qi and gets **[1 a]: (list (number or symbol))** in return. However, when he enters **(* 1 a)** instead of a type error, the computer crashes. Why?

To see why, remember that Qi treats the problem of proving that **(* 1 a)** has a type as equivalent to proving that for some C, **(* 1 a)** : C. Applying the second rule to this problem, unifies the polytype expression (A or B) with C giving **(* 1 a)** : A as an output. Standardising this apart, we get **(* 1 a)** : D. The same rule can now be applied again …. and again …. and again *ad infinitem*. Rules like this which can be continually applied to generate an infinite proof branch are called **expansive rules**.

Qi does include a facility for switching from unification over types to simple pattern-matching using **mode declarations**. The following definition declares that **(A or B)** must match to the relevant type by using **(mode … -)**. **(mode … +)** switches over to full unification.

```
 (datatype or

X : A >> P; X : B >> P;
X : (A or B) >> P;

X : A;
X : (mode (A or B) -);

X : B;
X : (mode (A or B -));)
```

The default is full unification over types and pattern-matching on expressions to the left of the colon. Modes can be nested and the innermost mode takes priority.

However this is not a free lunch. We now have to explicitly place the type into the proof; so **[1 a]** will not be proved as being of type **(list (number or symbol))**. Instead we have to explicitly ask Qi to prove this by entering the typing **[1 a] : (list (number or symbol))**.

In general there is no easy way out of these sorts of problem; to prove conclusions in systems using expansive rules requires judgement and intelligence. As we move to the study of these more complex systems, so we become involved in **automated deduction** – the study of how to automate reasoning on the computer. In difficult cases, it is no longer reasonable to expect the computer to manage the process of proof without some guidance from a person. This guidance may be fully interactive, with the human guiding every step of the proof. Alternatively, the human may decide to transfer some of her expertise into the machine in the form of a reasoning program that is more sophisticated than Qi's inbuilt procedure. The construction of such programs is the theme of the next part of this book.

Exercise 13

1. Prove using $\mathcal{T}^*$ that the following functions have the following types:
   a. **plus** of figure 3.7 has the type number → (number → number).
   b. **fibonacchi** of figure 3.8 has the type number → number.
   c. **join** of figure 4.8 has the type ∀A (list A) → ((list A) → (list A))
   d. **rev** of figure 4.9 has the type ∀A (list A) → (list A).
   e. **powerset** of figure 4.13 has the type ∀A (list A) → (list (list A))
   f. **fix** of figure 5.12 has the type ∀A (A → A) → (A → A).

2. Implement the unification function.

3. **Using your answer to 2., implement $\mathcal{T}^*$. Your program should load and typecheck a Qi program from a text file.

4. **Using your answer to 2., implement $\mathcal{T}$. Your program should load and typecheck a Qi program from a text file.

5. **The type definition of figure 13.14 fails because the type-checker gets trapped in an infinite loop. A similar problem arose for the Proplog theorem-prover of chapter 7. The Proplog theorem-prover can be cured by designing a check on looping that makes the program to break out of the loop. Design a similar solution for our type-checker. You have to take into account that the looping behaviour of the type-checker is more complex than that of Proplog.

6. Melvin Micro types in the rules for binary from chapter 11 into Qi and then types **[(+ 1 0)] : binary** and gets a type error. He is puzzled because **(+ 1 0)** is just **1** and **[1]** is a binary number. Explain why Melvin gets this error.

7. Melvin continues to experiment. He types in the definition of a function that removes all the internal [...]s from within a list.

```
(define flat
 {[[A]] - -> [A]}
  [ ] -> []
  [[X | Y] | Z] -> (append (flat [X | Y]) (flat Z))
  [X | Y] -> [X | (flat Y)])
```

   and gets another type error. Why?

8. Melvin argues that the definition of complement in chapter 11 is too long. He redefines it as

```
(define melvins_complement
   {binary - -> binary}
    [X] -> (if (= X 1) [0] [1])
    [X | Y] -> (if (= X 0)  [1 | (melvins_complement Y)]
                            [0 | (melvins_complement Y)]))
```

   He finds that with type checking disabled, this function works on binary numbers just like the old complement function does. However **melvins_complement** does not typecheck. Explain why.

Further Reading

The first practical typechecking algorithm is described in **Milner** (1975). It was incorporated into ML and an axiomatisation for a typed functional language was described by **Cardelli** (1984); our system is closely modelled around Cardelli's axiomatisation. Discussions of alternative type systems can be found in **Diller** (1988), **Girard** (1989), **Odifreddi** ed. (1990), **Turner** (1991), **Thompson** (1991), **Barendregt** (1992), **Gunter** (1992) and **Peirce** (2002). **Cerrito** and **Kesner** (1999) describe a type system for an extended lambda calculus based on pattern-matching. **Robinson**'s 1965 unification algorithm remains the standard approach to performing unification. **Martelli** and **Montanari** (1973) describe a linear time and space algorithm for performing unification. **Knight** (1989) provides a good overview of the uses and ramifications of unification.

Web Sites

Nikola Bogunovic "Avoiding Infinite Loops in Rule-Based Systems with Backward Chaining" is a useful reference; see http://citeseer.nj.nec.com/348918.html.

Part III:

Automated Deduction, Expert Systems and Intelligent Agents

14 Proof Systems

14.1 Building Automated Theorem Provers

We closed the previous chapter with an examination of the type system of Qi, and we saw that the process of type checking was actually a process of automated deduction which could be mechanised so a computer could perform type checking unaided. We also saw that refinement rules enabled us to define types where it was very difficult to efficiently automate type checking. In this chapter, we will look at some applications of Qi to automated deduction - the original application area for which the language was designed.

Every automated deduction program can be represented as an attempt to mechanise the application of refinement rules to enable the automatic proof of theorems. Such programs vary from systems that require human guidance at every step of the proof (called **proof checkers**) to fully **automated theorem-provers** (ATPs) that require little or no human guidance at all. In between is a large class of ATPs that are capable of performing some steps automatically, but may require human guidance from time to time. These programs have been written in many languages, including C, Prolog, Lisp and SML. Some have proved difficult theorems in mathematics and logic.

But is it justified to accept the verdict of a computer? Suppose we have encoded an ATP in 10,000 lines of C. What guarantee is there that this ATP is correct? How can we sure that the formulae that the ATP says are theorems are really theorems? It is almost impossible to write a C program over 1,000 lines that does not contain significant errors, and an ATP of 10,000 lines of C is almost certain not to work properly unless it has been debugged and tested over several man-years.

This is the **reliability problem** in building ATPs; we have a set of refinement rules and we have a theorem-prover; but how do we know that the program faithfully encodes what is in the refinement rules? To put the matter formally, suppose $\Delta \vdash_F A$ holds when A follows from a set of formulae Δ by the refinement rules of F. Suppose *der*(Δ, A) holds when our theorem-prover T for F says "proved" (or some equivalent verdict). How do we know that *der*(Δ, A) implies $\Delta \vdash_F A$? In other words, how do we know T is *sound* in the sense of chapter 7?

One way of increasing certainty is to write T so that the only operations that build up proofs are those operations that apply refinement rules of F. So as long as we have correctly represented the refinement rules of F, T should be sound. Qi was originally designed to facilitate the high-level specification of ATPs, and working in Qi, this encoding of refinement rules is very easy. We just write them to Qi and Qi compiler will compile these refinement rules to machine code, removing the need for us to write our own routines for manipulating the proof.

But working at the level of refinement rules is extremely tedious. The system is a proof checker, nothing more. Difficult proofs require large numbers of steps. But not every proof-building operation need be so low-level as the application of a single refinement rule. An operation in T may be defined in terms of many applications of refinement rules; there is no need to work at the lowest possible level.

As an analogy, consider the way in which computers are programmed. Every computer has a machine code instruction set within which all instructions to it are translated. Despite its very low level, that instruction set will support high-level mainstream computer languages like C++ and Prolog. Each high-level language is supported by a compiler from native source into the resident machine code instruction set. For ATPs, our machine code instruction set is the set whose elements apply one refinement rule. Our goal is to evolve a high-level **proof language** for calling up these primitive operations, so that long proofs can be executed in a few steps rather than many. In this chapter, we will see how to build such proof languages in Qi. We begin with a look at **proof systems**.

14.2 Proof Systems

A proof system is a notation devised for representing reasoning. There are many different kinds of proof system; but they all share a common cluster of features.

1. A proof system L has **syntax** rules: these are rules of formation that determine when an expression is a formula of that proof system and when it does not. For example, thinking of arithmetic as a proof

system, the expression 1 + 2 - (4 * 5) is a formula; but * 6 * -78 + + - 7 is not. Formulae of a proof system that meet the requirements of its syntax are called **well-formed formulae** of L or **wffs** for short.

2. A proof system L has a **proof theory**. A proof theory defines the provability relation, usually by giving refinement rules for proving sequents. If A is provable from a set Δ of wffs of L then we write $\Delta \vdash_L A$. If Δ is empty then A is a **theorem** of L, written $\vdash_L A$.

   For example, in algebra $a + b = b + a$ is a theorem; i.e. $\vdash_{algebra} a + b = b + a$. But $a + b = c + a$ is not a theorem, though $b = c \vdash_{algebra} a + b = c + a$ holds.
3. A proof system has a **semantics**. A semantics gives rules for assigning interpretations to wffs. When the symbols of a wff A are given meaning in accordance with the semantics, we say A has been given an **interpretation** *I*. When a wff A is true under *I* then *I* is a **model** of A, written $I \models_L A$. If A is false under I then I is a **countermodel** of the wff. When every interpretation of A is a model then A is **valid**, written $\models_L A$. A proof system in which every theorem is valid is referred to as **sound**. A proof system in which every valid wff is a theorem is **complete**. If a proof system is both sound and complete then its semantics is said to **characterise** that proof system.

For example, we may interpret *a*, *b*, *c* in $a + b = c$ as denoting 1,2 and 3 respectively, but not Tom, Dick and Harry. The semantics of algebra forbids this sort of interpretation. The interpretation that assigns *a* in $a = a * a$ the value 1 is a model since 1 = 1 * 1 is true. However $a = a * a$ is not valid in algebra since the interpretation that assigns *a* the value 2 is a countermodel, since 2 = 2 * 2 is false. $a * b = b * a$, on the other hand, is obviously valid.

4. A proof system may have a **heuristics**; or a set of principles to help people (or computers) reason with it. For example, in algebra we have a principle that tells us to isolate all occurrences of an unknown on one side of an equation. This is part of the heuristics of algebra.

5. The symbols of any proof system partition into a set of **logical constants** and a set of **logical variables**. For example in algebra, the constants are the **+, -, *** and **/** and the variables *a*, *b*, *c* etc. Two wffs are **variants** of each other if they can be derived from each other by the uniform substitutions of the logical variables within them. An important principle we will call the **variant principle** states.

The Variant Principle

If $\vdash_L$ A, and B is a variant of A, then $\vdash_L$ B.

So given $a * b = b * a$ is a theorem of algebra, so is $a * c = c * a$, $(a - b) * c = c * (a - b)$ etc.

14.3 Propositional Calculus

One familiar proof system is **propositional calculus** (PC), which captures some basic patterns of inference involving the connectives & (and), v (or), ~ (not), → (if ... then ...), and ↔ (if and only if). These elements are the logical constants of the propositional calculus.

As well as these connectives there is an unlimited supply of propositional variables p, q, r, s, p′, q′, r′, s′... . These propositional variables can be made to stand for sentences. Thus identifying p with the sentence "Logic is easy." and q with the sentence "Students like logic." determines the meaning of all propositional wffs containing only p and q as variables (figure 14.1).

Wff	*Interpretation*
(p → q)	If logic is easy then students like logic.
(p v q)	Either logic is easy or students like logic.
((~ p) v q)	Either logic is not easy or students like logic.
((~ p) & (~ q))	Logic is not easy and students do not like logic.
(p ↔ q)	Logic is easy if and only if students like logic.

Figure 14.1 A reading of some propositional calculus wffs

The syntax of propositional calculus can be stated quite simply.

A propositional variable on its own is a wff.
If A and B are wffs so is each of the following: (~ A), (A & B), (A → B), (A v B), (A ↔ B).

There are many ways of stating the proof theory of propositional calculus and the one we shall use is perhaps one of the most suited to the purposes of both human beings and computers. It is based on the **semantic tableau** approach in **Smullyan** (1968).

All proofs in semantic tableau use the strategy of **indirect proof**. An indirect proof establishes a conclusion A from a set of assumptions Δ by showing that (~ A) and Δ are jointly contradictory. The first step of an indirect proof assumes (~ A) as a preparation for deriving the contradiction. The refinement rule that enables this step we can call the rule of Indirect Proof.

$$\frac{(\sim A) >> A;}{A;}$$

This refinement rule states that if we are trying to prove A, then we are allowed to assume (~ A). In fact, we can derive any proposition B from a set of assumptions Δ by showing that Δ contains both A and ~A. The refinement rule that makes this claim is the refinement rule of *reductio ad absurdum* (raa).

$$\frac{}{A, (\sim A) >> B;}$$

The next two refinement rules deal with the behaviour of & and v. The first, &>>, allows us to break any assumption (A & B) into A and B. The second, v>>, allows the proof of C from (A v B) if C can be proved from A and C can be proved from B.

&>>

$$\frac{A, B >> C;}{(A \,\&\, B) >> C;}$$

v>>

$$\frac{A >> C;\ B >> C;}{(A \vee B) >> C;}$$

The following refinement rules are also useful; they explicate definitions of the other PC connectives and state certain basic properties of classical negation (figure 14.2).

→>>

$$\frac{((\sim A) \vee B) >> C;}{(A \rightarrow B) >> C;}$$

↔>>

$$\frac{((A \rightarrow B) \,\&\, (B \rightarrow A)) >> C;}{(A \leftrightarrow B) >> C;}$$

~ ↔>>

$$\frac{(\sim ((A \rightarrow B) \,\&\, (B \rightarrow A))) >> C;}{(\sim (A \leftrightarrow B)) >> C;}$$

~ →>>

$$\frac{(A \,\&\, (\sim B)) >> C;}{(\sim (A \rightarrow B)) >> C;}$$

~ &>>

$$\frac{((\sim A) \vee (\sim B)) >> C;}{(\sim (A \,\&\, B)) >> C;}$$

~ v>>

$$\frac{((\sim A) \,\&\, (\sim B)) >> C;}{(\sim (A \vee B)) >> C;}$$

~~>>

$$\frac{A >> C;}{(\sim\ (\sim A)) >> C;}$$

Figure 14.2 The proof rules of PC

14.4 Propositional Calculus in Qi

When building theorem-provers in Qi, it is important to know that the formulae we are expected to use form a type in themselves - the type **wff.** The programmer defines this type, so to state the syntax of a proof system, we have to define the conditions under which an expression can be proved to be of the type **wff.** The type **wff** is defined in figure 14.3 for propositional calculus; this follows the syntax laid down for the propositional calculus in the previous section. We use **=>** for $\rightarrow$ and **<=>** for $\leftrightarrow$. Finally, amongst the syntax of propositional calculus a declaration is included of all the logical constants in the propositional calculus.

```
(datatype wff

  X : symbol;
  X : wff;

  X : wff;
  =======
  [~ X] : wff;

  if (element? C [=> v & <=>])
  X : wff;   Y : wff;
  ============
  [X C Y] : wff;)

(define constant?
   {symbol - -> boolean}
    C -> (element? C [=> v <=> ~ &]))
```

Figure 14.3 The syntax of propositional calculus in Qi

We add the refinement rules for PC from the definitions in the previous section.

```
(theory pc

name indirect-proof
[~ P] >> P;
P;

name raa
______________
[~ P], P >> Q;

name &>>
P, Q >> R;
[P & Q] >> R;

name v>>
P >> R; Q >> R;
[P v Q] >> R;

name ->>>
[[~ P] v Q] >> R;
[P => Q] >> R;

name <->>>
[[P => Q] & [Q => P]] >> R;
 [P <=> Q] >> R;

name ~~>>
P >> Q;
[~ [~ P]] >> Q;

name ~&>>
[[~ P] v [~ Q]] >> R;
[~ [P & Q]] >> R;

name ~v>>
[[~ P] & [~ Q]] >> R;
[~ [P v Q]] >> R;

name ~=>>>
[P & [~ Q]] >> R;
[~ [P => Q]] >> R;

name ~<=>>>
[[~ [P => Q]] v [~ [Q => P]]] >> R;
[~ [P <=> Q]] >> R;)
```

Figure 14.4 The proof rules of propositional calculus in Qi

The keyword **theory** signals that the process of proving theorems of PC will allow human interaction and will not be left to the control of the computer.

With type checking disabled, the proof rules of the system are loaded directly into Qi with no attempt to check their correctness with respect to the syntax of the system. With type checking enabled, the rules are type checked to ensure that only syntactically valid sequents are generated by the system.

14.5 The Proof Tool

Having defined PC and loaded the program,🗀 the next step is to prove something. Qi provides a **proof tool** to help with this. The initial problem is a sequent that is entered to the proof tool and is then automatically placed on the goal stack. The proof tool gives the user control over the proof process, allowing the free choice of any refinement rule at any stage of the proof process. To illustrate, here is a proof in PC of **p, (p => (q v r)), (~ q) >> r.** To start the proof tool, the invocation is **(prooftool -).**

```
(4+) (prooftool -)

PC Theorem Prover[59]

ASSUMPTIONS:-

1>
```

The proof tool is inviting us to type in our assumptions. These are typed in as wffs.[60]

```
1> p
2> [p => [q v r]]
3> [~ "q"]
this is not a wff
3> [~ q]
4> ok
```

🗀 Qi Programs/Chap14/pc.qi

[59] The name of the system is generated from evaluating **(atp-credits "PC Theorem Prover").**

[60] In these pages the formulae are entered in the default mode using […]s. The function **display-mode** (see appendix A) will allow round brackets to be used instead.

CONCLUSION:-

?– r

On line 3, an erroneous input was entered. The type-checker, armed with pc syntax, detects the mistake, and allows us to reinput the entry. **ok** signs off the entry of assumptions. The proof tool asks for the conclusion. This is typed in too. Should we type **ok** as the conclusion the attempted proof will be aborted and returned to the top level. The problem is then restated as the first step of the proof.

```
Step 1. [1]                         0 refinements

?– r

1. p
2. [p => [q v r]]
3. [~ q]

PC: indirect-proof
```

The prompt appears with the name of the proof system.[61] A number also appears in square brackets that states how many sequents there are currently held in the goal stack. Next to this number is a performance monitor that tells us how much CPU time has been spent, how many inferences (called *refinements*) have been performed, and the number of inferences executed per second. The inevitable first step in a proof of our version of **pc** is to negate the conclusion and add it to the list of assumptions. This is encapsulated by the refinement rule of **indirect-proof** that was the first refinement rule of the theory **pc**.

```
Step 2. [1]                         1 refinement

?– r

1. [~ r]
2. p
3. [p => [q v r]]
4. [~ q]

PC: =>>>
```

[61] Generated by the evaluation of **(atp-prompt "PC: ")**.

The proof tool prints the second step. The third step results from expanding **[p => [q v r]]** according to the definition of **=>** in terms of **v.**

Step 3. [1] **2 refinements**

?– r

1. [[~ p] v [q v r]]
2. [~ q]
3. [~ r]
4. p

PC: v>>

The next step is to eliminate the **v; v>>** does this. The number in square brackets increases by 1 indicating that we are acquiring an extra sequent to solve.

Step 4. [2] **3 refinements**

?– r

1. [~ p]
2. [~ q]
3. [~ r]
4. p

PC: raa

There is a contradiction in this sequent, so it can be solved by typing **raa**. The number in square brackets decreases by 1 showing that a sequent has been successfully solved.

Step 5. [1] **4 refinements**

?– r

1. [q v r]
2. [~ q]
3. [~ r]
4. p

PC: v>>

```
Step 6. [2]                                5 refinements

?– r

1. q
2. [~ q]
3. [~ r]
4. p

PC: raa
____________________________________________________
Step 7. [1]                                6 refinements

?– r

1. r
2. [~ q]
3. [~ r]
4. p

PC: raa
0.0400576 seconds          7 refinements          175 RPS
proved : symbol
```

The **proved** at the end of the proof, indicates that the proof has been successfully completed and that the original problem is a theorem of the logic. Whenever a command is entered to the proof tool, it is type checked and badly typed commands will not be executed. Here is a short illustration.

```
Step 1. [1]          0 secs, 0 refinements, infinite RPS

?– r

1. p
2. [p => [q v r]]
3. [~ q]

PC: reverse
error: this is not a tactic

PC:
```

14.6 Solving Goals in Qi

In Qi there are 4 functions for generating new goals, **refine**, **swap**, **rotate** and **thin.**

Refine has the type **symbol → (number → ((list parameter) → (goals → goals)))**, where the symbol denotes a theory *T* and the integer number the *n*th refinement rule in *T*. This function also receives a list of parameters which by default is empty. Refine applies the *n*th refinement rule of *T* to the current sequent. If either the theory does not exist or no refinement rule is denoted by the integer or matching fails, then the call to **refine** behaves as a call to the identity function.

This is easily the most important function for changing goals. An invocation to an refinement rule by name is only a shorthand for the use of **refine**; **indirect-proof** could be replaced by the less mnemonic **refine pc 1**. The application of the **refine** function is the basis for measuring the speed of ATPs written in Qi in terms of **RPS** or **refinements per second.**[62]

Swap followed by integers *m* and *n* exchanges the *m*th and *n*th wffs in the list of assumptions in the leading goal. The type of swap is **number → (number → (goals → goals))**.

Rotate followed by integers *m* and *n* exchanges the *m*th and *n*th goals in the list of unsolved goals. The type of rotate is **number → (number → (goals → goals))**.

Thin followed by an integer *n*, removes the *n*th assumption from the leading goal. Its type is **number → (goals → goals).**

Some useful interactive commands which can be used in the proof tool are **back** *n*, which scrolls the proof back *n* steps and **^**, which aborts the proof. **(prooftool +)** typed to the top level repeats the last problem sent to the proof tool.

[62] Figures of 170,000 RPS are obtainable on a 2.6 GHz Pentium 4 processor from the theorem-prover described in the previous section. This very high performance is characteristic of Qi ATPs and has its foundation in the Qi rule compiler which reduces all sequent rules to byte code under CLisp.

14.7 A Decision Procedure for PC

In Qi, a **tactic** is a function that can change the goals of a proof. The first four functions mentioned in the previous section qualify as tactics under this definition. However, tactics may accomplish a lot more than single refinement steps. We could define a function *f* that made several refinements to a series of goals and *f* would be a tactic too. A tactic may require that a particular series of refinements be applied until a result is obtained. The only condition that we place on a function to be a tactic is that it receives amongst its inputs, a series of goals and output a series of goals.

There is a very special class of tactics that have the following features.

1. They are applicable to any problem of the proof system in question.
2. They are guaranteed to terminate with a completed proof if one exists.
3. They are guaranteed to terminate with a negative answer if no proof exists.

Such tactics are said to be **decision procedures** for the proof systems they work on, and the proof system is then said to be **decidable**. Propositional calculus is one proof system that is decidable and there are several decision procedures for it; one of them is implemented here. The rules of this decision procedure are simple.

1. Every proof begins with **indirect-proof** and this is the only place where this refinement rule is invoked.
2. All the other refinement rules are applied to a fixpoint.
3. If the proof tool returns **proved** then the formula is a theorem, otherwise it is not.

14.8 Expressing our Heuristics for PC in Qi

Any intelligent human being can follow the rules of the previous section, but since they are mechanical in nature it is easier to program the computer to do this task. Here is the program.

```
(define pc_decision_procedure
  {goals - -> goals}
  Goals -> (fix pc-solve (indirect-proof Goals)))

(define pc-solve
  {goals - -> goals}
  Goals -> (v>> (&>> (=>>> (<=>>> (~~>> (~v>> (~&>> (~=>>>
                              (~<=>>>  (raa Goals))))))))))
```

Figure 14.5 An ATP for propositional calculus in Qi

The tactic **pc-solve** applies all the refinement rules of PC (excepting **indirect-proof**) and is itself driven by the higher-order function **fix**, which applies this function to a fixpoint. **fix** is a **tactical** i.e. a higher-order function that takes a tactic as part of its input. Notice that the program contains no test to see if all the goals have been solved; this is because the proof tool will exit automatically with **proved** if an empty goal stack is dispatched to it. By saving this code into a file and entering this file as part of the heuristics of PC, we can use **pc_decision_procedure** to solve problems.

```
Step 1. [1]                              0 refinements
?– r

1. p
2. [p => [q v r]]
3. [~ q]

PC: pc_decision_procedure
0 seconds              51 refinements         infinite RPS
proved : symbol
```

If an insoluble problem is submitted, **pc_decision_procedure** lands us with an insoluble sequent.

```
Step 1. [1]                              0 refinements

?– [q => p]

1. [p => q]

PC:  pc_decision_procedure
```

```
Step 2. [2]                    21 refinements

?– [q => p]

1. [~ p]
2. q
3. [~ p]

PC: ^
error: input aborted
Exit proof?  (y/n)
```

Throughout user interaction, the proof tool maintains a history of every step the user makes. If the proof has been gained automatically, then the record of the interaction will give little information as to how the proof was derived. In this case, the user may decide to trace the execution of the program using the **prf** command that receives a tactic as an input. Thus **(prf v>>)** entered to the top level will record all proof steps involving **v>>.** **(unprf v>>)** will cause these proof steps to be unrecorded. **prf** does not work on functions that are not tactics. The function **dump-proof <filename>** will pretty-print these proof steps to the named file where **<filename>** is a string.

Once a theorem has been proved, then it may be turned into a derived rule of inference by **theorem introduction**. The function **thm-intro** of type symbol → symbol, receives a symbol input S and generates a refinement rule from the last successful proof. This refinement rule is sent to a file named S, and S is then loaded into the Qi image. Thus having proved, say, **(p & q) >> (q & p)**; the command **(thm-intro comm-&)** will generate a refinement rule of the form:

```
name comm-&
_______________
[P & Q] >> [Q & P];
```

which is placed in the file **comm-&**. The replacement of the lowercase letters by variables proceeds according to the uniform substitution of non-logical constants.

14.9 Special Variables

When a theory is entered into Qi, the Qi compiler generates a series of Qi functions that encapsulates the functionality of the rules of that datatype or theory. In the case of theory declarations, the code generated is type-secure with respect to the syntax for wffs entered by the user. In the course of generating this code, Qi uses several locally bound **special variables** that are machine generated and these variables have types associated with them. Each of these special variables performs a task within the generated code and they can be accessed, and, if necessary, rebound (within the bounds of type integrity) at the behest of the user. Figure 14.6 gives a table of these special variables, their types, and the function they perform.

Variable	Type	Function
Assumptions	(list wff)	Stores the list of assumptions in the goal at the head of the goal stack.
Parameters	(list parameter)	Stores the list of parameters passed by the refine function. If no parameters are passed; this list is empty. See the next chapter for the use of parameters.
Notes	(list note)	Stores the list of notes associated with the ongoing proof. If notes are not used in the proof, this list is empty. See the next chapter for the use of notes.
Sequents	(list ((list wff) * wff))	Stores the list of sequents still on the goal stack (including the one currently being processed). Each sequent is stored as a pair composed of a list of wffs (the assumptions) and a wff (the conclusion).

Figure 14.6 Special Variables in Qi

All these variables are locally bound and are both accessible to the user and alterable within the constraints imposed by their types. Some of these variables are of little direct use to the user outside of very advanced applications.

14.10 Modal Logic and Non-Monotonic Logic

This section looks at some features of Qi that enable us to represent modal logics and non-monotonic reasoning. Our treatment is not designed to introduce these areas in depth, but only to illustrate the power of Qi to capture different proof systems. Those interested in pursuing these areas in depth are referred to the further reading at the end of this chapter.

Modal logic is a study of inference involving necessity and possibility. Rather basically, it can be viewed as an extension of classical logic with two propositional operators ◊ and □ meaning “It is possible that….” and “It is necessary that…" respectively. One of the simplest systems of modal logic is system T, invented by **Robert Feys** in 1937. It adds to the rules of propositional calculus, the following rules.

1. If P is a theorem then (☐ P) is derivable.
2. From (☐ P), P follows.
3. (☐ (P → Q)) implies ((☐ P) → (☐ Q))
4. (◊ P) is equivalent to ~(☐ (~ P))

A straightforward extension of the syntax for PC suffices to represent the syntax of modal propositional logic to Qi. The special variables are very useful here, and, as a case in point, we can use one of them to encode the first rule of system T.

If P is a theorem then (☐ P) is derivable

The obvious question here is "What is meant by 'P is a theorem' ?". The logician's reply (which is basically correct) is that P is a theorem provided that P can be proved from an empty set of assumptions. We can encode this idea by binding the **Assumptions** variable to the empty list within the encoding of this rule. The effect of this binding is to empty the next sequent of all its assumptions. Here is the code.

```
name >>nec
let Assumptions [ ]
P;____
[nec P];
```

Coding the extra rules for this logic on top of the rules for PC is also straightforward, and we leave this task as an exercise in this chapter.

The systems so far studied are all **monotonic** logics; a monotonic logic obeys the principle that if a conclusion is derivable from a set of assumptions, then it remains derivable even if more assumptions are added. Researchers in AI have argued that human reasoning is frequently **non-monotonic**; learning extra information can lead us to withdraw conclusions that we inferred previously.

It is simple to represent the Proplog of chapter 7 as a proof theory using the approach in this chapter. It is also possible to enrich Proplog to behave as a non-monotonic logic by the addition of **negation-as-failure.** To do this we add a propositional operator **not** whose proof properties are that (**not** P) is provable just when P is not provable. To show that Proplog is now non-monotonic, consider that from the assumption set {P ← (not Q)}, P is derivable (since P is provable if Q is not provable; and Q is not provable), but from {P ← (not Q), Q} P is no longer derivable. Representing negation-as-failure in Qi is quite simple

```
name >>neg
if (not (provable? proplog [(@p Assumptions P)]))
_________________________________________________
[not P];
```

The function **provable?** has the type (goals → goals) → [([wff] * wff)] → boolean. It receives

1. a tactic of type goals → goals,
2. a list of tuples; where each tuple is a pair made up of a list of wffs (the assumptions) and a wff (the conclusion)

provable? attempts to solve the sequents using the tactic, returning **true** if the attempt is successful and **false** otherwise. Here we invoke recursively the top-level **proplog** tactic in defining negation-as-failure. We leave the implementation of Proplog with negation-as-failure as a final exercise.

Exercise 14

1. Prove the following using the proof tool, but without using pc_decision_procedure.

   a. p → q |- (r v p) → (r v q)
   b. |- (p → q) → ((r → p) → (r → q))
   c. (q → p), (~q → p) |- p
   d. r → p, p → q |- r → p
   e. |- p → p
   f. p v ~p
   g. |- p → ~~p
   h. |- ~q → (q → r)
   i. |- p v (q v r) → ((q v (p v r)) v p)
   j. |- (q v (p v r)) v p → (q v (p v r))
   k. |- p v (q v r) → (q v (p v r))
   l. |- (p → (q → r)) → (q → (p → r))
   m. |- (r → p) → ((p → q) → (r → q))
   n. (p → (q → r)), (p → q) |- (p → (p → r))

2. *Implement system T.
3. *Implement Proplog with negation as failure.

Further Reading

The article on formal systems by **Henkin** (1967) gives a good overview of proof systems. Good general introductions to automated reasoning are **Bundy** (1983) and **Duffy** (1991). For a fascinating study of the prehistory of the field, before the invention of the computer, **Gardner** (1983) presents a survey from the time of the Crusades to 1950. **Loveland** (1984) is an excellent summary of the work done in the late '50s and '60s in the area of automated deduction. The matrix method explained in **Bibel** (1981) has recently become popular as a new way of solving FOL problems and it has much in common with the tableau method though it is claimed to be more efficient. See **Ramsay** (1988) for an overview of the method.

Tarver (1992) described a machine learning algorithm to get the computer to program its own tactics. The experiment was conducted on a version of the tableau theorem-prover for PC and uses genetic programming techniques. **Lopes** (1998) developed this algorithm for a range of systems including modal, intuitionistic and second-order logics and showed that in certain cases, the computer-generated programs were superior to the humanly created ATPs for the same logic.

Reeves and **Clarke** (1990) explain how to extend tableau to provide a decision procedure for modal logic S4. **Hughes** and **Cresswell** (1968, 1984, 1996) is the standard source for modal logic. Philosophical discussions of modal reasoning can be found in **Linsky** (ed.) (1975) and **Kripke** (1971). There are now annual conferences on tableau reasoning systems (the proceedings are published in the Springer-Verlag series).

Some key readings in non-monotonic logic are **Reiter** (1980), **McDermott** and **Doyle** (1980) and **McCarthy** (1980).

Web Sites

The Stanford Online Encyclopaedia of Philosophy (http://plato.stanford.edu/entries/logic-modal) contains a discussion of various modal logics.

15 First Order Logic

In propositional calculus, atomic sentences are simple symbols. If we take the two sentences “Tarver likes the *Guardian*” and “Tarver likes the *Independent*”, and represent them in PC we derive two sentences **p** (representing “Tarver likes the *Guardian*”) and **q** (representing “Tarver likes the *Independent*”). Neither of these representations gives any clue that these two sentences share anything - that they both concern me and what I like.

First-order logic (FOL) captures these similarities by allowing atomic sentences, unlike those of PC, to have an internal structure. In FOL, an atomic sentence is composed of a **predicate** followed by zero or more **terms**. The predicate ascribes properties to the objects denoted by the terms. So in FOL, “Tarver likes the *Guardian*” is represented as “likes(Tarver, *Guardian*)” and “Tarver likes the *Independent*” as “likes(Tarver, *Independent*)”.

If we replace “Tarver” in “likes(Tarver, *Guardian*)” by a variable “x”, we derive the **open sentence “**likes(x, *Guardian*)” in which the x is **free**. Though both “Tarver likes the *Guardian*” and “Tarver likes the *Independent*” are true, an open sentence such as “likes(x, *Guardian*)” is neither true nor false because variables do not denote objects. One way to turn open sentences into sentences that are true or false is to replace the variables by denoting terms (like “Tarver”, or “the man next door to me in #28”). Another way is to **bind** the variables by the **quantifiers** ∀ or ∃. Here is an example.

for every substitution for x, “likes(x, *Guardian*)” is true.

which in FOL is written “∀x likes(x, *Guardian*)”, which is plainly false. Another way is to assert

for some substitution for x, “likes(x, *Guardian*)” is true.

which is itself true, and which in FOL is written “∃x likes(x, *Guardian*)”.

15.1 The Syntax and Proof Theory of FOL

Since FOL includes all of the logical constants of PC, as well as quantifiers and structured atomic sentences, its expressive capacity is much greater than PC. Correspondingly, the syntax of FOL is more complex than PC.

Syntax Rules for First Order Logic

1. A variable is any of x,y,z,x′,y′,z′..
2. A predicate is any non-variable symbol other than ~, &, →, v, ↔, ∃, ∀.
3. A name is any non-variable symbol other than ~, &, →, v, ↔, ∃, ∀.
4. A functor is non-variable symbol other than ~, &, →, v, ↔, ∃, ∀.
5. A term is either (a) a name or (b) a variable or (c) an expression (f t_1,...,t_n) consisting of a function symbol f followed by *n* (*n* ≥ 0) terms.
6. P(t_1,..t_n) is a wff if P is a predicate and t_1,..,t_n are terms (*n* ≥ 0).
7. If A and B are wffs so are (~ A), (A & B), (A → B), (A v B), (A ↔ B).
8. If A is a wff and v is a variable then (∃ v A) and (∀ v A) are wffs.

Our syntax for FOL allows for 0-place predicates, but these are, in effect identical to the propositional variables of PC. Hence we will borrow on the syntax of PC and write (P v Q) rather than (P( ) v Q( )). All the refinement rules of PC are also refinement rules of FOL. In addition, the refinement rules in figure 15.1 also hold. The expression $A_{t/v}$ is the result of replacing of all free instances of a variable v by a term t.

∀>>
where t is any term
$\underline{A_{t/v}, (\forall\ v\ A) >> B;}$
(∀ v A) >> B;

∃>>
where t is fresh
$\underline{A_{t/v} >> B;}$
(∃ v A) >> B;

~∃
$\underline{(\forall\ v\ (\sim A)) >> B;}$
(~ (∃ v A)) >> B;

~∀
$\underline{(\exists\ v\ (\sim A)) >> B;}$
(~ (∀ v A)) >> B;

Figure 15.1 The quantifier rules for FOL

15.2 Expressing FOL in Qi

In Qi, instead of ∀, **all** will be used and instead of ∃, **some** will be used. To begin, we define a set of logical constants by defining the function constant?.

```
(define constant?
 {A - -> boolean}
 X -> (member? X [v ~ => <=> & all some]))

(define member?
  {A - -> [B] - -> boolean}
   _ [] -> false
   X [Y | _] -> true           where (== X Y)
   X [_ | Z] -> (member? X Z))
```

Notice that **==** is used to give the **constant?** the set of all well-typed expressions as its domain. This allows us greater scope for using this function in subsequent definitions. The simplest wff is an atom of propositional calculus which is a symbol that is not a constant.

```
(datatype wff

  if (not (constant? P))
  P : symbol;
  P : wff;
```

Next comes an atom of first-order logic which consists of a predicate followed by a list of terms.

```
  F : predicate; X : [term];
  [F  X] : wff;
```

It is tempting to then add the left version of this rule.

```
  F : predicate; X : [term] >> P;
  [F  X] : wff >> P;
```

But this rule is not right; for we want to be able to assume **[~ P]** is a wff without assuming that **~** is a predicate and **P** a list of terms! However if it is verified that **F** is a predicate, then assuming that **[F X]** is a wff, we can indeed assume **F** is a predicate and **X** a list of terms.

```
  F : predicate,  X : [term] >> P;
  (predicate? F) : verified,  [F X] : wff >> P;
```

An oddity, but we need it, is a rule that allows us to extort the conclusion that **F** is a symbol from the fact that **[F X]** is a wff.

```
 F : symbol >> P;
 [F  X] : wff >> P;
```

The other rules are quite straightforward.

```
 P : wff;
======
 [~ P] : wff;

 if (element? C [v & => <=>])
 P : wff; Q : wff;
 ===========
 [P C Q] : wff;

 if (element? Q [all some])
  V : variable, X : wff >> P;
  ===============
  [Q V X] : wff >> P;
```

Variables and predicates are basically the same type, distinguished by their occurrence in a wff. They are both symbols that are not constants.

```
(datatype predicate

 if (not (constant? X))
 X : symbol;
==========
 X : predicate;)

(datatype variable

 if (not (constant? X))
 X : symbol;
==========
 X : variable;)
```

A term is a symbol or a number or a symbol (designating a function) heading a list of terms.

```
(datatype term

 F : functor; T : [term];
 ================
 [F | T] : term;
```

```
  T : number;
  T : term;

  T : symbol;
  T : term; )

(datatype functor

  if (not (constant? X))
  X : symbol;
  =========
  X : functor;)
```

15.3 Coding the Quantifier Refinement Rules

The proof rules of the previous chapter are inherited by FOL. The extra machinery of FOL revolves around the use of the quantifiers. The ~∀ and ~∃ refinement rules are easy (figure 15.2), but the ∃>> and ∀>> refinement rules are trickier.

```
name ~some
[all X [~ Y]] >> P;
[~ [some X Y]] >> P;

name ~all
[all X [~ Y]] >> P;
[~ [some X Y]] >> P;
```

Figure 15.2 Two quantifier rules for FOL in Qi

The ∃>> rule eliminates the leading existential quantifier in an assumption and replaces the existentially bound variables by a fresh symbol. The representation of ∃>> in Qi is given in figure 15.3 (**gensym** of type **string** → **symbol** generates a fresh symbol from a string).

```
name some>>
let FTerm/X (replace_by X (gensym "term") Fx)
FTerm/X >> P;
[some X Fx] >> P;
```

Figure 15.3 ∃>> *in* Qi

15.3 introduces a new feature in the exposition of sequent rules; **let** is usable within a sequent rule to locally assign a value to a variable in the rule itself. The ∃>> rule enables the construction of an expression that contains a fresh symbol (generated by **gensym**). Naturally, we have also to state the definition of **replace_by** which is not a system function. Here it is in figure 15.4. Notice how **where (predicate? F) i**nteracts with the verification condition in our syntax for FOL.

```
(define replace_by
 {term - -> term - -> wff - -> wff}
 V1 _ [all V1 P] -> [all V1 P]
 V1 _ [some V1 P] -> [some V1 P]
 V1 V2 [all X Y] -> [all X (replace_by V1 V2 Y)]
 V1 V2 [some X Y] -> [some X (replace_by V1 V2 Y)]
 V1 V2 [P v Q] -> [(replace_by V1 V2 P) v (replace_by V1 V2 Q)]
 V1 V2 [P & Q] -> [(replace_by V1 V2 P) & (replace_by V1 V2 Q)]
 V1 V2 [P => Q] -> [(replace_by V1 V2 P) => (replace_by V1 V2 Q)]
 V1 V2 [P <=> Q] -> [(replace_by V1 V2 P) <=> (replace_by V1 V2 Q)]
 V1 V2 [~ P] -> [~ (replace_by V1 V2 P)]
 V1 V2 [F Terms] ->  [F (map (/. Term (replace_by* V1 V2 Term)) Terms)]
                    where (predicate? F)
 _ _ P -> P)

(define replace_by*
 {term - -> term - -> term - -> term}
 V1 V2 V1 -> V2
 V1 V2 [Func | Terms]
 -> [Func | (map (/. Term (replace_by* V1 V2 Term)) Terms)]
 _ _ Term -> Term)
```

Figure 15.4 Replacing a term in a wff

In order to apply the refinement rule for ∀, we need to be able to instantiate a universally bound variable with a term of our own choice. To do this we need to pass a parameter from outside the rule into the rule itself. We do this through invoking the special variable **Parameters** which is bound to a list of parameters that is, by default, empty. In this case we do not want the parameter list to be empty, but we expect the parameter list to be a one element list whose head is the instantiating term.

The type **parameter** is not defined for us. Like the type **wff** it is a place-holder waiting for the user to supply a definition. A synonym definition takes care of this: a parameter is just a term.

```
(synonyms parameter term)

name (all>> 1)
let FTerm/X (replace_by X (head Parameters) Fx)
FTerm/X, [all X Fx] >> P;
[all X Fx] >> P;
```

Figure 15.5 ∀>> *in* Qi

Since our rule is designed to receive a parameter, the name of the function is enclosed in round brackets along with a number indicating the number of parameters that the rule expects to receive. Qi will then automatically generate the function below.

```
(define all>>
  {parameter - -> goals - -> goals}
    P Goals -> (refine fol 12 [P] Goals))
```

15.4 A Simple Proof in First-Order Logic

Here is a script of a simple proof in FOL.[🗀]

```
Step 1. [1]                              0 refinements

> [mortal [socrates]]

1. [all x [[man [x]] => [mortal [x]]]]
2. [man [socrates]]

FOL: indirect-proof
______________________________________________________________
Step 2. [1]                              1 refinement

> [mortal [socrates]]

1. [~ [mortal [socrates]]]
2. [all x [[man [x]] => [mortal [x]]]]
3. [man [socrates]]

FOL: all>> socrates
```

[🗀] Qi Programs/Chap15/fol.qi

```
Step 3. [1]                     2 refinements

> [mortal [socrates]]

1. [[man [socrates]] => [mortal [socrates]]]
2. [all x [[man [x]] => [mortal [x]]]]
3. [~ [mortal [socrates]]]
4. [man [socrates]]

FOL: =>>>

Step 4. [1]                     3 refinements
_____________________________________________
> [mortal [socrates]]

1. [[~ [man [socrates]]] v [mortal [socrates]]]
2. [all x [[man [x]] => [mortal [x]]]]
3. [~ [mortal [socrates]]]
4. [man [socrates]]

FOL: v>>
_____________________________________________
Step 5. [2]                     4 refinements

> [mortal [socrates]]

1. [~ [man [socrates]]]
2. [all x [[man [x]] => [mortal [x]]]]
3. [~ [mortal [socrates]]]
4. [man [socrates]]

FOL: raa
_____________________________________________
Step 6. [1]                     5 refinements

> [mortal [socrates]]

1. [mortal [socrates]]
2. [all x [[man [x]] => [mortal [x]]]]
3. [~ [mortal [socrates]]]
4. [man [socrates]]

FOL: raa
0.06 secs, 6 refinements, 100 RPS
proved : symbol
```

15.5 Automating Proof in FOL

pc_decision_procedure is decidable, sound and complete for PC; that is, for any PC formula:-

1. **pc_decision_procedure** terminates with a proof of the formula, if a proof exists.
2. **pc_decision_procedure** terminates with a negative answer, if no proof exists.

By enriching **pc_decision_procedure,** we can derive a sound and complete technique to deal with FOL problems. However, from foundational work done by Church and Turing in the 1930s, we know that there is no decision procedure for FOL. The first proposition does hold of our extended technique, but the second proposition does not hold. Using the extended technique requires using **unification** and **skolemisation**. Unification was covered in chapter 13, so we will now deal with skolemisation.

15.6 Skolemisation

Skolemisation can be seen as an extension of the ∃>> refinement rule which allows the elimination of existential quantifiers. Skolemisation presumes that all quantified formulae are in **prenex form**; that is, any quantifier occurring in a formula occurs to the far left of the formula. In cases where the formula is not in prenex form, there are precise and mechanical rules for converting the formula to a logically equivalent formula in prenex form. We shall shortly cover these rules and the associated procedure of rectification that precedes conversion to prenex form. We assume for the present that all formulae are in prenex form.

The ∃>> refinement rule allows the elimination of existential quantifiers when the existential quantifier occurs at the very front of the formula in question. Skolemisation extends this treatment to allow the elimination of existential quantifiers even when these quantifiers are fronted by universal quantifiers. Before stating skolemisation formally, it is useful to see why it works.

Consider the statement; ∃x man(x). As one of a list of assumptions, this statement may be replaced by man(c) as long as c is fresh. The validity of this move can be justified by pointing out that if we know that something is a man, then there is no harm in stating “and let c be the man in question”, provided that no assumptions are smuggled in about c.

Now consider the statement; $\forall x \exists y$ father(y,x) which claims everything has a father. If this is true then there is a function which, given any input x, will produce a father of x as an output. In the same way that the unknown man was given a fresh name, we can give the unknown function a fresh name, say "f_0". The statement $\forall x \exists y$ father(y,x) can thus be replaced by $\forall x$ father((f_0 x),x). Both man(c) and $\forall x$ father((f_0 x),x) are the skolemised versions of $\exists x$ man(x) and $\forall x \exists y$ father(y,x) respectively.

The relations between a formula and its skolemised version are interesting; they are obviously not logically equivalent since the sequents $\exists x$ man(x) >> man(c) and $\forall x \exists y$ father(y,x) >> $\forall x$ father((f_0 x),x) are not provable in FOL. However what we do know is that given any set of FOL formulae Δ, that Δ is consistent if and only if its skolemised equivalent Δ' is, and this is good enough for the purpose of theorem-proving. Using our semantic tableau approach, we begin by negating the conclusion, adding the negation to the list of assumptions. Our original conclusion is proved if we can demonstrate this new list is inconsistent; and this list is inconsistent just when its skolemised version is inconsistent. The rules for skolemisation are:-

Skolemisation Rules

1. Rectify the formula and convert it to prenex form (see next section).
2. Eliminate each existential quantifier as follows:-
   a. If $\exists v$ is any existential quantifier together with its variable, determine the number *n* of universal quantifiers that precede it, and the variables $v_1,...,v_n$ that these universal quantifiers bind.
   b. For each v bound by $\exists v$, replace v by (f_{sk} $v_1,...,v_n$) where f_{sk} is some fresh symbol.
   c. Delete the $\exists v$.

The function f_{sk} is a **skolem function**. If f_{sk} has an arity of 0, then f_{sk} is a name and is then referred to as a **skolem constant**. Skolem constants are generally written without parentheses (so instead of writing (f_{sk}) we just write f_{sk}).

Wff	*Skolem Form*
$\exists x \forall y$ R(x,y)	$\forall y$ R(c, y)
$\forall x \exists y \exists z$ F(z) & F(y)	$\forall x$ F(f_1 x) & F(f_2 x)
$\forall w \forall x \exists y \exists z$ R(w,x,y,z)	$\forall w \forall x$ R(w,x,(f_1 w x)(f_2 w x))

Figure 15.6 Some wffs and their Skolem forms

15.7 Prenex Form

Skolemisation presupposes that all wffs have first (a) been rectified and then (b) been placed in prenex form; that is to say, all quantifiers must occur to the left of the formula. The following rules, if applied to a fixpoint, will generate a prenex form of the wff..[🗀]

1. $\sim\forall x P \Rightarrow \exists x \sim P$
2. $\sim\exists x P \Rightarrow \forall x \sim P$
3. $(\forall x\ P)\ \&\ Q \Rightarrow \forall x(P\ \&\ Q)$
4. $(\exists x\ P)\ \&\ Q \Rightarrow \exists x(P\ \&\ Q)$
5. $P\ \&\ (\forall x\ Q) \Rightarrow \forall x(P\ \&\ Q)$
6. $P\ \&\ (\exists x\ Q) \Rightarrow \exists x(P\ \&\ Q)$
7. $(\forall x\ P)\ v\ Q \Rightarrow \forall x(P\ v\ Q)$
8. $(\exists x\ P)\ v\ Q \Rightarrow \exists x(P\ v\ Q)$
9. $(P\ v\ (\forall x\ Q) \Rightarrow \forall x(P\ v\ Q)$
10. $(P\ v\ (\exists x\ Q)) \Rightarrow \exists x(P\ v\ Q)$
11. $P \rightarrow Q \Rightarrow \sim P\ v\ Q$
12. $P \leftrightarrow Q \Rightarrow (P \rightarrow Q)\ \&\ (Q \rightarrow P)$
13. $P\ v\ (Q\ \&\ R) \Rightarrow (P\ v\ Q)\ \&\ (P\ v\ R)$
14. $(Q\ \&\ R)\ v\ P \Rightarrow (P\ v\ Q)\ \&\ (P\ v\ R)$
15. $\sim(P\ \&\ Q) \Rightarrow \sim P\ v \sim Q$
16. $\sim(P\ v\ Q) \Rightarrow \sim P\ \&\ \sim Q$
17. $\sim\sim P \Rightarrow P$

15.8 Rectification

A formula is **rectified** when no two distinct quantifiers bind typographically identical variables. For example, the wff $\forall x\ F(x)\ \&\ \exists x\ G(x)$ is not rectified since the variable x is bound by two different quantifiers. Unrectified formulae pose problems to the process of conversion to prenex form since applying the above rules to $\forall x\ F(x)\ \&\ \exists x\ G(x)$ gives the formula $\forall x \exists x\ F(x)\ \&\ G(x)$ and it is no longer clear exactly what quantifier binds what variable. The solution is to rectify the wff before applying the prenex rules so that $\forall x$

[🗀] See Qi Programs/Chap15/cnf.qi for the code for skolemisation, including conversion to prenex form.

$F(x) \ \& \ \exists x\ G(x)$ becomes (for example) $\forall x\ F(x) \ \& \ \exists y\ G(y)$.[63] Figure 15.7 shows a complete example, which includes rectification, conversion to prenex form, and then skolemisation.

	Wff	*Justification*
1.	$\sim\exists x\ F(x) \ \& \sim\forall x\ (F(x) \leftrightarrow G(x))$	_
2.	$\sim\exists x\ F(x) \ \& \sim\forall y\ (F(y) \leftrightarrow G(y))$	rectification
3.	$\forall x \sim F(x) \ \& \sim\forall y\ F(y) \leftrightarrow G(y))$	prenex rule 2
4.	$\forall x \sim F(x) \ \& \ \exists y \sim(F(y) \leftrightarrow G(y))$	prenex rule 1
5.	$\forall x\exists y \sim F(x) \ \& \sim(F(y) \leftrightarrow G(y)))$	prenex rule 6
6.	$\forall x \sim F(x) \ \& \sim(F((f_0\ x)) \leftrightarrow G((f_0\ x)))$	skolemisation

Figure 15.7 Converting a Wff to Skolem form

Skolemisation can proceed in step 6 because a formula in prenex form has been derived. It would be possible to apply further prenex form rules to the formula in step 5, before skolemising the result. This would derive a different skolemised formula. The consistency properties of skolemisation are maintained irrespective of what skolemised formula is used. However, prenex formulae that are derived by applying all the prenex rules to generate a fixpoint are logically significant as we shall see shortly.

15.9 Universal Quantifiers and Unification

Skolemisation leaves behind the universal quantifiers, so that $\forall x\exists y$ father_of(x,y) becomes $\forall x$ father_of(x, (f0 x)). The next step is to simply throw away the remaining universal quantifiers and rely on unification, treating all variables as variables within unification.

$\forall x$ father_of(x, (f0 x)) then becomes father_of(X, (f0 X)). The variable X becomes a proxy for any term we care to mention since the X can be instantiated by any term according to the rule for all>>. However, instead of choosing a term, we allow father_of(X, (f0 X)) to contradict any formula that arises from negating father_of(X, (f0 X)) and replacing the X by a term. Unification is used to determine if such a contradiction obtains.

The general strategy is to apply skolemisation, throw away the universal quantifiers to generate formulae that are free of quantifiers and treat all

[63] Some prenex rules (12, 13 and 14) can convert a rectified formula into an unrectified one; these duplicate a variable found once on the left-hand-side of the rule. It is necessary to rerectify the whole formula after these rules are applied.

variables as variables within unification. From then on, apply propositional calculus techniques to solve FOL problems except that contradictions obtain if there are two formulae in one context, A and ~B, such that A and B unify. A and ~B are then known as **complements**.

15.10 Skolemisation in Sequent Systems

Skolemisation in tableau has one extra complexity. If we are skolemising a conclusion rather than an assumption then the rules for dealing with ∀ and ∃ are transposed. That is to say the conclusion ∃x f(x) does not, after skolemisation and quantifier elimination reduce to f(a), where a is any skolem constant, but rather f(X). A conclusion ∀xf(x) does not, after skolemisation and quantifier elimination reduce to f(X) but to $f(f_{sk})$, where f_{sk} is any skolem constant.

The reason why is clear after a little reflection. We should get the same results from skolemising the conclusion and then applying indirect proof, as we would from applying indirect proof and then applying skolemisation. But suppose we were to apply indirect proof to ∃xf(x); in that case we would derive ~∃xf(x) as a hypothesis. However, ~∃x f(x) is equivalent to ∀x ~f(x), which after skolemisation and quantifier elimination, reduces to ~f(X). If we apply our skolemisation rules first without taking care, we skolemise ∃xf(x) to $f(f_{sk})$ and then apply indirect proof to derive $\sim f(f_{sk})$ - a very different result!

There are two ways round this problem. To skolemise formulae after >>, after conversion to prenex form we must invert the rules for eliminating existential and universal quantifiers treating ∀ as ∃ and vice-versa. In semantic tableau systems, there is another solution; we retain the skolemisation rules in their original form and bar any application of skolemisation to the conclusion of the sequent. Since the conclusion of the sequent plays no part in the tableau proof after the first step of indirect proof, nothing is lost in restricting skolemisation in this way.

15.11 Clause Form

In skolemising a wff using the rules for conversion to prenex form, it is not always necessary to apply these rules to a fixpoint in order to derive a prenex form. However, if we do so, and then eliminate the universal quantifiers in the usual way, we derive a formula which is in **conjunctive normal form** (CNF). The formal definition of conjunctive normal form is as follows.

Formation rules for Conjunctive Normal Form

1. An atomic wff is a literal.
2. The negation of an atomic wff is a literal.
3. If A and B are literals then (A v B) is a clause
4. If A and B are clauses then (A v B) is a clause.
5. If A is a clause then A is in CNF.
6. If A and B are clauses then (A & B) is in CNF.
7. If A and B are in CNF then (A & B) is in CNF.

The conversion of wffs to conjunctive normal form allows a significant gain in the efficiency with which proofs are conducted in FOL. After the first step of **indirect-proof**, conversion to CNF removes the necessity to use any refinement rules apart from **&>>**, **v>>** and **raa** to drive the proof process. We can go a step further, by applying **&>>** to a fixpoint all occurrences of **&** are driven from assumptions of the sequent and the proof can then be driven by two refinement rules alone - **v**>> and **raa**. Effectively this process, called **conversion to clause form**, allows us to think of a FOL sequent as a list of clauses.

15.12 The Refinement Rules for AUTOFOL

The refinement rules of our automated version of FOL we call "AUTOFOL". AUTOFOL includes **indirect-proof**, **v>>** and **&>>** as in PC. But in place of the other rules of PC, we have a single function that turns all wffs into CNF. This eliminates the →s and ↔s as well and eliminates the need for rules that mention → and ↔, ∀ or ∃.

To encode the new proof procedure in Qi, we define a new refinement rule that enables us to convert each assumption of the list of assumptions into CNF. Rebinding the special variable **Assumptions** allows us to globally change all the assumptions to CNF simultaneously (figure 15.8).

```
name conjunctive-normal-form
let Assumptions (map cnf Assumptions)
P;_
P;
```

Figure 15.8 Using a special variable to change the assumptions in a sequent

If we suppose that all the rules of PC are rules of FOL, we can now define the function **clause_form** that converts a problem into a list of clauses.

```
(define clause_form
  {goals - -> goals}
  Goals -> (fix &>>
              (conjunctive-normal-form (indirect-proof Goals))))
```

One rule is still missing; the rule **raa** that allows goals to be eliminated. But the **raa** rule for AUTOFOL operates with unification rather than simple equality. To handle this we need to understand the use of **notes**.

15.13 Representing Bindings with the Notepad

We now approach an important feature of Qi, which is the use of the *notepad*. The notepad is a list of elements, called *notes,* which can be carried with the ongoing proof. The special variable **Notes** is bound to the contents of the notepad.

The notepad is carried dynamically with the proof, so that if some notes are added to a proof and then backtracking occurs, the notepad returns to its state as it existed before these notes were added. This means that the notepad can be used to hold auxiliary information about the state of a proof. The nature of this auxiliary information is not fixed, because notes can be anything the user desires, and their contents are displayed by the proof tool. There is an important proviso however; which is that there is a type **note** that the user must define if notes are to be used effectively. For instance, if we wish to insert notes into wffs, then our rules of **wff** and **note** must harmonise to enable us to do it. Unless the user adds notes to the proof, the notepad remains empty.

The use of the notepad is essential for the automation of first-order reasoning in Qi and we will see it is used to hold variable-value associations. When unification occurs, variables are bound and some record of these associations must be maintained, so we *note* these associations by making notes that are pairs of terms.

```
(synonym note (term * term))
```

Using this type **note**, we create a program that unifies literals returning a list of notes as the MGU. Our new version of **raa** interprets the test for complements through the unification of literals using the contents of the notepad as the list of bindings. If the unification succeeds, then the MGU becomes the notepad. (figure 15.9). The variable **Notes** is always bound to the current notepad.

```
name raa+
let MGU (unify_literals P Q Notes)
if (unification_succeeds? MGU)
let Notes MGU
_____________
[~ P], Q >> R;
```

Figure 15.9 The sequent rule **raa+**

The definition of unification follows.

```
(define unify_literals
 {wff - -> wff - -> [note] - -> [note]}
[F1 Terms]  [F2 Terms*] MGU
 -> (unify_terms (deref_term Terms MGU)  (deref_term Terms* MGU) MGU)
          where (and (= F1 F2) (and (predicate? F1) (predicate? F2)))
 _ _ _ -> [(@p no mgu)])

(define unify_terms
 {[term] - -> [term] - -> [note] - -> [note]}
 Terms Terms MGU -> MGU
 [Term1 | Terms] [Term2 | Terms*] MGU
  -> (let MGU* (unify_term Term1 Term2 MGU)
             (if (unification_succeeds? MGU*)
                (unify_terms (deref_term Terms MGU*)
                               (deref_term Terms* MGU*) MGU*)
                MGU*))
 _ _ _ -> [(@p no mgu)])

(define unification_succeeds?
 {[note] - -> boolean}
 [(@p no mgu)] -> false
 _ -> true)

(define unify_term
 {term - -> term - -> [note] - -> [note]}
 Term Term MGU -> MGU
 Term Term* MGU -> [(@p Term Term*) | MGU]
   where (and (variable? Term) (= (occurrences Term Term*) 0))
 Term Term* MGU -> [(@p Term* Term) | MGU]
          where (and (variable? Term*)  (= (occurrences Term* Term) 0))
 [Functor | Terms] [Functor | Terms*] MGU
          -> (unify_terms Terms Terms* MGU)
 _ _ _ -> [(@p no mgu)])
```

```
(define deref_term
  {[term] - -> [note] - -> [term]}
   [] _ -> []
   [Term | Terms] MGU -> [(deref* Term MGU) | Terms])

(define deref*
  {term - -> [note] - -> term}
  [Functor | Terms] MGU -> [Functor | (map (/. X (deref* X MGU)) Terms)]
  Atom MGU -> (let NewTerm (fetch_value Atom MGU)
                     (if (= Atom NewTerm)  Atom (deref* NewTerm MGU))))

(define fetch_value
  {term - -> [note] - -> term}
  Atom [] -> Atom
  Atom [(@p Atom Value) | _] -> Value
  Atom [_ | MGU] -> (fetch_value Atom MGU))
```

Figure 15.10 The code for unification

15.14 Some AUTOFOL Proofs

The proof begins by converting the problem into a list of clauses; **f_7507** is a skolem constant and **X7508** and **X7509** are variables.

```
Step 1. [1]                    0 refinements

> [some x [~ [f [x]]]]

1. [all x [[f [x] => [[g [x]] v [h [x]]]]]
2. [some x [[~ [g [x]]] & [~ [h [x]]]]]

AUTOFOL: clause_form
________________________________________________
Step 2. [1]                    13 refinements

> [some x [~ [f [x]]]]

1. [~ [g [f_7507]]]
2. [~ [h [f_7507]]]
3. [f [X7508]]
4. [[~ [f [X7509]]] v [[g [X7509]] v [h [X7509]]]]

AUTOFOL: v>>
```

The preceding step and the next one splits the disjunction and applies **raa+.**

Step 3. [2]   14 refinements

```
> [some x [~ [f x]]]

1. [~ [f [X7509]] ]
2. [~ [g [f_7507]]]
3. [~ [h [f_7507]]]
4. [f [X7508]]

AUTOFOL: raa+
```

The effect of **raa+** is to bind **X7509** to **X7508** and this binding is printed.

Step 4. [1]   15 refinements

```
Notes: (@p X7509 X7508),

> [some x (~ (f x]]

1. [[g [X7509]] v [h [X7509]]]
2. [~ [g [f_7507]]]
3. [~ [h [f_7507]]]
4. [f [X7508]]

AUTOFOL: : v>>
```

Step 5. [2]   16 refinements

```
Notes: (@p X7509 X7508),

> [some x [~ [f x]]]

1. [g [X7509]]
2. [~ [g [f_7507]]]
3. [~ [h [f_7507]]]
4. [f [X7508]]

AUTOFOL: raa+
```

X7509 is bound to **X7508** and **(g (X7509))** unifies with **(g (f_7507))** binding **X7508** to **f_7507.**

```
Step 6. [1]                    17 refinements

Notes: (@p X7508 f_7507), (@p X7509 X7508),

> [some x [~ [f [x]]]]

1. [h [X7509]]
2. [~ [g [f_7507]]]
3. [~ [h [f_7507]]]
4. [f [X7508]]

AUTOFOL: raa+
0.11 secs, 18 refinements, 164 RPS
proved
```

By converting everything to clause form, and then applying **raa+** and **v>>** to a fixpoint, we generate a theorem-prover that (when it works) solves a FOL problem in one step (figure 15.11).🗀

```
Step 1. [1]                    0 refinements

> [some x [~ [f [x]]]]

1. [all x [[f [x]] => [[g [x]] v [h [x]]]]]
2. [some x [[~ [g [x]]] & [~ [h [x]]]]]

AUTOFOL: autofol
0.02 secs, 20 refinements, 1000 RPS
proved : symbol
```

Figure 15.11   Automating the proof of first order problems

15.15 The Incompleteness of AUTOFOL

Our **autofol** tactic is sound but incomplete. There are theorems which of FOL which it will not prove and some are very simple. For example, ∀x f(x) >> f(a) & f(b) is a theorem of FOL; but **autofol** will not prove it. It is instructive to see this; here is **autofol** failing in full view (figure 15.12).

Instead of returning **proved**, **AUTOFOL** returns an insoluble sequent. The problem is we have assumed that universally quantified formulae will be instantiated in only one way. But in the above case, we need to instantiate

🗀Qi Programs/Chap15/autofol version 1.qi the universally bound variable by first **a** and then **b** to derive a proof. Having created only one clause, we are not free to do this. If we had two, say **(f (X7583))** and **(f (X7584))**, then the proof would go through.

```
Step 1. [1]                    0 refinements

> [[f [a]] & [f [b]]]

1. [all x [f [x]]]

AUTOFOL: autofol
______________________________________________
Step 2. [1]                    17 refinements

Notes: (@p X7583 a),

> [[f [a]] & [f [b]]]

1. [~ [f [b]]]
2. [f [X7583]]

AUTOFOL:
```

Figure 15.12 A Blocked Proof in AUTOFOL

The solution is fairly obvious, create two clauses differing only in their free variables; a process we can call **amplification**. The problem with amplification is that we have no way of knowing in advance how many copies (i.e. the level of amplification) we should use. We know that any FOL problem that is solvable, is solvable at some finite level of amplification; (see **Fitting** (1990)) but it is also known that there is no decision procedure for determining what this level is. Often the best approach is to iteratively increase the level of amplification until a solution is found - or the computer memory or our patience runs out! Even worse, problems run at high levels of amplification take longer to solve since more clauses are generated.

Would **autofol** be complete if we just iteratively increased amplification? No; there is another source of incompleteness in **autofol**, which relates to the **raa+** refinement rule. Qi will apply this refinement rule to two wffs of the form P and (~ P*), where P and P* are unifiable.

However, choices exist when there is more than one pair of complements and the wrong choice can lead to a blocked proof. Figure 15.13 shows an example.

```
Step 1. [1]                        0 refinements

> [some x [[f [x]] & [g [x]]]]

1. [f [a]]
2. [f [b]]
3. [g [b]]

AUTOFOL: clause_form
________________________________________________
Step 2. [1]                        1 refinement

> [some x [[f [x]] & [g [x]]]]

1. [[~ [f X7641]] v [~ [g X7641]]]
2. [f [a]]
3. [f [b]]
4. [g [b]]

AUTOFOL: v>>
________________________________________________
Step 3. [2]                        2 refinements

> [some x [[f [x]] & [g [x]]]]

1. [~ [f X7641]]
2. [f [a]]
3. [f [b]]
4. [g [b]]

AUTOFOL: raa+
________________________________________________
Step 4. [1]                        3 refinements

Notes: (@p X7641 a),

> [some x [[f [x]] & [g [x]]]]

1. [~ [g X7641]]
2. [f [a]]
3. [f [b]]
4. [g [b]]

AUTOFOL:
```

Figure 15.13 Another Blocked Proof in AUTOFOL

The proof cannot be advanced past this point and **AUTOFOL** gets stopped right here. The attempted proof goes wrong in Step 3. The invocation of **raa+** in that step unifies **(f (X7641))** with **(f (a))** whereas the correct step unifies **(f (X7641))** with **(f (b)).** To discharge the proof, we need to swap assumptions 2 and 3 of step 3 before applying **raa+**. This proof shows a feature of refinement rules that we saw in chapter 7, namely that a refinement rule can sometimes be legally applied to the same problem in more than one way.

The remedy for the incompleteness of the proof procedure is clear. We must arrange for the program to consider all possible unifiers for contradictions by backtracking when necessary and combine this with a search process that iteratively increases amplification after every failed proof attempt. We will then have a sound and complete tactic for solving FOL problems - albeit one that will not halt when presented with an insoluble problem. We shall now see how to implement such a procedure in Qi.🗀

15.16 A Complete AUTOFOL

In the preceding sections, we saw that AUTOFOL was incomplete for two reasons.

1. It may be necessary to try higher levels of amplification to derive a proof.
2. Choosing the wrong complements to unify may block a proof.

By factoring in these points into our proof procedure, we can create a sound and complete theorem-prover for FOL🗀 (figure 15.14).

Do until a proof is found

1. Convert the problem into a list of clauses.
2. Apply **v>>** to a fixpoint to generate a list of literals.
3. Generate a non-repeating list of all the goals that can legally be generated by **raa+**.

🗀 Qi Programs/Chap15/fol prbs.qi contains many of the Pelletier problems. Load this program to see how well **autofol** performs. Despite its incompleteness, **autofol** does quite well on many of these problems, solving about 70% of them.

🗀 Qi Programs/Chap15/autofol version 2.qi

4. If the list is empty, then backtrack to the last choice point, and if there is no choice point to return to, then increment the level of amplification and return to step 1.
5. Otherwise, choose the first of this list of goals and return to step 2.

Figure 15.14 A Sound and Complete Proof Procedure for FOL

A simple way of representing amplification is to increment an integer value *n* where *n* indicates the amplification level of the proof. In the worst case, where there is no proof to be found; the program will endlessly increment *n*.

The second source of incompleteness stems from the indeterminism of the **raa+** refinement rule. Qi has a special function **collect** which allows non-deterministic refinement rules to be cleanly handled. This function receives

(a) A symbol *S*.
(b) An integer *n*.
(c) A (possibly empty) list of objects of type **parameter**.
(d) Goals *G*

and returns a non-repeating list of all the goals that can be generated from legally applying the *n*th refinement rule of the theory named by *S* to the goals *G* (if the rule cannot be applied, then this list will be empty). So, given that **raa+** is the first rule of the theory **autofol**, then **(collect autofol 1** *G***)** will return all the series of goals that can be generated by applying **raa+**.

To encode the new proof procedure in Qi, we define a new refinement rule **amplify** that enables us to amplify or duplicate universally quantified assumptions throughout the list of assumptions. The extent of the amplification will be determined by a global variable ***amplification*** which will hold an integer value. The main routine of the theorem-prover can now be coded (figure 15.15).

The program works by calling **autofolc** that passes control to an outer loop with a numeric parameter indicating the level of amplification, which is initially set to zero. The outer loop tries to solve the problem, incrementing the level after every failed proof attempt.

The function **try_to_prove** attempts to solve the problem by first applying **v>>** to a fixpoint - reducing the first goal to a list of literals and then generating the list of possible goals from applying **raa+** (which is rule #1 of the theory **autofol**). This sets up a mutual recursion with the function **inner_loop**.

The **inner_loop** function first tests the goals to see if the problem has been solved. This test was missing from the PC theorem prover, which applied its terminating proof procedure to a fixpoint. If the goals were solved in the PC program then **proved** would be returned as soon as control surfaced back to the proof tool. But here the proof procedure will not terminate unless a check is inserted into the program to determine whether it should be halted. The function **solved?** has the type **goals** → **boolean** and returns **true** just when there are no more goals to solve. Should this condition not apply, **inner_loop** selects each series of goals from a list and then **try**(s)**_to_prove** it. The first-order theorems above can be proved by our new ATP.[□]

```
(define autofolc
  {goals - -> goals}
  Goals -> (do (output "Amplification = 0~%")
               (set *amplification* 0)
               (outer_loop Goals)))

(define outer_loop
  {goals - -> goals}
  Goals <- (fail-if (/. X (not (solved? X)))
                       (try_to_prove (clause_form Goals)))
  Goals -> (do (output "Amplification = ~A~%"
               (set *amplification* (+ 1 (value *amplification*))))
               (outer_loop Goals)))

(define try_to_prove
  {goals - -> goals}
  Goals -> (let SubgoalsList   (collect autofol 1 [] (fix v>> Goals))
              (if (empty? SubgoalsList)
                 Goals
                 (inner_loop (head SubgoalsList) (tail SubgoalsList)))))
```

[□] **autofolc** can be run on the file Qi Programs/Chap15/fol II prbs.qi. (just load this file) containing the Pelletier problems. It solves many of them. However the more difficult ones have been commented out as beyond **autofolc**. We leave the reader to experiment with the techniques in section 15.17 on these difficult problems.

```
(define inner_loop
  {goals - -> [goals] - -> goals}
  Goals _ -> Goals              where (solved? Goals)
  Goals _ <- (fail-if (/. X (not (solved? X))) (try_to_prove Goals))
  Goals [ ] -> Goals
  _ [Goals ¦ Collection] -> (inner_loop Goals Collection)))
```

Figure 15.14 A Complete Theorem Prover for FOL

15.17 Optimising AUTOFOLC

AUTOFOLC is a very inefficient complete ATP for FOL. Several techniques improve performance. We shall focus on five techniques that are effective with AUTOFOLC and leave it to the reader to experiment with them.

1. Eliminating tautologies.
2. Subsumption.
3. Using >>**v** on small clauses.
4. Recognising explicit complements.
5. The use of negation normal form.

1. Eliminating Tautologies

Applied to clauses, a **tautology** is any clause in which a literal A occurs and so does the literal ~A. Tautologies are logically true and add nothing to the information content of any set of assumptions in which they appear. Consequently removing them from the assumptions does not affect the ability of the semantic tableau ATP to derive a proof. For example, the problem **(p ↔ q) >> (q ↔ p)** converted to clause form in Qi generates the sequent in figure 15.15. Of the assumptions in figure 15.15, 3 and 6 are tautologies and can be eliminated.

```
> [q <=> p]

1. [[~ p] v q]
2. [[~ q] v p]
3. [[~ q] v q]
4. [[~ q] v [~ p]]
5. [p v q]
6. [p v [~ p]]
```

Figure 15.15 A series of assumptions including two tautologies

2. Subsumption

A clause C **propositionally subsumes** another clause C′ if every literal in C is in C′. In such a case C′ may be discarded from the proof. For example, in figure 15.16, clause 1 propositionally subsumes clause 3, showing that clause 3 can be discarded from the tableau proof.

```
> p

1. [~ p]
2. [[~ q] v p]
3. [[~ q] v [~ p]]
4. [p v q]
```

Figure 15.16 A candidate for subsumption

Subsumption can be extended to embrace first-order logic. A clause C then subsumes a clause C′ if there is an assignment σ of values to the variables of C such that σ(C) propositionally subsumes C′. Thus the clause ((f (X)) v (g (Y))) subsumes ((f (a)) v (g (b))) where $\sigma = \{X \mapsto a, Y \mapsto b\}$. In vanilla resolution[64], extended subsumption can be freely applied to eliminate subsumed clauses without loss of completeness. In tableau, the use of amplification generates clauses that do subsume each other in the extended manner, but which cannot, for obvious reasons, be simply discarded. Propositional subsumption can be freely used in AUTOFOLC. Subsumption is one of the most powerful techniques for improving the performance of clause-based reasoning systems.

3. Using v>> on Small Clauses

The **v>>** rule causes the proof tree to branch, but it is not fixed which clause should be split by this rule. A good heuristic suggests that splitting on the smallest clause must often be sensible since small clauses tend to be good at subsuming larger ones and this helps to reduce the search. In terms of Qi, this optimisation technique is best accomplished by using **swap** to sort the assumptions into ascending size depending on the number of literals in each clause. Since Qi will branch from the front, performing this sort before running the tableau proof procedure will guarantee that the smallest clause will always used. This technique works well in conjunction with the next technique discussed.

[64] Another clause-based technique for solving FOL problems. See Chang and Lee (1973) for a good description.

4. Explicit Complements

Two complements are **explicit** if no new bindings are made in the course of unifying them; i.e. essentially we have a propositional contradiction. Two complements are **implicit** otherwise. Explicit complements do not represent choice points in the proof, since no variables are bound. Recognising that a successful application of **raa+** involves only explicit complements removes the need for unnecessary backtracking. The detection of explicit complements should actually be carried out in preference to any other operation - including **v>>**, since the goal is then pruned from the stack by **raa+** before it proliferates redundant subgoals. The application of **v>>** to small clauses maximises the number of literals in the goal and hence maximises the chances of finding explicit complements.

5. Using Negation Normal Form

An interesting variation on semantic tableau theorem-proving uses **negation normal form** (NNF) instead of clause form. NNF is the form derived from omitting the distributive laws from our rules for conversion to prenex form; namely:-

$$P \text{ v } (Q \,\&\, R) \Rightarrow (P \text{ v } Q) \,\&\, (P \text{ v } R)$$
$$(Q \,\&\, R) \text{ v } P \Rightarrow (P \text{ v } Q) \,\&\, (P \text{ v } R)$$

This approach requires reinstating the **&>>** rule throughout the conduct of the proof. The advantage of using NNF is that the distributive laws generate very large numbers of clauses when &s are deeply embedded within **v**s (see **Eder** (1985) for a study of this problem). **Beckert** and **Posegga** (1997) use NNF to advantage in their leanT^{A}P semantic tableau theorem prover which contains some interesting insights about how to control the **v>>** rule.

Exercise 15

1. Using the proof tool (but not using autofolc) prove the following, translating these problems into Qi input.
   a. ∀x (F(x) → ~G(x)), ∀x(H(x) → G(x)) |- ∀x F(x) → ~(H(x))
   b. ∀x (F(x) v G(x)) → H(x), ∀x ~(H(x)) |- ∀x ~F(x)
   c. ∀x(G(x) → ~H(x)), ∃x(F(x) & G(x)) |- ∃x F(x) & ~(H (x))
   d. ∀x(F(x) v G(x)) → H(x), ∃x~(H(x)) |- ∃x~(F (x))
   e. ∀x∀y∀z F(x,y,z) |- ∀z∀y∀x F(x,y,z)
   f. ∀x∃y∀zF(x,y,z) |- ∀x∀z∃yF(x,y,z)
   g. ∃x∃y∀z F(x,y,z) |- ∀z∃y∃x F(x,y,z)

h. $|- \exists y \exists x(F(x)) \rightarrow (F(y)))$
i. $|- \forall x \exists y(R(x,y) \rightarrow \exists x \exists y R(x,y))$
j. $R(x,x), \forall x \forall y \forall z((R(x,y) \& R(y,z)) \rightarrow R(x,z))$
   $|- \forall x \forall y(R(x,y) \rightarrow {\sim}(R(y,x)))$
k. $|- \exists y(\exists x(F(x)) \rightarrow (F(y)))$
l. $|- \forall x \exists y R(x,y) \rightarrow \exists x \exists y R(x,y)$

2. *Implement the refinements discussed in the final section of this chapter. Write a paper "Refinement Techniques in Tableau Theorem Proving" in which you test the effectiveness of these refinements.

3. We defined the type operator list so that x : (list A) if and only if every element of x inhabited A. Formally, we can write this as $\forall x \forall y(\text{inhabits}(x, (\text{list } y))) \leftrightarrow (\exists z (\text{in}(z,x)) \rightarrow (\text{inhabits}(z,y)))$. Given ${\sim}\exists x(\text{in}(x,[\,]))$, explain in terms of FOL why [ ] has the type $\forall x$ (list x).

4. Skolemise the following formulae.
   $\exists x(Fx \& \forall y(G(y) \rightarrow H(x,y)))$
   $\forall x \exists y(I(y,x,m) \& G(y,m))$
   $\exists x \exists y \forall z\ F(x,y,z)$
   $\forall x \exists y(R(x,y)) \rightarrow \exists x \exists y(R(x,y))$
   $\exists x(F(x) \rightarrow G(x)) \rightarrow (\forall x\ (F(x)) \rightarrow \exists y(G(y)))$

5. Convert each of the following into a list of clauses:
   a. $(r \rightarrow p) \rightarrow (r \rightarrow q)$
   b. $q \rightarrow (q \& r)$
   c. $((q \vee (p \vee r)) \vee p$
   d. $(r \rightarrow p) \rightarrow ((p \rightarrow q) \rightarrow (r \rightarrow q))$
   e. $\forall x(H(x) \rightarrow G(x))$
   f. $\forall x\ (F(x) \vee G(x)) \rightarrow H(x))$
   g. $\exists x\ (F(x) \& G(x))$
   h. $\forall x \forall y \forall z\ F(x,y,z)$
   i. $\exists y \exists x(F(x)) \rightarrow (F(y)))$
   j. $\forall x \exists y(R(x,y) \rightarrow \exists x \exists y R(x,y))$
   k. $\forall x \forall y \forall z((R(x,y) \& R(y,z)) \rightarrow R(x,z))$

Further Reading

There are many introductions on FOL including **Hodges** (1977) which is based on (non-automated) tableau. For a natural deduction treatment see **Lemmon** (1978); and **Mendelson** (1987) for a Hilbert (or axiomatic) approach.

Skolemisation was introduced by Thoralf Skolem and the consistency properties of skolemisation and the proof of convertibility to prenex form are shown in **Chang and Lee** (1973). The prenex rules in Chang and Lee are a little subtler than the ones given here. **Stickel** (1986) introduces some work done since Chang and Lee -

the most important advances being those in high-performance theorem proving and many-sorted unification (**Walther** (1988)) on the latter).

Smullyan (1968) expounds first-order logic using tableau, and **Fitting** (1990) deals with tableau using the techniques described here and provides Prolog code for implementing it. **Beckert** and **Posegga** (1997) implement a high-performance Prolog version.

Boyer and **Moore** (1979) describe the Boyer-Moore theorem prover which handles inductive proofs of formulae written as Lisp expressions. **Boyer** and **Moore** (1997) describes a more advanced version of their theorem-prover called "NQTHM" (New Quantified Theorem Prover) which includes facilities for handling problems in linear arithmetic; see **Bundy** (1983) for details on this procedure. NQTHM has racked up some impressive successes, most notably the verification of a high-level computer language right down to the gate level design of the microprocessor (See Journal of Automated Reasoning 1989, vol. 5.).

Web Sites

There are currently a great number of automated theorem proving systems developed or under development throughout the world. Some of the main sites are given here and the reader can explore the links.The **Argonne National Laboratory** (http://www-fp.mcs.anl.gov/division/research/ar_summary.htm) is a world research centre for automated reasoning and is responsible for providing OTTER - a powerful clause-based theorem-prover based on resolution. A list of papers and reports produced by **Computational Logic Incorporated** (the company headed by Boyer and Moore for the industrial application of their work) dating from 1986 to 1997 is obtainable over the web (http://www.cli.com). The **Mathematical Reasoning Group** (http://dream.dai.ed.ac.uk) at Edinburgh is a centre of excellence for automated reasoning.

16 Expert Systems

An expert system is a program that takes on the tasks normally associated with human experts; medical diagnosis, prospecting, computer configuration, to name but a few, are all areas where computers have been successfully applied to produce expert systems. This is a big field in which a lot of work has been done. This chapter shows how Qi can be used as an **expert system shell**; that is, an environment for rapidly producing expert systems.

Our expert system is derived from **Sell** (1985); it is a simple rule-based system that diagnoses diseases based on symptoms. Sell's rules have the form (if *disease* then *symptoms*) or (if *disease* then *recommendation*). The task of the system is to diagnose a disease from the symptoms and produce a recommendation of the proper treatment.

16.1 Diagnosis as Explanation

Diagnoses are a subclass of the class of explanations; a diagnosis is just a medical explanation of a set of symptoms. The process of producing explanations is called **abduction**. The classical rule for abduction is:-

Given that you know that *If P then Q;* given *Q* to explain, produce *P* as an explanation.

Abduction works in reverse to a well-known logical rule called **modus ponens**;

Given that you know that *If P then Q;* given *P*, derive *Q*.

Not all explanations that proceed according to abduction would qualify as satisfactory. For example, we know that for all *P*, *If P then P* is true, but *P* is not a satisfactory explanation of *P*.[65] In addition to abduction, we require

[65] This form of explanation is sometimes produced by harassed parents coping with tiresome questions; e.g. Question: "But ***why*** is the sky blue?" Answer: "Because it is".

an **explanation function** whose task it is to rate an explanation as satisfactory or unsatisfactory. People adopt different explanation functions; for example a patient suffering from stomach pain may have his pain diagnosed as spastic colon by one doctor who will be satisfied with that diagnosis. A therapist or nutritionist might agree with this diagnosis, but not find it satisfactory; wanting to understand the reasons for the spastic colon from the point of view of the patient's lifestyle.

So rather than plunge straight into coding a medical diagnosis program, what we want to do is first code a general-purpose explanation program which can be specialised to the needs of our medical diagnosis program. In the next section we shall see how to express these insights in computational terms.[66]

16.2 Explanation and Proof Trees

A proof tree can be viewed as an attempt to provide explanations for the goal that triggers its construction. Thus given P to prove, if we generate

$$\begin{tikzcd} & P \arrow[dl] \arrow[dr] & \\ Q & & R \end{tikzcd}$$

then both Q and R together can be seen as an explanation of P. Let ε be the explanation function; if ε(Q) = ε(R) = true then the conjunction Q and R is a satisfactory explanation of P. In our computer implementation, the explanation function will receive an explanation as input and return an output (either **true**, showing the explanation is acceptable or **false**, showing it is not).

Suppose either ε(Q) = false or ε(R) = false; what should be done? In such a case, the proof tree can be grown from the goals that are unsatisfactory in the hope that the subgoals generated will be satisfactory. As an example, assume that ε(Q) = true and ε(R) = false. Refining on R happens to generate two subgoals S and T. S is satisfactory; T is not. But T can be proved by a subgoal U that is satisfactory. So we derive a satisfactory explanation of P; the conjunction of Q, S and U.

66 This illustrates one of the most profound aspects of programming; namely that a good solution is not a solution to a specific problem. It is a solution to a class of problems in which the original problem was found.

$$\begin{tikzcd}
& \varepsilon(P) = \text{false} \arrow[dl] \arrow[dr] & & \\
\varepsilon(Q) = \text{true} & & \varepsilon(R) = \text{false} \arrow[dl] \arrow[dr] & \\
& \varepsilon(S) = \text{true} & & \varepsilon(T) = \text{false} \arrow[d] \\
& & & \varepsilon(U) = \text{true}
\end{tikzcd}$$

Figure 16.1 A labelled proof tree showing the explanation of P

The means of generating an explanation is to develop the tree in the usual way, but to halt the growth of the tree at the point where all the leaves of the tree are marked as satisfactory and return those leaves as the explanation. Lets call this the **basic explanation procedure** or **BEP**.

Before passing on to representing BEP in Qi, it is important to see that:

1. BEP may not find a satisfactory explanation.
2. There may be more than one satisfactory explanation to be found.
3. Satisfactory goals may not be part of any satisfactory explanation.

The first two points are obvious. The last situation arises when BEP backchains from P to a series of goals G_1,..G_i, where one or more of G_1,..G_i are satisfactory but the remainder G_j,..,G_n are not, and there is no set of goals explaining G_1,..,G_i which constitutes a satisfactory explanation of P. In this case BEP must undo its commitment to G_1,..,G_i. This means that BEP is naturally encoded as a non-deterministic algorithm.

16.3 Coding BEP in Qi

In this section we will learn how to invoke refinement proofs outside of the Qi proof tool. The two important functions for this are **to-goals** and **from-goals**. The **to-goals** function has the type **([([wff] * wff)] * [note]) → goals))**. The job of this function is to coerce a list of tuples (each tuple representing a sequent) and a list of notes into a series of goals. These goals can then be given to the **refine** function. Two useful functions we will use are **theory-size** - which returns the number of rules in a theory and **notes-in**, which extracts the list of notes from an ongoing proof.

In representing BEP in Qi, it is simple to arrange for satisfactory explanations to be removed from the list of goals and placed on the notepad. BEP then returns the contents of the notepad when all the goals have been solved. To begin, let's use some useful synonyms and a datatype declaration.

```
(synonyms
  explanation [note]
  note wff
  exp-function (wff - -> boolean)
  theory symbol
  problem wff
  rule-counter number)

(datatype globals
_______________________________
   (value *explains?*) : exp-function;)
```

Figure 16.2 The types used in BEP

Our version of BEP is a variation on the backward-chaining algorithm of chapter 7. The inputs to BEP are a problem (wff), an explanation function, and the name of a theory (symbol). Our system is designed to operate independently of the proof tool, so the initial step is to form an object of type **goals**. This is accomplished through the function **to-goals.** The problem is paired with an empty list of assumptions and the result (a sequent) is placed in a list that is in turn paired with an empty list of notes. The whole object is sent to **to-goals,** which coerces it into an object of type **goals**. The input is now in a form suitable for refinement proofs.

```
(define explain
  {problem - -> exp-function - -> theory - -> explanation}
  Problem ExpF Theory -> (let Sequents [(@p [] Problem)]
      (let Goals (to-goals (@p Sequents []))
          (bc ExpF Goals Theory 1))))
```

Figure 16.3 The top level of BEP in Qi

bc performs the task of generating explanations. **bc** is actually a variation on the backward-chaining algorithm of chapter 7. The idea is that **bc** performs backward-chaining proofs, storing the explanation on the notepad and returning the explanation (notepad) when all goals are solved. A rule counter is used to keep track of the rule being used in the theory and backtracking occurs whenever the rule counter exceeds the number (**theory-size**) of rules in the theory. The empty list **[]** is used as the failure object, since an empty explanation is no explanation at all!

```
(define bc
{exp-function - -> goals - -> theory - -> rule-counter - -> explanation}
  ExpF Goals _ _ -> (notes-in Goals)      where (solved? Goals)
  ExpF Goals Theory N -> []  where (> N (theory-size Theory))
  ExpF Goals Theory N <- (fail-if empty?
      (let NewGoals (refine Theory N Goals)
     (if (= NewGoals Goals) []
            (bc ExpF (fix (/. X cull-explanation ExpF X)) NewGoals)
              Theory 1))))
  ExpF Goals Theory N -> (bc ExpF Goals Theory (+ 1 N)))
```

Figure 16.4 The main routine of the explanation generator

Our program does not explain how the explanation ends up in the notepad; the function **cull_explanation** actually does this job. In our version, the task of culling an explanation is assigned to the rule of a theory that stores the goal on the notepad if the goal is part of an acceptable explanation.

```
(theory exptheory

  name (cull-explanation 1)
  if ((head Parameters) P)
  let Notes (if (element? P Notes) Notes [P | Notes])
  ______________________
  P;)
```

Figure 16.5 Extracting an explanation using side-conditions

16.4 micro-MYCIN in Qi

We can now tackle the coding of Sell's medical expert system. Our system will be called "micro-MYCIN", after the pioneering **MYCIN** medical expert system (**Shortliffe** (1976)).🗀 To begin we need two **knowledge bases**[67] (a) a diagnosis knowledge base that relates diseases and symptoms and (b) a

🗀 Qi Programs/Chap16/micromycin.qi

[67] The knowledge base of an expert system is all the information that the expert system actually holds online. The term "knowledge base" arose as a counterpoint to "data base", to indicate that the sorts of data held by an expert system include types of information not usually associated with traditional data bases (such as rules for instance.)

treatment knowledge base that relates treatments and diseases. The diagnosis rule base has a simple logic rule that says a conjunction can be proved if its components can be proved.

```
(theory diagnosis_rule_base

   P; Q;
   [P and Q];

   [disease smallpox];
   [symptom scabs];

   [disease smallpox];
   [symptom high_temperature];

  [disease smallpox];
  [symptom no_pain];

   [disease cholera];
   [symptom red_rash];

  [disease cholera];
  [symptom stomach-ache];

  [disease cholera];
  [symptom high_temperature];)

(theory treatment_rule_base

  P; Q;
  [P and Q];

  [recommend sulfa-100mg];
  [disease cholera];

  [recommend penicillin-50mg];
 [disease smallpox];)
```

Figure 16.6 The micro-MYCIN knowledge base

Our theories will not load without defining the what we count as a wff. A wff is either a predicate symbol followed by a term which is a symbol, or it is the conjunction of two wffs. We add some handy synonyms to make our code clearer.

```
(datatype wff

 P : wff; Q : wff;
 ===========
 [P and Q] : wff;

 Pred : symbol; Term : symbol;
 =====================
 [Pred Term] : wff;)

(synonyms
  explanation [note]
  note wff
  exp-function (wff - -> boolean)
  theory symbol
  problem wff
  rule-counter number
  symptom wff
  disease wff
  diagnosis explanation
  recommendation wff
  parameter exp-function)
```

Figure 16.7 The syntax of micro-MYCIN

Finally, we can tackle micro-MYCIN. Our system gathers symptoms, diagnoses the disease and then recommends the treatment. If our explanation generator comes up with an empty list of explanations in either diagnosis or treatment, then the machine has no idea what to do.

```
(define micro-MYCIN
  {A - -> string}
   _ -> (let Symptoms (gather-symptoms start)
         (let Disease (diagnose Symptoms)
           (if (no-idea? Disease)  (abort "no diagnosis")
              (do (output "Diagnosis is ~A~%" (conjoin Disease))
                 (let Recommendation (recommend-treatment Disease)
                     (if (no-idea? Recommendation)
                        (abort "no recommendation")
                        (output "Recommendation is ~A~%"
                                       (conjoin Recommendation)))))))))
```

```
(define no-idea?
 {[wff] - -> boolean}
  [] -> true
  _ -> false)
```

Figure 16.8 The top level of micro-MYCIN

Gathering symptoms requires querying the user with a series of options held as list of symptoms. The global variables are set to their predefined values.

```
(datatype globals

  ____________________________________________
  (value *temperature-options*) : (list symptom);

   ___________________________________
  (value *skin-options*) : (list symptom);

  _____________________________________
  (value *pain-options*) : (list symptom);)

(set *temperature-options* [[symptom high_temperature]
                               [symptom low_temperature]
                            [symptom normal_temperature]])

(set *skin-options*  [[symptom scabs]
                       [symptom red_rash]
                       [symptom normal_skin]])

(set *pain-options* [[symptom headache]
                      [symptom stomach-ache]
                      [symptom no_pain]])
```

Figure 16.9 The menu options of micro-MYCIN

The **gather-symptoms** routine that gathers the symptoms simply presents the options in a menu and asks the user to choose.

```
(define gather-symptoms
 {A - -> [symptom]}
  _ ->  [(query-symptoms (value *temperature-options*))
         (query-symptoms (value *skin-options*))
         (query-symptoms (value *pain-options*))])
```

```
(define query-symptoms
 {[symptom] - -> symptom}
 Symptoms -> (do (display-symptoms 1 Symptoms)
                 (nth (input+ number) Symptoms)))

(define display-symptoms
 {number - -> [symptom] - -> symbol}
 _ [] -> (output "~%~%Choose Symptom: ")
 N [Symptom ¦ Symptoms]
 -> (do (output "~A. ~A~%" N Symptom)
        (display-symptoms (+ N 1) Symptoms)))
```

Figure 16.10 The micro-MYCIN user interface

Having gathered the symptoms, the next step is to obtain a diagnosis. To do this we call on our **explain** function, with the request that explanations be cast in terms of diseases. A disease is just a wff whose predicate is **disease**. The list of symptoms is formed into a conjunction before being passed to **explain**.

```
(define diagnose
 {[symptom] - -> diagnosis}
  Symptoms -> (explain (conjoin Symptoms)
                disease?
                diagnosis_rule_base))


(define conjoin
 {[wff] - -> wff}
 [P] -> P
 [P ¦ Ps] -> [P and (conjoin Ps)])

(define disease?
  {disease - -> boolean}
  [disease _] -> true
  _ -> false)
```

Figure 16.11 Diagnosis routine

This leaves recommendations to be made. We can think of the question "Recommend a treatment for disease *d*" as equivalent to "Explain how you would treat disease *d*". Hence recommendations follow the same pattern as diagnoses; the difference being that the kind of explanation being looked for is a recommendation and the rule base used is not the diagnosis rule base but the treatment rule base.

```
(define recommend-treatment
  {diagnosis - -> [recommendation]}
  Diagnosis -> (explain (conjoin Diagnosis)
                                  recommendation? treatment_rule_base))

(define recommendation?
   {recommendation - -> boolean}
   [recommend _] -> true
   _ -> false)
```

Figure 16.12 Treatment routine

Here is a script of micro-MYCIN in action.

```
(5+) (micro-MYCIN go)

1. [symptom high_temperature]
2. [symptom low_temperature]
3. [symptom normal_temperature]

Choose Symptom: 1

1. [symptom scabs]
2. [symptom red_rash]
3. [symptom normal_skin]
Choose Symptom: 2

1. [symptom headache]
2. [symptom stomach-ache]
3. [symptom no_pain]
Choose Symptom: 2

Diagnosis is [[disease cholera] and [disease smallpox]]
Recommendation is [[recommend penicillin_50mg]
and [recommend sulfa_100mg]]

"done" : string
```

Figure 16.13 micro-MYCIN in action

This is only a beginning of an effective expert medical system. Exercise 16 supplies some avenues for exploration.

Exercise 16

1. *BEP produces only one explanation of any situation. Modify the program so that it produces the set of all possible explanations that can be derived from the knowledge base and pairs each one with a recommendation.

2. *Ockham's Razor is a scientific principle that derives from the fourteenth century logician, William of Ockham. In its original Latin form it states "Pluralitas non est ponenda sine neccesitate"', which translates as "Entities should not be multiplied unnecessarily". For example, since the motion of planets can be explained through Newtonian mechanics, it is not necessary to postulate angels to keep the planets in their courses. However Ockham's Razor has a rather wider application to scientific theories, because it insists that given a situation and two competing theories to explain it, we should adopt the simpler one.

   Our micro-MYCIN program violates Ockham's Razor because it uses a conjunction of two diseases, cholera and smallpox. as an explanation for the symptoms of high temperature, red rash and stomach ache. However cholera on its own, would explain all these symptoms. Change the program so that it orders all possible explanations from the most economical to the least.

3. *Our system allows us to diagnose diseases based on symptoms. But symptoms are not only used to diagnose diseases, but to rule them out. For example, smallpox is typically accompanied by circular lesions on the skin. If this is absent, we can rule out smallpox. Adapt the program so that it uses negative answers to rule out bad explanations.

4. *Our drug presciption program is unrealistic because it diagnoses drugs in a fixed quantity. In real life, drugs are prescribed according to body weight and age. Amend the prescription program so that it prescribes drugs according to these parameters.

5. *micro-MYCIN attaches no probabilities to the rules in its knowledge base; read Russell and Norvig (1995) chapter 15 and Jackson (1999) and, in the light of your reading, modify micro-MYCIN so that it attaches probabilities to its explanations.

Further Reading

This chapter is only designed to show you how to build interactive reasoning systems in Qi. To learn about expert systems in depth, **Jackson** (1999) is a good beginning. **DENDRAL** and **MYCIN** were two of the earliest expert systems and both are discussed in **Charniak & McDermott** (1984). The use of probability theory, specifically **Bayes Law**, is very well covered in that book which discusses

MYCIN and successor systems such as **Caduceus** and **Prospector**. **Crevier** (1993), is a fascinating history of A.I. including a chapter on expert systems (chapter 6). **EBL** (Explanation Based Learning) analyses proof trees to optimise expert systems – see **Norvig** and **Russell** (2002) for an overview.

Web Sites

CLIPS http://www.ghgcorp.com/clips/CLIPS.html is a tool for building expert systems written in C. **Giarratano** and **Riley** (1998) comes with a CD-ROM containing CLIPS 6.05 executables, source code, and documentation. The first half of the book is theory oriented and the second half covers rule-based programming using CLIPS.

Judith Federhofer maintains a comprehensive web site (http://www.computer.privateweb.at/judith), devoted to expert medical systems complete with many links and a history page.

PCAI, http://www.pcai.com, is a publication devoted to AI applications. PCAI runs a site with many links to commercial and non-commercial expert systems; for details see http://www.pcai.com/web/ai_info/expert_systems.html.

17 Intelligent Agents

Expert systems, automated reasoning and intelligent agents show a certain evolutionary process. A theorem-prover is an automated reasoning program for deducing conclusions from a body of assumptions. An expert system is a an automated reasoning program with a mission – to supply the expertise in decision-making associated with long years of human experience. An intelligent agent has a mission too - to simulate an intelligent being performing a task. Unlike an expert system, intelligent agents are not constructed purely for the purpose of reproducing human expertise. Intelligent agents are also constructed for helping model human behaviour, perform routine tasks, and possibly to exist and interact in the physical world.[68] There is a class inclusion relation linking automated reasoners, expert systems and intelligent agents: expert systems are automated reasoners designed to model the human expert and automated reasoning is part of the study of intelligent agents (figure 17.1).

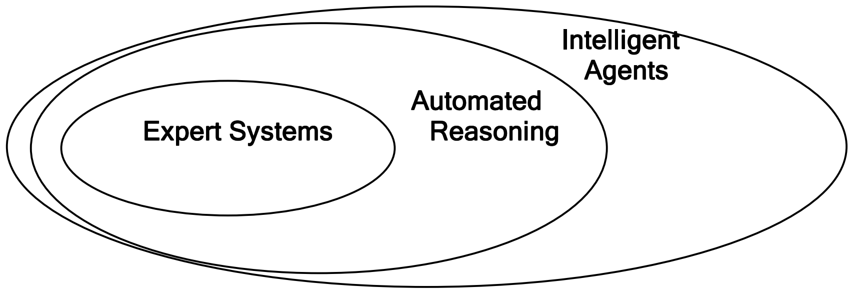


Figure 17.1 The conceptual relations between expert systems, intelligent agents and automated reasoning.

Programs that model human behaviour are not new. **Colby**'s (1975) PARRY was an early attempt to model a paranoid patient using then

[68] This last feature brings in **robotics** and **vision** which is why the study of intelligent agents is broader than automated reasoning. According to Norvig and Russell (1995), the study of intelligent agents is the study of artificial intelligence itself.

current psychological models. **Winograd**'s SHRDLU program (1972) would manipulate imaginary coloured blocks in response to English commands. So if intelligent agents are old, why have intelligent agents become a hot topic at the beginning of the 21st century?

We can isolate two key developments. The first key development is based on the idea that intelligent agents can *interact* with each other. Not only can they reason, but they can act on the conclusions they draw and *change* their environment. In changing the environment, they influence each other. This marks out intelligent agents from older programs like PARRY.

The second key development is not conceptual, but very practical. If the power of using intelligent agents lies in being able to create hundreds or even thousands of interacting agents, then processor performance is at a premium. PARRY and SHRDLU soaked up the entire CPU of the computers on which they ran. Simply, processor performance significantly prior to the beginning of the 21st century did not allow scientists to model large numbers of interacting agents. In this chapter we shall see how to model a system of commuters using intelligent agents in Qi. The work described here is covered in **Faé and Tarver** (2002).

17.1 Modeling Commuters With Intelligent Agents

Our model is based on the following scenario. Every day a number of commuters sets off to work along a certain number of routes. If a commuter is the only user of a route then he travels at maximum speed (say 50 m.p.h.). However every additional commuter on a route slows the travelling speed by 10 m.p.h. down to an absolute minimum speed on a route of 10 m.p.h. Every commuter wants to travel on the fastest route and knows all about the past average speeds on the routes available. Any commuter can travel on any route. What is the aggregate behaviour of this group of commuters over time?

We model🗀 this domain as a pair made up of a **history** H and a set of **agents** A. H is a list of **moments** M_i ordered from present to past. A moment is a set of **events** (atoms of FOL). Each agent has a unique **name** and a **personality**, which consists of a set of **assertions** (wffs of FOL) partitioned into

🗀 Qi Programs/Chap17/comm_model.qi.

a. a set of **beliefs,** which will be wffs of (a subset of) FOL.
b. a set of **desires** represented as dispositions to act; plus ….
c. … a set of **heuristics**; a means of reasoning with those beliefs and desires.

A disposition to act is a conditional whose consequent of the form *todo*(*agent, intention*) and whose antecedents are wffs of FOL. If the antecedents are met, the intention is acted on and the appropriate event added to the present moment M_1. The heuristics will be represented as a tactic. We describe the behaviour of this group by giving a description of the travel changes through <M_1, M_2, M_3, ...>. The subset of FOL we use dispenses with existential quantifiers which are not needed in this model. We define atoms as a type.

```
 (datatype atom

  P : symbol;
  P : atom;

  if (not (= F ~))
  F : symbol; Terms : [term];
  ====================
  [F Terms] : atom; )

(datatype term

 Term : symbol;
  Term : term;

  Term : number;
  Term : term;

 Term : string;
 Term : term;

 Term : boolean;
 Term : term;

  Term : character;
  Term : term;

  F : symbol; Terms : [term];
 ====================
  [F | Terms] : term;)
```

```
(datatype wff

   P : atom;
   P : wff;

   if (not (= F ~))
   [F X] : atom >> Y : A;
   [F X] : wff >> Y : A;

   if (element? C [v => &])
   P : wff; Q : wff;
   ===========
   [P C Q] : wff;

   P : wff;
  =======
   [~ P] : wff;

   if (element? Q [some all])
   X : symbol; R : symbol; P : wff;
 ========================
   [Q X R P] : wff;)
```

The notation is very similar to that of chapter 15, with one significant difference. Universal quantifiers have an extra symbol attached to them which indicates the *range* of the bound variable. The range of the bound variable is the set of objects over which the quantification proceeds. To illustrate; let

$$\forall x\ (\text{solid}(x) \rightarrow \text{free\_fall\_acceleration}(x, 9.8\ m/s^2))$$

mean 'All solid objects accelerate at 9.8 m/s^2 in free fall.' Allowing for approximation, this statement is true – provided that we take the range of the quantifier to be restricted to objects less than 10,000 feet from the surface of the Earth. If we were to consider objects over 100 miles above the surface of the Earth or objects dropped on other planets, the statement would be false. Therefore the truth or falsity of the statement depends on the range of the quantifier.

In our notation, the range of the quantifier is explicitly given as an extra symbol. To make our assertion clear we would write

$$\forall x \in\ E\ \text{solid}(x) \rightarrow \text{free\_fall\_acceleration}(x, 9.8\ m/s^2)))$$

where E denotes the set of all solid objects less than 1,000 feet from the surface of the Earth.

When the range of the quantifier is finite in size, then every universally quantified assertion can be seen as a shorthand for a conjunction. Thus let MP be all the set of men at a party, then

$$\forall x \in \text{ MP wearing\_a\_tie}(x)$$

asserts that every man at the party is wearing a tie. If Tom, Dick and Harry are the only men, then this assertion is true if and only if 'wearing_a_tie(Tom) & wearing_a_tie(Dick) & wearing_a_tie(Harry)' is true. Now let us return to our model. We define some synonyms;

```
(synonyms

   model ([agent] * history)
   history [moment]
   moment [event]
   action term
   event atom
   personality [wff]
   tactic (goals - -> goals))

(datatype globals

______________________
(value *history*) : history;

__________________________
(value *personality*) : personality;

__________________________
(value *routes*) : [symbol];)
```

We define an agent as a structure.

```
(structure agent

  name symbol
  personality [wff]
  heuristics tactic)
```

In our notation, *r* denotes the set of all routes. We assert that a prototypical agent **c**hooses the route which he considers the fastest route provided he has not already chosen to travel a route. In our notation this is written

```
 (all x r(((fastest (agent, x)) & (all y r(~ (travels (agent, y))))) => (todo(a, (travels, a, x)))))
```

We also assert that the fastest route is the one whose average speed is greater than or equal to any other route.

```
(all x r ((all y r (=> ((average_speed, agent, x)
                         (average_speed, agent, y)))) => (fastest (agent, x))))
```

These two assertions constitute the personality of our agents. Next we need to state the heuristics used by this personality; how the agent actually reasons. To do this we need to state a series of rules that define how we can reason with our fragment of FOL.

```
name expall_l
let E (expand X R P)
E >> Q;
[all X R P] >> Q;

name expall_r
let E (expand X R P)
E;
[all X R P];
```

The first two rules allow the replacement of a universally quantified formula by the appropriate conjunct. The function **expand** does this replacement.

```
(define expand
   {symbol - -> symbol - -> wff - -> wff}
   Var Range Wff -> (exp* Var (jumble (den Range)) Wff))
```

expand grabs the denotation of the symbol denoting the range of the function. This range is held as a list of symbols. The point of **jumble** is to return this list in a random order. Returning routes in a random order means that agents will not consider the best routes in the same order.

```
(define jumble
  {[A] - -> [A]}
   [ ] -> [ ]
   L -> (let N (+ (random (length L)) 1)
          (let X (nth N L)  [X | (jumble (remove_nth N L))])))
```

```
(define remove_nth
 {number - -> [A] - -> [A]}
  1 [_ | Y] -> Y
  N [X | Y] -> [X | (remove_nth (- N 1) Y)])

(define den
  {symbol - -> [symbol]}
  r -> (value *routes*))
```

The function **exp*** actually does the job of building a conjunction by substituting each item of the range for the universally quantified variable.

```
(define exp*
 {symbol - -> [symbol] - -> wff - -> wff}
  Var [Val] Wff -> (sub Val Var Wff)
  Var [Val | Vals] Wff  -> [(sub Val Var Wff) & (exp* Var Vals Wff)])

(define sub
 {symbol - -> symbol - -> wff - -> wff}
  Val Var [all Q R P] -> [all Q R (sub Val Var P)]
  Val Var [some Q R P] -> [some Q R (sub Val Var P)]
  Val Var [P & Q] -> [(sub Val Var P) & (sub Val Var Q)]
  Val Var [P v Q] -> [(sub Val Var P) v (sub Val Var Q)]
  Val Var [P => Q] -> [(sub Val Var P) -> (sub Val Var Q)]
  Val Var [~ P] -> [~ (sub Val Var P)]
  Val Var [F X] -> [F (sub* Val Var X)])

(define sub*
 {symbol - -> symbol - -> [term] - -> [term]}
  Val Var [] -> []
  Val Var [Term | Terms] -> [Val | (sub* Val Var Terms)]
                              where (== Var Term)
  Val Var [[Func | Terms] | MoreTerms]
 -> [[Func | (sub* Val Var Terms)] | (sub* Val Var MoreTerms)]
  Val Var [Term | Terms] -> [Term | (sub* Val Var Terms)])
```

The next three rules are simpler. We have two rules for splitting conjunctions and a rule for doing backward chaining with conditionals.

```
name split_r
P; Q;
[P & Q];
```

```
name split_l
P, Q >> R;
[P & Q] >> R;

name backchain
[P => Q] >> P;
[P => Q] >> Q;
```

A rule **planactions** enables an agent to plan actions. Given a desire to do something, if the preconditions of the desire can be shown to hold, then the agent takes the appropriate action.

```
name planactions
P; [does [Agent Action]]; begin;
[P => [todo [Agent Action]]] >> begin;
```

A rule **drop_desire** enables an agent to abandon desires.

```
name drop_desire
begin;
[P => [todo [Agent Action]]] >> begin;
```

The rule **act** says that if an agent is to do something, then he does it!

```
name act
if (execute-action Action)
[does [Agent Action]];
```

execute-action always returns **true.** The side condition changes the history of the computation by adding the **Action** to the current moment of the history. An action is a term, so adding the action to the history means creating an event which is a wff. The function **action_to_event** coerces an action to an event.

```
(define execute-action
  {action - -> boolean}
  Action -> (do (set *history* (ea* Action (value *history*))) true))

(define ea*
  {action - -> history - -> history}
    Action [M | Ms] -> [[(action_to_event Action) | M] | Ms]
    Action [] -> [[(action_to_event Action)]])
```

```
(define action_to_event
  {action - -> event}
   [F | T] -> [F T])
```

Finally **evalatom** allows an atom to be proved providing it is evaluable to **true**.

```
name evalatom
if (evaluable-to-true? P)
P;
```

evaluable-to-true does a lot of work. Given an assertion that the average speed for the agent on route X is greater than or equal to that on route Y, the function measures the relative speeds to see if this is so. Given an assertion that the agent is not traveling on a route, the function checks to see if he is.

```
(define evaluable-to-true?
   {wff - -> boolean}
   [=> [[average_speed Agent X] [average_speed Agent Y]]]
          => (>= (av_speed Agent X (value *history*))
                 (av_speed Agent Y (value *history*)))
   [~ [travels [Agent Route]]]
  -> (not-traveled [travels [Agent Route]] (value *history*))
    _ -> false)

(define not-traveled
  \ Returns true iff the agent has not traveled on the route in the present
moment. \
  {wff - -> history - -> boolean}
   P [[Q | Qs] | _] -> false          where (== P Q)
   P [[_ | Qs] | History] -> (not-traveled P [Qs | History])
   _ _ -> true)

(define av_speed
  \Works out the average speed for an agent on a given route. \
   {term - -> term - -> history - -> number}
   _ _ [_] -> 50
   Agent Route History
   -> (average (map (/. Moment (speed Agent Route Moment))
                    (past History))))
```

```
(define past
  \The past is the tail of the history.\
  {history - -> history}
   [Present | Past] -> Past)

(define speed
   \Speed on a route is 50 mph – Slowdown to a minimum of 10.\
   {term - -> term - -> moment - -> number}
   A R M -> (let Speed (- 50 (slowdown A R M))
                (if (> 10 Speed) 10 Speed)))

(define slowdown
 \Slowdown is 10 mph for every additional user.\
  {term - -> term - -> moment - -> number}
   _ _ [] -> 0
  A R [[travels [A R]] | M] -> (slowdown A R M)
  A R [[travels [_ R]] | M] -> (+ 10 (slowdown A R M))
  A R [_ | M] -> (slowdown A R M))

(define average
  {[number] - -> number}
  Ns -> (/ (total Ns) (length Ns)))

(define total
  {[number] - -> number}
  [] -> 0
  [X | Y] -> (+ X (total Y)))
```

The heuristics of an agent is defined by giving the tactic which it uses to process its beliefs and desires. This tactic is as a loop that processes desires and discards them when they are either fulfilled or shown to be incapable of fulfillment (figure 17.2).

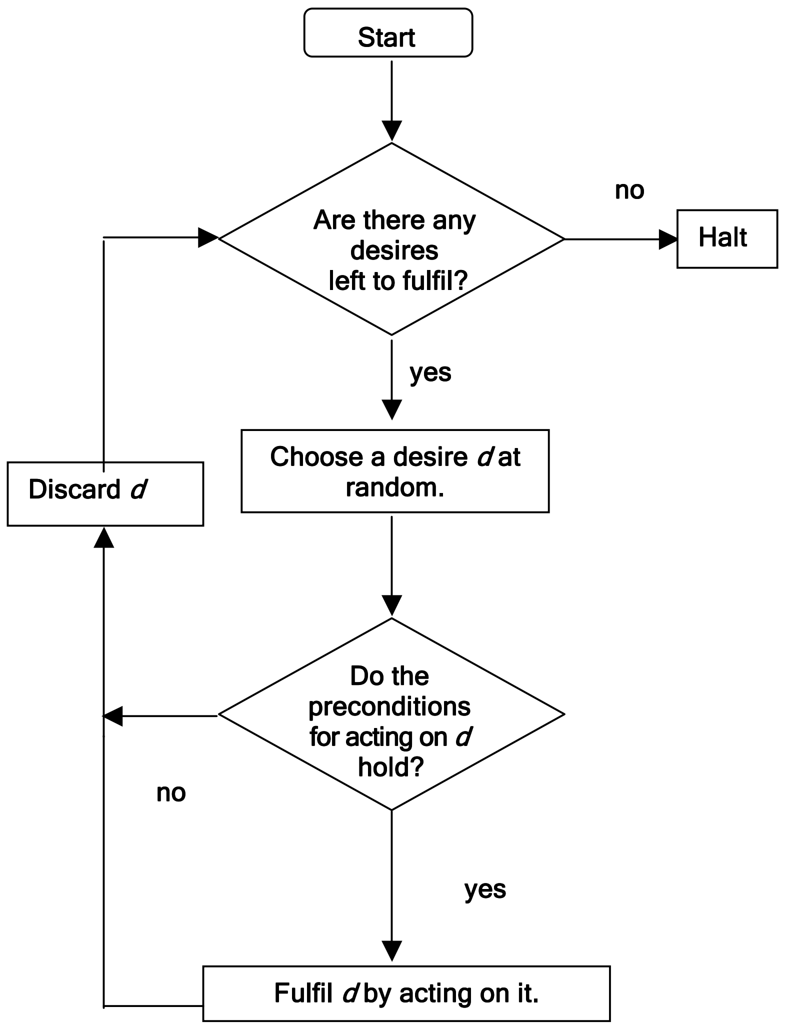

Start
Are there any desires left to fulfil?
no
Halt
yes
Choose a desire *d* at random.
Discard *d*
Do the preconditions for acting on *d* hold?
no
yes
Fulfil *d* by acting on it.

Figure 17.2 The heuristics of an agent.

These heuristics are related to the rules of inference for the subset of FOL we have chosen as follows.

1. All universally quantified assertions held in the personality of the agent are reduced to conjunctions by expansion. All these conjunctions are eliminated by splitting.
2. The first assumption of the form **(P => (todo (Agent, Action)))** is used by **planactions** to generate the preconditions **P**. If there are no such assumptions (no desires) then the agent ceases to act. If the preconditions **P** can be shown to hold then the agent acts by performing the action. If the preconditions cannot be proved then the assumption **(P => (todo (Agent, Action)))** is dropped.
3. Step 2 is then repeated until the agent ceases to act.

Preconditions are proved by a mixture of splitting, expansion, evaluating atoms and backward chaining. The program is given in figure 17.3.

```
(define run_agent
  {goals - -> goals}
   Goals -> (plan_and_do (derive_ground_instances Goals)))

(define derive_ground_instances
\All universally quantified assertions representing the personality of the
agent are reduced to conjunctions by expansion.  All these conjunctions
are eliminated by splitting.\
  {goals - -> goals}
  Goals -> (fix split_l (fix expall_l Goals)))

(define plan_and_do
   {goals - -> goals}
     \ Find a desire and the plan to achieve it. \
   Goals -> (let Plan (planactions Goals)
              (if (no_plan? Plan Goals)
               \If no plans (no desires), then return Goals.\
                 Goals
               \Otherwise, attempt to prove the preconditions of the plan.\
                 (let Actions (prove_preconditions Plan)
                          \If the preconditions are proved, act then plan and do.\
                    (if (preconditions_proved? Actions)
                       (plan_and_do (act Actions))
                \Else drop the desire, and return to planning and doing.\
                      (plan_and_do (drop_desire Goals)))))))
```

```
(define no_plan?
   \If the planactions rule does not change the goals,
   then there are no desires.\
   {goals - -> goals - -> boolean}
   Goals Goals -> true
   _ _ -> false)

(define prove_preconditions
  \Prove the preconditions of the desire by applying the
   composition of several rules to a fixpoint.\
  {goals - -> goals}
  Goals -> (fix (/. Tactic
         (backchain (split_r (evalatom (expall_r Tactic)))))
                 Goals))

(define preconditions_proved?
 \The preconditions are proved if
  the leading conclusion is an action sentence.\
  {goals - -> boolean}
  Goals -> (action_sentence? (fst-conc Goals)))

(define action_sentence?
  {wff - -> boolean}
  [does [Agent Action]] -> true
  _ -> false)
```

Figure 17.3 The heuristics of an agent.

The final part of the program (figure 17.4) sets the model up and runs it, printing out the results.

```
(define model
\Run the model for M routes, N Agents, O Moments.\
  {number - -> number - -> number - -> history}
   M_Routes N_Agents O_Moments
   -> (run_model O_Moments
         (initialise_model M_Routes N_Agents)))
```

```
(define initialise_model
  \Set routes and personality templates, and build agents.\
   {number - -> number - -> model}
    M N -> (do (set *routes* (initialise_routes M))
               (set *personality* [
              [all x r [[[fastest [agent x]]
                              & [all y r [~ [travels [agent y]]]]]  => [todo [agent
                              [travels agent x]]]]]
                      [all x r [[all y r [=> [[average_speed agent x]
                                             [average_speed agent y]]]]
                                       => [fastest [agent x]]]]    ])
                     (@p (make-agents N (value *personality*))
                          (set *history* [])))))

(define initialise_routes
  \Generate M arbitrary routes.\
  {number - -> [symbol]}
  0 -> []
  M -> [(gensym "route_") | (initialise_routes (- M 1))])

(define make-agents
   \Generate N agents with the same personality and different names.\
   {number - -> personality - -> [agent] }
   0 _ -> []
   N Personality
   -> (let Name (gensym "agent_")
        [(make-agent Name
          (init_personality Name Personality) run_agent)
             | (make-agents (- N 1) Personality)]))

(define init_personality
  \Replace the word 'agent' by the name of the agent  to personalise him.\
   {symbol - -> personality - -> personality}
     Name Personality
           -> (map (/. X (sub Name agent X)) Personality))

(define run_model
   \Run the model for O moments.\
   {number - -> model - -> history}
    0 (@p Agents History) -> (pphistory 1 (reverse History))
    O (@p Agents History)
   -> (do (set *History* [[] | (value *History*)])
         (map agent_action Agents)
         (run_model (- O 1) (@p Agents (value *History*)))))
```

```
(define agent_action
  \An agent acts by applying his heuristics to his goals.\
   {agent - -> goals}
   Agent -> ((agent-heuristics Agent) (agent_goals Agent)))

(define agent_goals
 \His goals are derived from his personality  by coercing it to a series of
goals.\
 {agent - -> goals}
 Agent -> (to-goals (@p [(@p (agent-personality Agent) begin)] [])))

\The final piece of code prints the results.\

(define pphistory
 \Print the results down the screen, enumerating the moments.\
 {number - -> history - -> history}
  _ [] -> []
  N [Moment | Moments]
  -> (do (output "~A. " N)
         (scroll Moment)
         (pphistory (+ N 1) Moments)))

(define scroll
 {moment - -> [A]}
  [] -> []
  [Event | Events] -> (do (output "~A~%" Event) (scroll Events)))
```

Figure 17.4 Setting up the model and running it.

17.2 The Story of Agent_95

Lets run this model for 7 commuters using 7 routes for 30 moments. A **fixed commuter** is one who always travels by the same route. As the moments mount, the number of fixed commuters increases. Figure 17.5 asterixes new fixed commuters. This figure shows that by moment #7, all the commuters are fixed in their travel behaviour. Look at agent_95. We can read a story into the behaviour of this agent.

1. (travels (agent_95 route_87)) **(travels (agent_94 route_84))*** **(travels (agent_93 route_86))*** (travels (agent_92 route_82)) (travels (agent_91 route_82)) (travels (agent_90 route_82)) (travels (agent_89 route_88))	2. (travels (agent_95 route_83)) **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** (travels (agent_92 route_85)) **(travels (agent_91 route_83))*** (travels (agent_90 route_85)) (travels (agent_89 route_85))
3. (travels (agent_95 route_87)) **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** (travels (agent_92 route_87)) **(travels (agent_91 route_83))** (travels (agent_90 route_87)) **(travels (agent_89 route_88))***	4. (travels (agent_95 route_83)) **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** (travels (agent_92 route_88)) **(travels (agent_91 route_83))** (travels (agent_90 route_85)) **(travels (agent_89 route_88))**
5.(travels (agent_95 route_87))* **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** (travels (agent_92 route_82)) (travels (agent_91 route_83)) (travels (agent_90 route_82)) **(travels (agent_89 route_88))**	**6. (travels (agent_95 route_87))** **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** (travels (agent_92 route_85)) **(travels (agent_91 route_83))** (travels (agent_90 route_85)) **(travels (agent_89 route_88))**
7. (travels (agent_95 route_87)) **(travels (agent_94 route_84))** **(travels (agent_93 route_86))** **(travels (agent_92 route_82))*** **(travels (agent_91 route_83))** **(travels (agent_90 route_85))*** **(travels (agent_89 route_88))**	8. Ditto 7 for 23 more moments

Figure 17.5 The first 7 moments of history

He starts off (1.) on route_87; and then (2.) tries route_83 – because he knows that nobody used route_83 before. But agent_91 (2.) has chosen it too because he got stuck on route_82 last time (1.) with two others. Consequently, the next time (2.) nobody travels on route_82. In cycle 3, agent_95 has regretted the experiment with route_83 and has tried route_87 again. But this time it's slower than it was when he first used it because two others are using it. In fact it's worse than traveling on route_83 which he shared with only one other. So next (4.) he goes back to route_83. However, agent_91 has become a fixed commuter and always travels on this road, so its back to route_87 in 5. Now agent_95 finds happiness, because commuters are settling down and he's the only user of this route and so he stays with this route throughout.

Agents in this model arrive at the optimal situation of traveling in 1-1 correspondence to routes. Other trials with this model show that the system tends to stabilise – after trial and error – to optimal loading on all routes with fixed commuters. This feature of commuting systems was first identified by Wardrop (1952) and is known as **Wardrop's Principle of Equilibrium.**

17.3 Mathematical Modeling vs. Intelligent Agents

The intelligent agents approach is fascinating, but what do we learn about the intelligent agent approach from this program? Could we not have drawn the same conclusion using **mathematical modeling**? The mathematical approach_attempts to isolate a set of variables and then captures the relationships between those variables in a set of equations. This is the classical approach found in physics and engineering. The model is good if it accurately predicts how the values of variables change in relation to each other. This modeling may be influenced by a 'picture' or analogy which determines what variables are chosen and what the equations are. But the test is whether the model is a good predictor.

In transport, the mathematical approach is **macroscopic**. Mathematical models are concerned with aggregate behaviour of many commuters. The mathematical approach is also **quantitative**. The intelligent agent approach is **microscopic**, it is concerned with what is happening inside the head of an individual commuter. The model is **qualitative**, based on a hypothesis about the beliefs and desires of the agent and how he/she interacts with his/her environment. Each intelligent agent is modeled within the computer and here we use symbolic logic to model the personality of the agent. Powerful computers are used to simulate the interaction of hundreds of these agents and the **quantitative** aspects of the model emerge from an analysis of the way the whole system behaves. The intelligent agents approach is *completely dependent* on the best computing technology.

The intelligent agents approach is tested in the same way as the mathematical approach - by seeing if agents do behave in the predicted way. For instance, our model performs according to Wardrop's Principle which is a good confirmation. Figure 17.6 summarises the differences.

Mathematical Approach	Intelligent Agents Approach
Quantitative	Qualitative
Macroscopic	Microscopic
Tested by seeing if it is a good predictor	Tested by seeing if it is a good predictor
May use powerful computers	Must use powerful computers

Figure 17.6 Mathematical Modeling and Intelligent Agents Contrasted.

The advantage of modeling using intelligent agents is that it allows us to see how simple qualitative desires and attitudes, supported by the basic numerical skills that most people have, results in complex quantitative behaviour when these agents are allowed to interact. The reason computer scientists are excited over this field, is that it can be applied to many fields from commuting to stock markets to ecology, replacing older models based directly on 'number crunching'. In order to unleash the full power of intelligent agents and build models for the complex economic and social systems we have created, we will need computers a million times faster than our best desktop models. But to get the best out of these new wonder machines, we need to abandon languages like C++ and Java and to learn to program in the sorts of high-level representations discussed in this chapter.

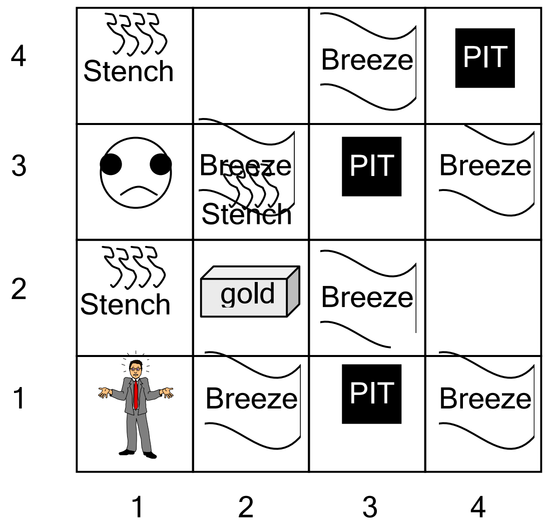
Exercise 17

1. Real transport systems have periodic disruptions due to road works, fog, accidents and so forth. How would you build these into the model? What effects do these incidents have?

2. A "Sunday afternoon driver" is a driver whose psychological profile is different from those of our agents. The Sunday afternoon driver is not interested in getting from A to B as quickly as possible, but chooses routes at random, trying new ones in preference to old. What effect do Sunday afternoon drivers have on ordinary commuters?

3. The **wumpus world** is a grid of squares surrounded by walls, where each square can contain agents and objects. The agent always starts on the lower left corner, a square that we will label [1,1]. The agent's task is to find the gold, return to [1,1] and so climb out of the cave. An example wumpus world is shown below.

4.



The rules of the wumpus world are as follows.

(a) In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
(b) In the squares directly adjacent to the pit the agent will perceive a breeze.
(c) In the square where the gold is the agent will perceive a glitter.
(d) When the agent walks into a wall, it will perceive a bump.
(e) When the wumpus is killed it gives a woeful scream that can be perceived anywhere in the cave.
(f) The permitted agent actions are to go forward, turn right by 90^o and to turn left by 90^o. The action *shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it hits and kills the wumpus or hits the wall. The action *grab* can be used to pick up an object in the same cave as the agent.
(g) The action *climb* is used to climb out of the cave; it is effective only when the agent is at the start square.
(h) Every agent has only one arrow.
(i) The agent dies a miserable death if it enters a square containing a pit or a live wumpus. It is safe to enter a square with a dead wumpus.

The goal is to find the gold and bring it back. 1000 points is given for climbing out of the cave while carrying the gold. 1 point is deducted for each action taken, and 10,000 points are deducted for getting killed. Implement the wumpus world using an intelligent agent. (Taken from Russell and Norvig (1995), p154.)

5. **Apart from transport, can you identify other areas for using intelligent agents? Choose one such area and build an intelligent agent model for it.

Further Reading

Intelligent agents is a fast-growing area of computer science, and this chapter only scratches the surface. **Murch** and **Johnson** (1998) is a study of the commercial aspects and history of intelligent agents. **Tecuci** (1998) contains several case studies of intelligent agent systems. **Ferber** (1999) is a very thorough examination of several different ways of representing intelligent agents from an A.I. perspective. Intelligent agents has its own conferences; the **International Workshops in Multi-Agent-Based Simulation** and the **Second International Symposium Agent Systems, Mobile Agents, and Applications** are printed in the Lecture Notes in Computer Science Series by Springer-Verlag. **Autonomous Agents and Multi-Agent Systems** published by Kluwer is a journal devoted to intelligent agents.

Web Sites

University of Maryland runs a web page http://agents.umbc.edu with many links on intelligent agents.

Appendix A

System Functions and their Types in Qi

- **and**
  boolean → (boolean → boolean)
  Boolean and.

- **append**
  (list A) → ((list A) → (list A))
  Appends two lists into one list.

- **apply**
  (A → B) → (A → B)
  Applies a function to an input.

- **atp-credits**
  string → string
  Returns it's input and as a side-effect, prints the credits when the proof tool is invoked.

- **atp-prompt**
  string → string
  Returns it's input and as a side-effect, prints the credits when the proof tool prompts the user for a proof command.

- **boolean?**
  A → boolean
  Recognisor for booleans.

- **cd**
  string → string
  Changes the home directory. **(cd "My Programs")** will cause **(load "hello_world.txt")** to load **My Programs/hello_world.txt**. **(cd "")** is the default.

- **character?**
  A → boolean
  Recognisor for characters.

- **collect**
  symbol → (number → ((list parameter) → goals → (list goals))))
  Given theory T, number *n*, a list of parameters P and goals G; collects into a list all the goals that can be legally generated from applying the *n*th rule of T to G with parameters P. If no such goals can be legally generated, returns the empty list.

- **complex?**
  A → boolean
  Recognisor for complex numbers.

- **concat**
  symbol → (symbol → symbol)
  Concatenates two symbols.

- **congruent?**
  A → (A → boolean)
  Returns true if objects are identical or else if they are strings or characters which are identical differing at most in case or numbers of equal value (e.g. 1 and 1.0) or tuples or structures composed of congruent elements.

- **cons**
  _
  A special form that takes an object *e* of type A and a list *l* of type (list A) and produces a list of type (list A) by adding *e* to the front of *l*.

- **cons?**
  A → boolean
  Returns true iff the input is a non-empty list.

- **destroy**
  _
  Receives the name of a function and removes it and its type from the environment.

- **debug**
  A → string
  The input is ignored and **debugging** is returned; but all terminal output is echoed to the file **debug.txt** until the **undebug** function is executed.

- **delete-file**
  string → string
  The file named in the string is deleted and the string returned.

- **difference**
  (list A) → ((list A) → (list A))
  Subtracts the elements of the second list from the first.

- **display-mode**
  symbol → symbol
  Receives + or - as inputs. With -, all formulae in the prooftool are displayed and entered in square bracket notation. With +, round brackets are used. The command reader to the proof tool is also changed to read round brackets as list constructors and not function calls.

- **do**
  _
  A special form: receives *n* well-typed expressions and evaluates each one, returning the normal form of the last one.

- **dump**
  _
  Receives a file of Qi source denoted by a string *f* and loads it, dumping the Lisp translation into a file *f*.lsp

- **dump-proof**
  string → string
  Receives a filename as string *f* and dumps the last proof into *f*. Returns **"done"**.

- **element?**
  A → ((list A) → boolean)
  Returns true iff the first input is an element in the second.

- **empty?**
  A → boolean
  Returns true iff the input is [ ].

- **eval**
  _
  Evaluates the input.

- **explode**
  A → (list character)
  Explodes an object to a list of characters.

- **error**
  _
  A special form: takes a string followed by n ($n \geq 0$) expressions. Prints error string.

- **fix**
  (A → A) → (A → A)
  Applies a function to generate a fixpoint.

- **float?**
  A → boolean
  Recognisor for floating point numbers.

- **from–goals**
  goals → ((list ((list wff) * wff) * (list note))
  Applied to goals, returns a tuple composed of (a) a list of individual sequents remaining to be solved of the form((list wff) * wff), each sequent being a tuple of a list of wffs (the assumptions) and a single wff (the conclusion); (b) the list of notes.

- **fst**
  (A * B) → A
  Returns the first element of a tuple.

- **fst-ass**
  goals → (list wff)
  Returns the assumptions of the first sequent of the goals.

- **fst-conc**
  goals → wff
  Returns the conclusion of the first sequent of the goals.

- **fst-goal**
  goals → ((list wff) * wff)
  Returns the first sequent of the goals.

- **gensym**
  string → symbol
  Prints a new symbol consisting of the contents of the string followed by an integer.

- **get-array**
  (array A) → ((list number) → (A → A))
  3-place function that takes an array of elements of type A, an index to that array as a list of natural numbers and an expression E of type A. If an object is stored at the index, then it is returned, otherwise the normal form of E is returned.

- **get-prop**
  _
  3-place function that takes a symbol S, a pointer P (which can be a string, symbol or number), and an expression E of any kind and returns the value pointed by P from S (if one exists) or the normal form of E otherwise.

- **get-rule**
  symbol → (number → (goals → goals))
  Given a name of a theory and a number *n*, gets the *n*th rule of that theory as a function of type goals → goals. If no such theory or rule exist, then the identity function is returned.

- **head**
  (list A) → A
  Returns the first element of a list.

- **if**
  _
  A special form: takes a boolean *b* and two expressions *x* and *y* (both of type A) and evaluates *x* if *b* evaluates to true and evaluates *y* if *b* evaluates to false.

- **if-with-checking**
  string → symbol
  If type checking is enabled, raises the string as an error otherwise returns **ok**.

- **if-without-checking**
  string → symbol
  If type checking is disabled, raises the string as an error otherwise returns **ok**.

- **include**
  (list symbol) → (list symbol)
  Includes the datatype theories or synonyms for use in type checking.

- **include-all-but**
  (list symbol) → (list symbol)
  Includes all loaded datatype theories and synonyms for use in type checking apart from those entered.

- **inferences**
  A → number
  The input is ignored. Returns the number of refinements executed in the proof.

- **input**
  _
  0-place function. Takes a user input *i* and returns the normal form of *i*.

- **input+**
  _
  Special form. Takes inputs of the form **: <expr>**. Where *d*(**<expr>**) is the type denoted by the choice of expression (e.g. 'number' denotes the type number). Takes a user input *i* and returns the normal form of *i* given *i* is of the type *d*(**<expr>**).

- **integer?**
  A → boolean
  Recognisor for integers.

- **length**
  (list A) → integer
  Returns the number of elements in a list.

- **lineread**
  _
  Top level reader of read-evaluate-print loop. Reads elements into a list. **lineread** terminates with carriage return when brackets are balanced. ^ aborts lineread.

- **list**
  A ..... A → (list A)
  A special form. Assembles *n* (*n* ≥ 0) inputs into a list.

- **load**
  string → symbol
  Takes a file name and loads the file, returning **loaded** as a symbol.

- **map**
  (A → B) → ((list A) → (list B))
  The first input is applied to each member of the second input and the results listed.

- **macroexpand**
  _
  By default operates as the identity function. Macroexpand is applied to all inputs that are loaded into Qi or typed to the top level. This function can be overwritten by the user to allow arbitrary code changes.

- **make-string**
  string A_1 … A_n → string
  A special form: takes a string followed by n ($n \geq 0$) well-typed expressions; assembles and returns a string.

- **maxinferences**
  number → number
  Returns the input and as a side-effect, sets a global variable to a number that limits the maximum number of inferences that can be expended on attempting to typecheck a program. The default is 1,000,000.

- **newfuntype**
  _
  Receives the name of a function and a type and adds it to the Qi type environment.

- **notes-in**
  goals → (list note)
  Extracts the list of notes carried with the goals.

- **not**
  boolean → boolean
  Boolean not.

- **nth**
  number → ((list A) → A)
  Gets the nth element of a list.

- **number?**
  A → boolean
  Recognisor for numbers.

- **occurrences**
  A → (B → number)
  Returns the number of times its first argument occurs in its second.

- **or**
  boolean → (boolean → boolean)
  Boolean or.

- **output**
  string $A_1 \ldots A_n$ → string
  A special form: takes a string followed by *n* (*n* ≥ 0) well-typed expressions; prints a message to the screen and returns an object of type string (the string **"done"**).

- **preclude**
  (list symbol) → (list symbol)
  Removes the mentioned datatype theories and synonyms from use in type checking.

- **preclude-all-but**
  (list symbol) → (list symbol)
  Removes all the datatype theories and synonyms from use in type checking apart from the ones given.

- **prf**
  (A → B) → (A → B)
  Takes a tactic and silently traces its execution in a proof. See **dump-proof**.

- **print**
  A → A
  Takes an object and prints it, returning it as a result.

- **profile**
  (A → B) → (A → B)
  Takes a function represented by a function name and inserts profiling code returning the function as an output.

- **profile-results**
  A → symbol
  The input is ignored. Returns a list of profiled functions and their timings since **profile-results** was last used.

• **prooftool**
symbol → symbol
Takes + or - as arguments and invokes the Qi proof assistant. With + as an argument repeats the last problem entered.

• **provable?**
(goals → goals) → ((list ((list wff) * wff) → boolean)
Given a tactic and a list of tuples represented as sequents; returns true if the tactic solves the sequents and false otherwise.

• **put-array**
(array A) → ((list number) → (A → A))
3-place function that takes an array of elements of type A, an index to that array as a list of natural numbers and an expression E of type A. The normal form of E is stored at that index and then returned.

• **put-prop**
_
3-place function that takes a symbol S, a pointer P (a string symbol or number), and an expression E. The pointer P is set to point from S to the normal form of E which is then returned.

• **quit**
_
0-place function that exits Qi.

• **random**
number → number
Given a positive number *n*, generates a random number between 0 and *n*-1.

• **rational?**
A → boolean
Recognisor for rational numbers.

• **read-char**
A → character
The input is discarded and the character typed by the user is returned.

• **read-file**
string → (list unit)
Returns the contents of an ASCII file designated by a string. Returns a list of *units*, where a *unit* is an unspecified object.

- **read-file-as-charlist**
  string → (list character)
  Returns the list of characters from the contents of an ASCII file designated by a string.

- **read-chars-as-stringlist**
  (list character) → ((character → boolean) → (list string))
  Returns a list of strings whose components are taken from the character list. The second input acts as a tokeniser. Thus **(read-chars-as-stringlist [#\H #\i #\Space #\P #\a #\t] (/. X (= X #\Space)))** will produce **["Hi" "Pat"]**.

- **real?**
  A → boolean
  Recognisor for real numbers.

- **refine**
  symbol → (number → ((list parameter) → (goals → goals)))
  Given inputs a symbol *t*, a positive integer *n,* a list of parameters *p*, and goals *g*, applies refinement rule *n* of theory *t* to *g* with parameters *p*.

- **reverse**
  (list A) → (list A)
  Reverses a list.

- **round**
  number → number
  Rounds a number.

- **save**
  $\bar{0}$ place function. Saves a Qi image.

- **snd**
  (A * B) → B
  Returns the second element of a tuple.

- **solved?**
  goals → boolean
  Returns **true** iff there are no goals left to solve.

• **specialise**
symbol → symbol
Receives the name of a function and turns it into a special form. Special forms are not curried during evaluation or compilation.

• **spy**
symbol → boolean
Receives either + or – and respectively enables/disables tracing the operation of *T**.

• **sqrt**
number → number
Returns the square root of a number.

• **step**
symbol → boolean
Receives either + or – and respectively enables/disables stepping in the trace package.

• **string?**
A → boolean
Recognisor for strings.

• **strong-warning**
symbol → boolean
Takes + or -; if + then warnings are treated as error messages.

• **symbol?**
A → boolean
Recognisor for symbols.

• **tail**
(list A) → (list A)
Returns all but the first element of a non-empty list.

• **tc**
symbol → boolean
Receives either + or – and respectively enables/disables static typing.

• **theory-size**
symbol → number
Returns the number of refinement rules in a theory. 0 if no such theory exists.

- **thm-intro**
  symbol → symbol
  Compiles the last theorem proved into a proof rule, whose name is the symbol cited.

- **to–goals**
  ((list ((list wff) * wff) * (list note)) → goals
  Inverse of **from-goals**.

- **time**
  A → A
  Prints the run time for the evaluation of its input and returns the normal form.

- **time-proof**
  (goals → goals) → ((list wff) → (wff → boolean))
  Takes a tactic of type goals → goals and a list of wffs (the assumptions) and a wff (the conclusion) and returns true if the tactic can solve the problem and false if not. The performance timings are printed out.

- **track**
  (A → B) → (A → B)
  Tracks the I/O behaviour of a function.

- **tuple?**
  A → boolean
  Recognisor for tuples.

- **undebug**
  A → string
  The input is ignored, **undebugging** is returned and all terminal output is closed to the file **debug.txt.**

- **union**
  (list A) → ((list A) → (list A))
  Forms the union of two lists.

- **unprf**
  (A → B) → (A → B)
  Undoes **prf**.

- **unprofile**
  (A → B) → (A → B)
  Unprofiles a function.

• **unspecialise**
symbol → symbol
Receives the name of a function and deletes its special form status.

• **untrack**
(A → B) → (A → B)
Untracks a function.

• **value**
_
Applied to a symbol, returns the global value assigned to it.

• **variable?**
A → boolean
Applied to a variable, returns true.

• **version**
(string → string)
Changes the version string displayed on startup.

• **warn**
(string → string)
Prints the string as a warning and returns “done”. See **strong-warning**.

• **write-to-file**
string → (A → string)
Writes the second input into a file named in the first input. If the file does not exist, it is created, else it is overwritten. If the second input is a string then it is written to the file without the enclosing quotes. The first input is returned.

• **@c**
_
Variation of **cons**.

• **@p**
_
Takes two inputs and forms the ordered pair.

• **+**
number → (number → number)
Number addition.

- –
  number → (number → number)
  Number subtraction.

- *
  number → (number → number)
  Number multiplication.

- /
  number → (number → number)
  Number division.

- /.
  _
  Abstraction builder, receives a variable and an expression; does the job of λ in the lambda calculus.

- >
  number → (number → boolean)
  Greater than.

- <
  number → (number → boolean)
  Less than.

- =
  _
  Equal to. A special form that compares two objects of the same type for equality and returns **true** if they are equal and **false** otherwise.

- **==**
  A → (B → boolean)
  Equal to.

- >=
  number → (number → boolean)
  Greater than or equal to.

- <=
  number → (number → boolean)
  Less than or equal to.

Appendix B

Installing and using Qi under Lisp

Currently Qi runs under Clisp under the GNU licence. There are binaries for Qi under Windows and the system includes source code for installing Qi under other Lisps and operating systems. The code size is 250K. The system is provided "as is" – without assuming any liability. To download, see the Lambda Associates site www.lambdassociates.org.

You are free to change the sources as long as you do not remove the copyright notice on the source listings and the copyright notice that appears on starting up Qi. If you do change the source and plan to distribute it, then the **version** function must be used to change the version variable.

Using Lisp in a Qi Function Definition

Qi is written in Lisp, and it is very trivial to get Qi to communicate with the underlying Lisp. All Common Lisp system functions are in uppercase; **REVERSE**, **APPEND** etc. Here is an example which uses the Lisp pretty-printer and pretty-prints the reverse of the input.

```
(define print-reverse-list
  X -> (PPRINT (reverse X)))
```

Talking to Lisp via the Qi Top Level

The Qi reader reads in input and converts it into Lisp, currying and inserting quotes where necessary. Lisp system macros like DEFUN and DEFMACRO can be used in the top level. Here is an example.

```
(1-)  (DEFUN what (x)
          (COND ((EQ x 'yes) 'no)
                     (T 'yes)))
what

(2-) (what yes)
no
```

Hence you *can* type Lisp definitions into the top level and mix Lisp with Qi in one file and load them together. Uppercase Lisp system functions (like REVERSE etc) can also be used at the top level. Lisp system functions will not sustain currying the way Qi ones will, but by defining a Lisp function in Qi, you get currying for free.

However it has to be said that since the Qi reader will tweak symbols like = (see appendix C) in order to preserve conformity with Qi function conventions, typing Lisp definitions to the top level is not the best way to go. Rather better is to use **LOAD** (note the uppercase) to load a Lisp file with the conventional Lisp reader syntax. Apart from case-sensitivity and the semi-colon, which is defined as a standard character in Qi, the conventions are standard Lisp.

If you have a legacy system written in Common Lisp where you used the default case insensitive versions of Lisp, then the default meanings of **if** etc. will not obtain even if you LOAD the source into Qi. In this case you must edit the file. Smart Lisp programmers will write a short routine that uppercases the ifs etc. without changing function names.

Generating and Loading Lisp Code

Qi is written in and is compatible with Lisp code and any Qi program can be turned into a corresponding Lisp program by means of the **dump** function in Qi. Given an Qi program in a text file (call it **foo**), the command **(dump "foo")** will load the Qi code in **foo** into Qi and produce a file **foo.lsp** with the equivalent Lisp program within it.

Communicating to the Lisp Compiler

Qi compiles into Lisp with compiler settings set for maximum speed and minimum regard for memory usage. The Qi type-checker will also insert type declarations into the Lisp code using the Common Lisp compiler directives. Based on a study by Ma and Kessler (1990), these compiler directives will produce a speedup in the object code of between 16% and 75% with an average of 25%. A routine interprets the type information provided by the Qi type-checker and transforms that information into Common Lisp compiler directives. This routine depends on information correlating the types in Qi, with the types that Common Lisp recognises. This information is held in a global A-list **qi::*assoctypes*.** The full range of Common Lisp types is given in **Franz Inc.** (1988) and **Steele** (1990). The **qi::*assoctypes*** global has the following default value.

```
[[list LIST] [symbol SYMBOL] [character CHARACTER]
[boolean SYMBOL] [string STRING] [goals LIST] [number NUMBER]]
```

Therefore, if we defined a datatype **foo**, where a **foo** was a symbol of some kind, we could add **[foo SYMBOL]** to this list. Caution should be exercised in changing **qi::*assoctypes***, since bad associations will cause dynamic type failures.

Enlarging and Shrinking Qi

Lisp contains hundreds of functions; in the interests of simplicity, Qi includes only the most basic. However armed with only knowledge of a little Lisp, you can build versions of Qi that are tailored to include whatever part of Lisp you desire. As a case example, we will consider enlarging Qi by adding logarithms. Common Lisp contains the system function **LOG** that calculates the log of X to the base Y; suppose we want to add this function to Qi. First, we construct a wrapper function **log** that defines the Qi version of **log** in terms the Lisp version.

```
(7-) (define log
     X Y -> (LOG X Y))
log
```

We can calculate logarithms, but we cannot type-check functions using **log** because **log** is not part of Qi's type system. We use the **newfuntype** function to enter log into Qi.

```
(8-) (newfuntype log (number - -> number - -> number))
log

(9-) (tc +)
true

(10+) (log 100.0 10.0)
2.0 : number
```

Finally having built our new version of Qi, we want to give it a name and save it. The following commands do that.

```
(11+) (tc -)
false

(12-) (version "Logarithmic Version")
"Logarithmic Version"
```

(13-) (save)

The final command saves a new executable version of Qi. Your new version will appear with the prompt.

Qi, Copyright …… Mark Tarver
Logarithmic Version

These techniques allow the construction of personalised versions of Qi that include powerful Lisp packages like LispStat, CLOS and CLIM.

Qi can be shrunk as well as enlarged. The function **destroy** followed by the function name will remove a function definition from the environment and remove its type from Qi's type system. It is easy to create a pure functional subset of Qi in this way. Care must be exercised with **destroy** for obvious reasons.

The Qi Readtable

Qi changes the standard Lisp readtable only by making the semi-colon a standard character and by enabling case-sensitivity. In other cases the Qi uses its own reader which changes input according to the scheme in the next appendix.

Macroexpansions

Qi was intended to be easily extended and modified by the expert programmer and **macroexpansions** are a product of this approach. The system function **macroexpand** is applied to all inputs typed to the top level and to every element of every file loaded into Qi. By default, **macroexpand** is defined as the identity function.

However the **macroexpand** function can be redefined to allow the user to define her own shortcuts or notational styles. A simple macroexpansion can be written that enables arithmetic operations to be written in infix notation. Figure B 1. shows how this is done.

```
(0-) (define macroexpand
        [X Arith Y]  -> [Arith (macroexpand X) (macroexpand Y)]
                                  where (element? Arith [+ - * /])
        [X | Y] -> (map macroexpand [X | Y])
        X -> X)
macroexpand

(1-) (tc +)
true

(2+) (define factorial
        {number - -> number}
         0 -> 1
         X -> (X * (factorial (X - 1))))
factorial : (number - -> number)
```

B 1. *Using a macroexpansion to introduce infix notation*

Appendix C

The Syntax of Qi and the Qi Reader and Printer

The syntax of Qi is presented as a context-free grammar in BNF notation annotated where necessary to explain certain context-sensitive restrictions. Terminals which represent parts of the Qi language are bolded, and in particular the bar | in Qi is bolded to **|** to distinguish it from the | used in the BNF to separate alternatives. For all *X*, the expansion <*X*> ::= ε | … indicates that <*X*> may be expanded to an empty series of expressions.

Syntax Rules for Qi Function Definitions

<def> ::= **(define** < lowercase > **{**<types>**}** <rules>**)**
| (define <lowercase> <rules>**)**
<lowercase> ::= any <symbol> not beginning in uppercase
<rules> ::= <rule> | <rule> <rules>
<rule> ::= <patterns> **–>** <item> | <patterns> **<–** <item>
| <patterns> **–>** <item> **where** <item>
| <patterns> **<–** <item> **where** <item>
<patterns> ::= ε | <pattern> <patterns>
<pattern> ::= <base> (except **–>** **|** and **<–**)
| **[**<pattern> <patterns> **|** <pattern>**]**
| **[**<patterns>**]** | **(cons** <pattern> <pattern>**)**
| **(list** <patterns>**)** | **(@p** <pattern> <pattern>**)** | **NIL** | **( )**
<item> ::= <base> | **[**<items> **|** <item>**]** | **[**<items>**]**
| <application> | <abstraction>
<items> ::= <item> | <item> <items>
<base> ::= <symbol> (except **|**) | <string> | <character>
| <boolean> | <number> | **( )** | **[ ]** | **NIL**
<application> ::= **(**<items>**)**
<abstraction> ::= **(/.** <variable> <item>**)**
<variable> ::= any <symbol> beginning in uppercase (except **NIL**)
<types> ::= ε | **(**<types>**)** | **[**<types>**]** | <types> <types> | <symbol>

Syntax Rules for Qi Datatype and Theory Definitions

```
<datatype_definition> ::= (datatype <lowercase> <α-rules>)
<theory> ::= (theory <lowercase> <β-rules>)
<α-rules> ::= <α-rule> | <α-rule> <α-rules>
<β-rules> ::= <β-rule> | <β-rule> <β-rules>
<α-rule> :: = <α–side-conditions> <schemes> <underline> <scheme>;
                | <side-conditions> <simple α-schemes>
                         <double underline> <simple α-scheme>;
<α–side-conditions> ::= commit! <side-conditions> | <side-conditions>
<β-rule> :: = <title> <side-conditions> <schemes> <underline> <scheme>;
<title> ::= ε | name <lowercase> | name (<lowercase> <natnum>)
<natnum> ::= a natural number
<side-conditions> ::= ε | <side-condition>
                         | <side-condition> <side-conditions>
<side-condition> ::= if <item> | let <variable> <item>
<underline> ::= _ | one or more concatenations of the underscore _
<double underline> ::= = |  one or more concatenations of =
<simple α-schemes> ::= <formula> ; | <formula> ; <simple α-schemes>
<formula> := <item> : <item> | <item>
<schemes> ::= ε | <scheme> ; <schemes>
<scheme>  ::=  <assumptions> >> <formula> | <formula>
<assumptions> ::= <formula> | <formula>, <assumptions>
```

The Qi Reader

The Qi reader reads in the user input from either file or keyboard in the same manner. Input is read as a stream of characters (e.g. **hello** is read as **#\h #\e #\l #\l #\o**) which are read in one by one and mapped to their internal representation in Qi by the tokeniser. The reader behaves as the identity function except for the following cases. **<sep>** is the set of separators {**#\Space, #\Newline, #\(, #\), #/], #/[**}

Character Input	Special Reader Action
#\,	Read as **#\Space #\+ #\+ #\Space**
#\}	Read as **#\Space #\} #\Space**
#\{	Read as **#\Space #\{ #\Space**
#\[	Read as **#\(#\l #\i #\s #\t #\i #\t**
#\]	Read as **#\)**
# not followed by **#\#**	Comment character. Skip reading until the matching **#** is found.
#\\\|	Read as **#\Space #\@ #\Space**
#\:	Read as **#\Space #\$ #\$ #\Space**
#\"	Suspend all special reader actions and read in the stream verbatim until the matching **#\"** is found.
<sep> #\= <sep>	Read as **<sep> #\q #\i #_ #\= <sep>**
<sep> #\> <sep>	Read as **<sep> #\q #\i #_ #\> <sep>**
<sep> #\< <sep>	Read as **<sep> #\q #\i #_ #\< <sep>**
<sep> #\=> <sep>	Read as **<sep> #\q #\i #_# \= #\> <sep>**
<sep> #\<= <sep>	Read as **<sep> #\q #\i #_# \< #\= <sep>**

The Qi Printer

The Qi printer reads the result of evaluating the user input and prints it to the screen. We say that a symbol is *unscoped* if it does not occur within a string, or within the scope of a tuple operator **@p**. In all cases not listed below, the printer prints the output verbatim. Strings are printed verbatim.

Symbol Input	Special Printer Action
Unscoped (	Print as [
Unscoped)	Print as]
qi_>	Print as >
qi_<	Print as >
qi_=>	Print as =>
qi_<=	Print as <=
qi_=	Print as =
++	Print as ,
$$	Print as :

Appendix D

The Semantics of $\mathcal{L}$

This appendix describes the semantics of the pure functional language $\mathcal{L}$ that is Turing complete. This appendix also gives a series of recursive equations that describes how Qi can be compiled into $\mathcal{L}$. We assume as $\mathcal{L}$ system functions, **++** (the successor function), **−−** (the predecessor function), **@p**, **if**, **let**, **cases**, **where** and **cons**. $\mathcal{L}$ contains two special symbols; $\perp$ designates the error condition and $\otimes$ the special error condition generated from unsuccessfully matching an extended abstraction to an input. A **primitive object** of $\mathcal{L}$ is a string, number, character, boolean or symbol which is not used as a system function.

The Syntax of $\mathcal{L}$

A primitive object is a formula of $\mathcal{L}$.
( ) is a formula of $\mathcal{L}$.
$\perp$ and $\otimes$ are formulae of $\mathcal{L}$.
If *x*, *y* and *z* are formulae of $\mathcal{L}$, so is (if *x y z*), (cons *x y*), (@p *x y*), (where *x y*), (= *x y*), (*x y*).
If *x* is a variable and *y* and *z* are formulae of $\mathcal{L}$, then (let *x y z*) is a formula of $\mathcal{L}$.
If $x_i,\ldots,x_n$ ($0 \leq i \leq n$) are formulae of $\mathcal{L}$, so is (cases $x_i \ldots x_n$).

A primitive object is a pattern.
A variable is a pattern.
( ) is a pattern.
If p_1 and p_2 are patterns then so is (cons p_1 p_2) and (@p p_1 p_2).

If *p* is a pattern and *x* is a formula of $\mathcal{L}$ then the abstraction (λ *p x*) is a formula of $\mathcal{L}$.

If *v* is a variable and the abstraction (λ *v x*) is a formula of $\mathcal{L}$ then (**Y** (λ *v x*)) is a formula of $\mathcal{L}$.

We assume that all patterns are linear; i.e. there are no repeated variables. Next the rules giving the operational semantics of $\mathcal{L}$.

The Operational Semantics of $\mathcal{L}$

The symbol $\Rightarrow$ means "rewrites to" and $\Downarrow$ indicates the normal form of an expression under rewriting; error(x) indicates x evaluates to $\otimes$ or $\bot$. Rules are applied in order of appearance.

Rule #1. (= x y) $\Rightarrow$ true if $\Downarrow x = \Downarrow y$ and not error($\Downarrow x$)

Rule #2. (= x y) $\Rightarrow$ false if $\Downarrow x \neq \Downarrow y$
and not error($\Downarrow x$) and not error($\Downarrow y$)

Rule #3. (= x y) $\Rightarrow \bot$ if error($\Downarrow x$) or error($\Downarrow y$)

Rule #4. ((λ p x) y) $\Rightarrow$ let σ be *match*(p, $\Downarrow y$); if $\sigma = \otimes$ then $\otimes$ else *sub*(σ, x)

Rule #5. (if x y z) $\Rightarrow$ if $\Downarrow x$ = true then y else if $\Downarrow x$ = false then z else $\bot$

Rule #6. (let x y z) $\Rightarrow$ ((λ x z) y)

Rule #7. (cons x y) $\Rightarrow$ if error($\Downarrow x$) or error($\Downarrow y$) then $\bot$
else (cons $\Downarrow x$ $\Downarrow y$)

Rule #8. (@p x y) $\Rightarrow$ if error($\Downarrow x$) or error($\Downarrow y$) then $\bot$
else (@p $\Downarrow x$ $\Downarrow y$)

Rule #9. (++ x) $\Rightarrow$ if $\Downarrow x$ is a number then 1 + $\Downarrow x$ else $\bot$

Rule #10. (- - x) $\Rightarrow$ if $\Downarrow x$ is a number then $\Downarrow x$ - 1 else $\bot$

Rule #11. (cases x_1 … x_n) $\Rightarrow$ if $\Downarrow x_1 = \otimes$ then (cases … x_n) else $\Downarrow x_1$.

Rule #12. (cases) $\Rightarrow \bot$

Rule #13. (where x y) $\Rightarrow$ (if x y $\otimes$)

Rule #14. ($\mathbf{Y}$ (λ v x)) $\Rightarrow$ $[x]_{v/(\mathbf{Y}\ (\lambda\ v\ x))}$

Rule #15. (x y) $\Rightarrow$ if error($\Downarrow x$) or error($\Downarrow y$) then $\bot$ else ($\Downarrow x$ $\Downarrow y$)

Rule #16. $x \Rightarrow x$

The equations for *match* and *sub* are as follows.

Match #1. *match*(x, x) = {}
Match #2. where x is a variable; *match*(x, y) = {<x, y>}
Match #3. *match*((cons x y), (cons w z))
= let σ_1 be *match*(x, w), let σ_2 be *match*(y, z);
if $\sigma_1 = \otimes$ or $\sigma_2 = \otimes$ then $\otimes$ else $\sigma_1 \cup \sigma_2$
Match #4. *match*((@p x y), (@p w z))
= let σ_1 be *match*(x, w), let σ_2 be *match*(y, z);
if $\sigma_1 = \otimes$ or $\sigma_2 = \otimes$ then $\otimes$ else $\sigma_1 \cup \sigma_2$
Match #5. *match*(x, y) = $\otimes$ in all cases not covered by Match #1-#4

Sub#1 *sub*({}, x) = x
Sub#2 *sub*({<x, y>} $\cup$ S, z) = *sub*(S, $[z]_{y/x}$)

The following equations describe a mapping from pure Qi into $\mathcal{L}$; variables are italicised and object language symbols are bolded. We assume all Qi definitions are in cons form. $\oplus$ is our metalevel consing list constructor.

compile_Qi(**(define** $f \oplus$ *rules***)**) = translate_to_$\mathcal{L}$(f, remove_type(*rules*))
compile_Qi(**(**$x \oplus y$**)**) = curry(**(**$x \oplus y$**)**) if $x \neq$ **define**
compile_Qi(**( )**) *=* **( )**
compile_Qi(x) *=* x where x is a primitive expression

\ *Remove type attached to* Qi *function definition* \
remove_type(**({** $\oplus$ x**)**) = remove_type *1*(x)
remove_type(x) = x if x is not fronted by **{**

remove_type *1*(**(}** $\oplus$ x**)**) = x
remove_type *1*(**(**$x \oplus y$**)**) = remove_type *1*(y)
remove_type *1*(**( )**) = $\perp$

\ *place the definition into a canonical form for processing and left linearise the rules* \
translate_to_$\mathcal{L}$(f, *rules*) = <f, mk_abs(map left_linearise, canonical_rules(*rules*)))>

\ *straightforward mapping function* \
map(f, $\perp$) = $\perp$
map(f, **( )**) = **( )**
map(f, **(**$x \oplus y$**)**) = **(**f(x) $\oplus$ map(f, y)**)**

canonical_rules(*rules*) = canonical_rules *(elim_wildcards(*rules*)), **( )**)

\ *eliminate wildcards* \
elim_wildcards(_) = *x*　　　　where *x* is a fresh variable
elim_wildcards(**(***x* ⊕ *y***)**) = map(elim_wildcards, **(***x* ⊕ *y***)**)
elim_wildcards(*x*) = *x*　　　　in all other cases

\ *bracket each case in the definition* \
canonical_rules*(**( ) ( )**) = **( )**
canonical_rules*(**(**→ *result* **where** *test* ⊕ *rules***)**, *patterns*)
= **((where** curry(*test)* reverse(*patterns*) → curry(*result)***)**
⊕ canonical_rules*(*rules*, **( )**))
canonical_rules*(**(**← *result* **where** *test* ⊕ *rules***)**, *patterns*)
= **((where** curry(*test)* reverse(*patterns*) ← insert_fail_if(curry(*result)*)**)**
canonical_rules*(**(**→ *result* ⊕ *rules***)**, *patterns*)
= **((**reverse(*patterns*) → curry(*result)***)** ⊕ canonical_rules*(*rules*, **( )**))
if *result* ≠ **where**
canonical_rules*(**(**← *result* ⊕ *rules***)**, *patterns*)
= **((**reverse(*patterns*) ← *insert_fail_if*(curry(*result*))) canonical_rules*(*rules*, **( )**)) if *result* ≠ **where**
canonical_rules*(**(***p* ⊕ *rules***)***, patterns*) = canonical_rules*(*rules*, **(***p* ⊕ *patterns***)**)
canonical_rules*(*x*) = ⊥　　　　in all other cases

curry(**(cons** *x y***)**) = **(cons** curry(*x*) curry(*y*)**)**
curry(**(@p** *x y***)**) = **(@p** curry(*x*) curry(*y*)**)**
curry(**(if** *x y z***)**) = **(if** curry(*x*) curry(*y*) curry(*z*)**)**
curry(**(=** *x y***)**) = **(=** curry(*x*) curry(*y*)**)**
curry(**(let** *x y z***)**) = **(let** *x* curry(*y*) (curry(*z*)**)**
curry(**(set** *x y***)**) = **(set** *x* curry(*y*)**)**
curry(**(λ** *x y***)**) = **(λ** *x* curry(*y*)**)**
curry(**(**input+ : *x***)**) = **(**input+ : *x***)**
curry(**(***x* ⊕ *y***)**) = **(**curry(*x*) ⊕ curry(*y*)**)**　　where **(***x* ⊕ *y***)** is not a special form
curry(**(***x* ⊕ *y* ⊕ *z***)**) = curry(**((***x* ⊕ *y***)** ⊕ *z***)**) where **(***x* ⊕ *y* ⊕ *z***)** is not a special form
curry(*x*) = *x*　　　　where *x* is not an application

\ *insert fail ifs* \
insert_fail_if(**(fail-if** *f x***)**) = **(fail-if** *f x***)**
insert_fail_if(*x*) = **(fail-if (λ y (= y #\Escape))** *x***)**

\ *make rule left linear* \
left_linearise(*rule*) = fixpoint(left_linearise*, *rule*)
fixpoint(*f*, *x*) = fixpoint*(*f*, *x*, *f*(*x*))
fixpoint*(*f*, *x*, *y*) = *x*　　if *x* = *y*

```
fixpoint*(f, x, y) = fixpoint*(f, y, f(y))          if x ≠ y
left_linearise*((where test patterns arrow result))
= let v be rpted_v(patterns)
        if v = false then (where test patterns arrow result)
        else let x be any fresh variable
        (where (and (= x v) test) left_linearise_v(v, x, patterns) arrow
result))
left_linearise*((patterns arrow result))
= let v be rpted_v(patterns) if v = false then (patterns arrow result)
   else let x be any fresh variable
        (where (= x v) left_linearise_v(v, x, patterns) arrow result)
rpted_v(patterns) = rpted_v*(flatten(patterns))
flatten(( )) = ( )
flatten((x ⊕ y) ⊕ z) = flatten(append((x ⊕ y), z))
flatten((x ⊕ y)) = (x ⊕ flatten(y))

rpted_v*(( )) = false
rpted_v*((x ⊕ y))  =  true if x is a variable that occurs in y
rpted_v*((x ⊕ y))  = rpted_v*(y)   otherwise

left_linearise_v(v, x, v) = x
left_linearise_v(v, x, (y ⊕ z)) = let ll be left_linearise_v(v, x, y)
                                   if ll = y then (ll ⊕ left_linearise_v(v, x, z))
left_linearise_v(v, x, y) = y                    in all other cases

\ create an abstraction \
mk_abs(rules) = case_body(fparams(rules), rules)
fparams(rules) = mk_fparams(arity(rules))

arity((rule)) = arity_rule(rule)
arity((rule ⊕ rules)) = if arity_rule(rule) = arity(rules) then arity_rule(rule)
else ⊥
arity_rule((where test patterns arrow result)) = length(patterns)
arity_rule((patterns arrow result)) = length(patterns)

length(( )) = 0
length((x ⊕ y)) = 1 + length(y)

mk_fparams(0) = ( )
mk_fparams(n) = (x ⊕ mk_fparams(n - 1))                  where x is a fresh
variable

\ create the cases \
case_body(params, rules) = let rules1 be map(mk_enriched_abs, rules)
```

```
                        let rules2 map((λ x mk_app(params, x)),
                    rules1)
                            mk_abs(reverse(params), (cases ⊕
                             rules2))
mk_enriched_abs((where test (pattern ⊕ patterns) arrow result))
= (λ pattern mk_enriched_abs((where test patterns arrow result)))
mk_enriched_abs(((pattern ⊕ patterns) arrow result))
        = (λ pattern mk_enriched_abs(patterns arrow result))
mk_enriched_abs((where test ( ) → result)) = (where test result)
mk_enriched_abs(( ) → result) = result
mk_enriched_abs((where test ( ) ← (fail-if f result)))
        = (where (and test (not (f result))) result)
mk_enriched_abs(( ) ← (fail-if f result))) = (where (not (f result)) result)

mk_app((v ⊕ vs) abs) = mk_app(vs, (abs v))
mk_app(( ), app) = app
mk_abs(( ), x) = x
mk_abs((v ⊕ vs), x) = mk_abs(vs, (λ v x))
```

Appendix E

The Type Security of the $\mathcal{L}$ Type System

We define a **primitive data structure** (p.d.) as a symbol, boolean, number, string, character, or list or tuple of p.d.s. The expression '|- e : A' is taken to mean 'e : A is provable in the $\mathcal{L}$ type system'.

Our proof of the correctness of the $\mathcal{L}$ type system is covered in 3 theorems. In the first theorem, we demonstrate that if e is p.d. and |- e : A, then e : A. In the second theorem, we demonstrate the **subject reduction property**, namely;

if |- e : A and e $\Rightarrow$ e* then |- e* : A.

The third theorem is a consequence of both these theorems; namely that if |- e : A, then the evaluation of e to a p.d. e* will produce an expression of type A.

Notice that it is not true that if e $\Rightarrow$ e* and |- e* : A then |- e : A. The expression (if (= 1 0) a "a") evaluates to "a", but although |- "a" : string, it is not a theorem that (if (= 1 0) a "a") : string.

Theorem 1: if e is a p.d. and |- e : A then e : A.

Proof: Suppose e is a p.d. and |- e : A; then e is either

(a) a primitive expression (symbol, boolean, number, string or character), or
(b) a list of p.d.s, or
(c) a tuple of p.d.s.

Suppose (a), then the type of e is determined by τ, so that e : string iff string $\in$ τ(e) etc. The only type rule in $\mathcal{L}$ for proving the type of e is the *Primitive Rule* which establishes e : A only if A $\in$ τ(e). Hence if |- e : A then A $\in$ τ(e) and so e : A.

Suppose (b), then e is a list and the proof proceeds by induction. Let the **ultimate length** (υ) be defined as follows.

υ(x) = 1 where x is a primitive object.

For any list l, $\upsilon(l) = \Sigma(\upsilon(i))$ where i is an element in l.

Base Case: $\upsilon(e) = 0$

then e = []; according to the *Primitive Rule*, which is the only applicable rule, |- e : ∀A (list A). By the inhabitation rule for lists (chapter 10);

$$\forall A(e : (\text{list } A) \leftrightarrow (\forall x \ \text{element}(x,e) \supset x : A))^{69}$$

Since ~∃x element(x,e) then e : ∀A: (list A).

Inductive Case: the theorem holds for all lists l where $\upsilon(l) \leq n$.

Let e = [X | Y] be a list expression of ultimate length n + 1. Then if e has a type under Qi, it has the type (list A) for some A and so |- e : (list A). This can only be established by the *Cons (right) Rule* and so |- X : A and |- Y : (list A). By the inductive hypothesis we conclude that X : A and Y : (list A). In which case [X | Y] : (list A) and so e : (list A).

Suppose (c), then e is a tuple (@p *x y*). Let the **tuple size** (σ) be defined as follows.

$$\sigma(x) = 0 \text{ if } x \text{ is not a tuple.}$$
$$\sigma(@p\ x\ y) = 1 + \sigma(x) + \sigma(y).$$

The proof is by induction on the size of σ(e).

Base Case: σ(e) = 1

Suppose |- e : A; then for some B, C, A = (B * C) and |- (@p *x y*) : (B * C). But if |- (@p *x y*) : (B * C) then |- *x* : A and |- *y* : B. But since σ(*x*) = σ(*y*) = 0, *x* and *y* are not tuples and are primitive, *x* : B and *y* : C. By the inhabitation rule for tuples:

$$((@p\ x\ y) : (B * C)) \leftrightarrow ((x : B)\ \&\ (y : C))$$

[69] A potential for confusion exists between the use of → for logical implication and → for the function space type operator. Rather than relying on context alone to disambiguate the intended meaning, the proof uses the older ⊃ for logical implication and → for the type operator.

Since the RHS of this equivalence is true, then (@p *x y*) : (B * C) and so e : A.

Inductive Case: the theorem holds for all tuples t where σ(t) ≤ n.

Suppose σ(e) = n + 1 and |- e : A; then for some B, C, A = (B * C) and |- (@p *x y*) : (B * C). But if |- (@p *x y*) : (B * C) then |- *x* : A and |- *y* : B. Also σ(x) ≤ n and σ(y) ≤ n and so by the inductive hypothesis, *x* : B and *y* : C. By the inhabitation rule for tuples:

((@p *x y*) : (B * C)) ↔ ((*x* : B) & (*y* : C))

Since the RHS of this equivalence is true, then (@p *x y*) : (B * C) and so e : A.

Theorem 2: if |- e : A and e ⇒ e* then |- e* : A.

The proof proceeds by cases. The reduction of e to e* must be one of the rules #1-16 in the semantics for *ℒ*.

By rule #1; then e = (= *x y*) and if |- e : A then by the *Equality Rule* A = boolean. By rule #1, e* = true and by the *Primitive Rule* |- true : boolean.

By rule #2; then e = (= *x y*) and if |- e : A then by the *Equality Rule* A = boolean. By rule #2, e* = false and by the *Primitive Rule* |- false : boolean.

By rule #3; then e = (= *x y*) and suppose |- e : A. By rule #3, e* = ⊥ or e* = ⊗ and by the rules for error conditions, e ∗ : A.

By rule #4; then e = ((λ *p x*) *y*). Assume |- e : A, then by the *Applications Rule* for some B

|- (λ *p x*) : B → A
|- *y* : B

Either *match*(*p*, *y*) = ⊗ or *match*(*p*, *y*) ≠ ⊗. Assume *match*(*p*, *y*) = ⊗, then e* = ⊗ and |- e* : A (since |- ⊗ : (∀ A A)).

Assume *match*(*p*, *y*) ≠ ⊗, then either

(a) *p* is a constant and *p* = *y* and so e* = *x*. We have |- (λ *p x*) : B → A which by the *Patterns Rule* is provable only if |- *p* : B and *p* : B |- *x* : A. But if |- *p* : B then *p* : B |- *x* : A iff |- *x* : A. Hence |- e* : A.

(b) p is a variable; then e* = $[x]_{y/p}$. We have |- (λ p x) : B → A which by the *Patterns Rule* is provable only if

z : C |- z : B where z is any arbitrary name and C is a fresh type variable.
z : B |- $[x]_{z/p}$: A

Note: the sequent z : C |- z : B is trivially soluble by unification of B with the fresh C. In this case the *Patterns Rule* just acts like the *Abstractions Rule* in Simply Typed Lambda calculus.

But if z : B |- $[x]_{z/p}$: A for any arbitrary z, then certainly y : B |- $[x]_{y/p}$: A and given |- y : B then |- $[x]_{y/p}$: A and so |- e* : A.

(c) p is a pattern of the form (cons v w) or (@p v w); then since match(p, y) ≠ ⊗ then *match*(p, y) is a set σ of bindings. Each element <v, b> of σ is an association of a variable v with a value b. We know from the *Patterns Rule* that (λ p x) : B → A is provable only if

1. *N1* : A1,...*Nn* : An |- p^*: B;
2. p^*: B |- x^* : A;

where *N1,...Nn* are fresh names and A1,...,An are fresh type variables, p^* results from p by replacing all the variables *x1,..,xn* in p by *N1*,...,*Nn* and x* results from x by replacing all the free variables from *x1,..,xn* by *N1*,...,*Nn*.

p^*: B |- x^* : A states that assuming an arbitrary substitution instance of the variables in p (i.e. p^*) to be of type B, that same substitution applied to the free variables in x produces an object (i.e. x^*) that can be proved to be of type A. Since *match*(p, y) succeeds y is a substitution instance of p. So we have

σ(p) : B |- σ(x) : A

But σ(p) = y and σ(x) = e*. So we have

y : B |- e* : A

But given |- y : B then |- e* : A

By rule #5; then e = (if x y z); and e* = y or e* = z or e* = ⊥. Assume that |- e : A, then this is provable by the *Conditional Rule* and so |- y : A and |- z : A. Since |- ⊥ : A, then |- e* : A.

By rule #6; then e = (let *x y z*), suppose |- e : A. Then by the *Local Rule* for some B, where x^* is fresh;

|- *y* : B and x^* : B |- $z_{x^*/x}$: A

Here e* = ((λ *x z*) *y*); by the *Applications Rule* e* : A if for some B;

|- *y* : B and
|- (λ *x z*) : (B → A)

By the *Patterns Rule*, |- (λ *x z*) : (B → A) just when, where x^* is fresh and C is a fresh type variable.

x^* : C |- x^* : B and
x^* : B |- $z_{x^*/x}$: A

Since *x* is a variable the first sequent is easily soluble (unify C with B). So |- e* : A if both the following are provable.

|- *y* : B and
x^* : B |- $z_{x^*/x}$: A

By hypothesis these are both provable, hence |- e* : A.

By rule #7; then e = (cons *x y*) and |- e : A and either e* = ⊥ or e* = (cons ⇓*x* ⇓*y*). If e* = ⊥ then |- e* : A. Suppose e* = (cons ⇓*x* ⇓*y*). We write 'e $\Rightarrow_n$ e*' when e can be normalised to e* using *n* rewrite rules. The proof of subject reduction is by induction on *n*. Suppose *n* = 0; then e = e* and the proof is immediate. Suppose that subject reduction holds when *n* rules are used; i.e. we assume if |- e : A and e $\Rightarrow_n$ e_n, then |- e_n : A as an inductive hypothesis. We wish to show |- e : A and e $\Rightarrow_{n+1}$ e_{n+1}, then |- e_{n+1} : A. In this case we need to show only that each individual rule of our semantics preserves subject reduction where no further normalisation is needed above what is performed in the rule itself. Such a proof is essentially nothing more than a reiteration of the cases already cited. In the case of rule #7, if no normalisation of *x* or *y* is required, then again e = e* and the proof is immediate.

What our reasoning establishes is that the presence of the ⇓ symbol is irrelevant to the subject reduction property. Provided *every* rule preserves subject reduction in *every* case where no further normalisation is needed above what is performed in the rule itself, then subject reduction will obtain for the system overall. In future we will invoke this argument to banish the ⇓ by the phrase "by induction on the order of rewriting".

By rule #8; then e = (@p *x y*) and |- e : A and either e* = ⊥ or e* = (@p ⇓*x* ⇓*y*). By induction on the order of rewriting, we drop the ⇓ and consider only e* = ⊥ or e* = (@p *x y*). If e* = ⊥ then |- e* : A. If e* = (@p *x y*) then the proof is immediate.

By rule #9; then e = (++ *x*) and |- e : number. Either e* = ⊥ or e* = (1 + ⇓*x*). If e* = ⊥ then certainly |- e : number. If ⇓*x* is a number then 1 + ⇓*x* is a number and by the *Primitive Rule*, |- 1 + ⇓*x* : number and so |- e* : number.

By rule #10; then e = (- - *x*) and |- e : number. Either e* = ⊥ or e* = (⇓*x* - 1). If e* = ⊥ then certainly |- e : number. If ⇓*x* is a number then ⇓*x* - 1 is a number and by the *Primitive Rule*, |- ⇓*x* - 1 : number and so |- e* : number.

By rule #11; then e = (cases x_1 … x_n) and |- e : A. Either ⇓x_1 = ⊗ or ⇓x_1 ≠ ⊗. By induction on the order of rewriting, we drop the ⇓ and consider only the cases x_1 = ⊗ or not x_1 = ⊗.

If x_1 = ⊗ then e* = (cases … x_n). By the *Cases Rule*, |- e : A just when each x_i in (cases x_1 … x_n) is such that |- x_i : A. So certainly it must be true that |- (cases … x_n) : A and therefore |- e* : A.

If x_1 ≠ ⊗, then e* = x_1 and since by the *Cases Rule* for each x_i in (cases x_1 … x_n), |- x_i : A then |- x_1 : A and so |- e* : A.

By rule #12; then e* = ⊗ and since |- ⊗ : (∀ A A)), subject reduction holds.

By rule #13; then e = (where *x y*) and e* = (if *x y* ⊗). Suppose |- e : A, then by the *Guard Rule*, |- *x* : boolean and |- *y* : A. By the *Conditional Rule* to prove |- e* : A, it suffices to prove |- *x* : boolean and |- *y* : A and |- ⊗ : A. By hypothesis, the first two are provable and the third follows from |- ⊗ : (∀ A A)).

By rule #14; in that case e is a combinator expression, (***Y*** (λ *v x*)) and e* = $[x]_{v/(\boldsymbol{Y}\ (\lambda\ v\ x))}$. Assume e : A, then (***Y*** (λ *v x*)) : A and this is provable only if *v* : A |- *x* : A. Consider the proof tree for *v* : A |- *x* : A. For every subgoal of the form *v* : A, the assumption *v* : A will be used to solve it. Now replace this subgoal by (***Y*** (λ *v x*)) and drop the assumption *v* : A. By the *Combinator Rule*, this is provable if *v* : A |- *x* : A, which by hypothesis *is* provable.

Hence if there is a proof of v : A |- x : A then there is a proof of $[x]_{v/(\mathbf{Y}\ (\lambda\ v\ x))}$: A and so |- e* : A.

By rule #15; in that case e is an application (x y). Suppose |- e : A and e* = ⊥ then |- e* : A. Suppose e* = (⇓x ⇓y). By induction on the order of rewriting, we drop the ⇓ and consider only the case (x y) and the proof is trivial.

By rule #16; then e = e* and the proof is immediate.

Theorem 3: if |- e : A then, if ⇓e is a p.d., then ⇓e : A.

Assume |- e : A and ⇓e is a p.d.. We write 'e $\Rightarrow_m$ e*' when e can be normalised to e* using m rewrite rules. The proof is by induction on m. If e $\Rightarrow_0$ ⇓e, then e is a normal form and by theorem 1, ⇓e : A.. Assume the theorem holds for n rewrites. Assume e $\Rightarrow_{n+1}$ ⇓e; then for some e*, e ⇒ e* $\Rightarrow_n$ ⇓e. By theorem 2 we know that if |- e : A then |- e* : A, and by the inductive hypothesis, we have if |- e* : A then ⇓e : A. So |- e : A implies ⇓e : A.

Appendix F

$\mathcal{T}$ and the $\mathcal{L}$ Type System

The AB Theorem

One desirable property of a type checking procedure is that it is terminating, sound and complete; that is to say, the procedure constitutes a decision procedure for the type system in question. Fairly obviously, $\mathcal{T}$ is sound (since it uses only the type rules for $\mathcal{L}$). Moreover the previous appendix established the type security of the system itself. We will prove $\mathcal{T}$ is terminating. We shall see later $\mathcal{T}$ is incomplete in respect of the type rules for $\mathcal{L}$, but can be made complete to a subset of $\mathcal{L}$.

We first have to prove a theorem based on the nature of the type rules that $\mathcal{T}$ uses. The type rules are of two kinds.

A. Type rules that eliminate goals without producing subgoals.
B. Type rules that generate subgoals but reduce the bracketing of an expression in the original goal.

We will call these 'A-rules' and 'B-rules' respectively. Let us say that a sequent system is an **AB system** if every rule in it is an A-rule or a B-rule. Intuitively, any proof procedure which uses an AB system by applying all the rules in it will eventually generate a fixpoint. In other words, it needs to be proved that there is no infinite chain of rule applications R_1, R_2, R_3, …. which can be successively applied to a series of goals G such that for all n, $R_n(\ldots(R_1(G))) \neq R_{n+1}(\ldots(R_n\ (R_1(G))))$. The theorem that states this is the **AB theorem**.

To prove this formally, we need a function, β, which returns a value based on the bracketing found in the expressions used in the proof procedure. We begin by inductively defining β for typings. A typing is an expression of the form $x : \tau$, where x is an expression of $\mathcal{L}$ and τ is a type expression. Typings are effectively the well-formed formulae of a proof conducted by $\mathcal{T}$. Type expressions are defined inductively.

A symbol is a type expression.
If a is a type expression and b is a type expression and v is a variable, then ($a \rightarrow b$), (a * b), (list a), ($\forall\, v\ a$) are type expressions.

The function β is defined inductively over typings as follows.

$\beta(x : t) = \beta(x) + 1$

For $\mathcal{L}$ expressions, β is defined as follows.

$\beta(x) = 0$ if x is a primitive object
$\beta(x_1 \ldots x_n) = \Sigma(x_i) + 1$ for i =1 to i = n.

If $\beta(x) = n$ we say that the **B-value** of *x* is *n*. Given a sequent $\Delta >> C$, we associate it to the B-value of this sequent by the following equation.

$\beta(\Delta >> C) = \Sigma(\beta(x_i)) + \beta(C)$ for all x_i in Δ.

In other words to calculate the B-value of a sequent we simply total the B-values of its constituent wffs. We can now define a B-rule precisely.

A rule R is a B-rule just if whenever R is successfully applied to a tuple of goals $<G_0, G_1,\ldots, G_n>$, the resulting tuple of goals $<G^*_1,\ldots,G^*_m, G_1, \ldots, G_n>$ is such that $\beta(G^*_i) < \beta(G_0)$ for all i, where $1 \leq i \leq m$.

Theorem: all the rules of the type system for $\mathcal{L}$ are B-type rules with the exception of the Primitive, Sequents, Generalisation and Specialisation Rules.

Proof: the proof is long but straightforward. We will cover one case and leave it to the reader to complete the other cases.

The Conditional Rule states

$\underline{X : \text{boolean}; Y : A;\ Z : A;}$
(if X Y Z) : A;

Let $<G_0, G_1, \ldots, .G_n>$ be a tuple of goals. Assume $\beta(G_0) = m$ Assume the Conditional Rule is successfully applied to $<G_0, G_1, \ldots, .G_n>$. If so, then G_0 is of the form $\Delta >>$ (if X Y Z) : A, and the output of the rule application is $<G_x, G_y, G_z, G_1, \ldots, .G_n>$ where G_x, G_y, and G_z are defined as follows.

$G_x = \Delta >> X : \text{boolean}$
$G_y = \Delta >> Y : A$
$G_z = \Delta >> Z : A$

We have the following equalities

$\beta(G_0) = m = \beta(\Delta) + \beta((\text{if X Y Z}) : A)$
$\beta(G_x) = \beta(\Delta) + \beta(X : \text{boolean})$
$\beta(G_y) = \beta(\Delta) + \beta(Y : A)$
$\beta(G_z) = \beta(\Delta) + \beta(Z : A)$

But by the definition of β, the following inequalities hold.

$\beta(X : \text{boolean}) < \beta((\text{if X Y Z}) : A)$
$\beta(Y : A) < \beta((\text{if X Y Z}) : A)$
$\beta(Z : A) < \beta((\text{if X Y Z}) : A)$

Abbreviating $\beta(\Delta)$ by d and $\beta((\text{if X Y Z}) : A)$ by i, and $\beta(X : \text{boolean})$, $\beta(Y : A)$ and $\beta(Z : A)$ by x, y and z respectively, we have the following equalities.

$\beta(G_0) = m = d + i$
$\beta(G_x) = d + x$ where $x < i$
$\beta(G_y) = d + y$ where $y < i$
$\beta(G_z) = d + z$ where $z < i$

Hence $\beta(G_x) < \beta(G_0)$ and $\beta(G_y) < \beta(G_0)$ and $\beta(G_z) < \beta(G_0)$ and the Conditional Rule is a B-rule.

Theorem: the Primitive and Sequents Rules are A-rules.

Proof: by inspection of the rules.

Corollary: the type system for $\mathcal{L}$, excepting the Specialisation Rule, is an AB system.

The omission of the *Specialisation Rule* is generally not important in $\mathcal{T}$, since unification is used to bind variables. The omission of this rule and the *Generalisation Rule* will be discussed later.

An **AB series** is a series of lists of goals where (a) the first list contains a single goal (b) for any element G_n in the series, the immediate successor G_{n+1} is generated from G_n by the successful application of an A-rule or a B-rule. To prove the termination of $\mathcal{T}$, we have to prove that there is no infinite AB series.

Since the β function maps every sequent to its B-value, we can associate each element in an AB series with a list of numbers. Each number is the B-value of the sequent in the goal. Therefore we extend the concept of a B-value (and hence the domain of β) to embrace goals and AB series.

Let G be a goal, where G = $[s_1,\ldots,s_m]$ and let $\beta(s_1) = n_1,\ldots.,\beta(s_m) = n_m$. The B-value of the goal G is the list of numbers $[n_1,\ldots,n_m]$. Let $G_0, \ldots., G_n$ be any series of goals. The B-value of this series is just the series $\beta(G_0),\ldots,\beta(G_n)$.

The B-value of an AB series is therefore a numeric representation of that series; it consists of a series of lists of numbers. Let us call such a series, an **ABn series**. Since this series is derived by a mapping from an AB series, it has the following property.

Let L_n and L_{n+1} be elements of an ABn series and let *a* be the number at the head of L_n. Then either

(a) L_{n+1} is the tail of L_n (L_{n+1} = tail(L_n)) or
(b) L_{n+1} is identical to the result of appending a list $[b_1,\ldots,b_k]$ to tail(L_n) such that for each b_i, $b_i < a$.

In the case (a), the inverse of L_{n+1} under β is derived from its predecessor by an A-operation. In the case of (b), the inverse of L_{n+1} under β is derived from its predecessor by a B-operation.

We will call the operation on L_n that corresponds to case (a) **the A_n operation**, and an operation that corresponds to case (b) is **a B_n operation**. ABn series are thus built up from a list containing a single number, by successively iterating A_n and B_n operations. Since the elements of an AB series are correlated 1-1 with elements of the corresponding ABn series, we can show that there is no infinite AB series by proving the following theorem.

Theorem: There is no infinite ABn series.

Let us say that an ABn series is **protracted**, if there is no element L in the series that lacks a successor when an An or Bn operation could be successfully applied to L. We shall prove that there is no infinite ABn series by proving the following result.

Theorem: If L_n is a non-empty list of numbers in a protracted ABn series, then tail(L_n) occurs in the series.

Proof: the proof proceeds by use of strong induction over the value of the first number *a* in L_n.
Base Case: *a* = 0. Then the only admissible ABn operation that can be carried out on L_n is the An operation, and so L_{n+1} = tail(L_n) and the theorem is proved.

Inductive Case: the theorem holds for all values for *a* of less than *m*. Let L_n = [*m*,....]. Consider L_{n+1}. Either

(a) L_{n+1} is derived from L_n by the A_n operation. If so, L_{n+1} = tail(L_n) and the theorem is proved.
(b) L_{n+1} is derived from L_n by a B_n operation. In which case L_{n+1} = [b_1,...,b_k | tail(L_n)]. Since each of the b_1,...,b_k is less than *m*, the inductive hypothesis applies to each of [b_1,...,b_k | tail(L_n)], [b_2,...,b_k | tail(L_n)] ..... and so on to [b_k | tail(L_n)]. But if the inductive hypothesis applies to [b_k | tail(L_n)], then tail(L_n) occurs in the series and the theorem is proved.

Theorem: every protracted ABn series is terminated by the empty list.

Proof: every ABn series begins with a list containing a single number and the tail of that list is the empty list. Since no ABn operation can be applied to the empty list, any protracted ABn series terminates with the empty list.

Theorem: there is no infinite ABn series.

Proof: assume X is an infinite ABn series. Every infinite ABn series must be protracted, since if it were not, it would terminate with a list to which an ABn operation could be applied. So X is protracted and since every protracted ABn series terminates with [ ], X is a terminating infinite series. By *reductio*, X does not exist.

Hence we can finally assert.

The AB Theorem: there is no infinite AB series.

Proof: since the elements of every AB series are correlated 1-1 with every ABn series, since there is no infinite ABn series, there is no infinite AB series.

Theorem: $\mathcal{T}$ is terminating.

Proof: If $\mathcal{T}$ failed to terminate, then there would be an infinitely long AB series.

Unification and the AB Theorem

The AB theorem seems to contradict with an example of non-termination that was raised at the end of chapter 13. We saw that an attempt to define the type operator **or** lead to an infinite regress – precisely the case that the AB theorem was designed to block. The rule that lead to this regress was

$$\frac{X : A;}{X : (A \text{ or } B);}$$

Inspection shows this rule to behave like a B-rule. How, in the light of our reasoning so far, can this rule lead to an infinite regress?

The culprit is of course, unification. The AB theorem assumes that all AB rules are interpreted conventionally through pattern-matching. However, even an innocuous B rule can become expansive if filtered through unification. The $\mathcal{T}$ algorithm secures termination via the AB theorem because it dispenses with the non-B *Specialisation Rule.* But unification brings its own dangers, so it remains to be shown that this sort of non-terminating case does not arise.

Fortunately this is not too hard. $\mathcal{T}$ applies *unification* with respect to *types* and *pattern-matching* with respect to $\mathcal{L}$ expressions. To verify that the AB theorem still applies, we need only show that, after any type rule is applied, the B-value decreases through a reduction in the B-value of the $\mathcal{L}$ expression and not because of a reduction in the B-value of the type. Inspection shows this to be true of every rule in the system bar one – the *Generalisation Rule*. Allowing this rule requires us to make some caveats.

Using backward chaining, the *Generalisation Rule* eliminates the universal quantifiers at the front of type expressions. The only place where this rule is invoked is in the proof of the type-security of polymorphic functions. Moreover since Qi is an **explicitly typed language**[70] where all types are shallow, if we allow this rule to be interpreted *modulo* pattern-matching, without unification, just at the beginning of the proof, then with this caveat,

[70] That is, the type the function is supposed to have must be explicitly given in the function itself. Other functional languages (like SML for instance) are **implicitly typed**. This means that the language does not require the user to state the type, but instead tries to infer it.

the AB theorem still applies. Alternatively, we may follow $\mathcal{T}^*$ and simply eliminate such universally bound variables by fresh terms – eliminating any need for the *Generalisation Rule* at all.

$\mathcal{T}$ and Completeness

$\mathcal{T}$ is not complete with respect to the type system of $\mathcal{L}$. Functions that involve **generic type variables**[71] will not be type-checked by $\mathcal{T}$. The following example is adapted from Field and Harrison (1988).

```
(define f
  {A --> (number * boolean)}
   X -> (let F (/. Y Y) (@p (F 3) (F true))))
```

The function does have the type A $\rightarrow$ (number * boolean) in our type system. However $\mathcal{T}$ will not recognise it as well typed, generating the problems.

```
X : a >> (/. Y Y) : B
X : a, (/. Y Y) : B >> (@p (F 3) (F true)) : (number * boolean)
```

Application of the *Generalisation Rule modulo* unification will solve the first problem, but then B has to be instantiated in two different ways, so this function will not be type-checked in Qi.

71 See Field and Harrison (1988) (p 148-149) for a discussion of generic types.

Appendix G

A Case Study: an ℒ Interpreter in Qi

This appendix presents a complete interpreter for ℒ written in Qi and based on the semantics for ℒ given in appendix D. This program🗀 is designed both illustrate the semantics of ℒ, and also to present a guide to writing type-secure programs in Qi by reference to a small program with a complex collection of datatypes. We begin by defining the datatypes of ℒ. I've slightly simplified the syntax of ℒ in this program.

```
(datatype number
 ____________________________________
 (number? X) : verified >> X : number;)

(datatype primitive_object

   X : variable;
   ___________________
   X : primitive_object;

   X : symbol;
   ___________________
   X : primitive_object;

   X : string;
   ___________________
   X : primitive_object;

   X : boolean;
   ___________________
   X : primitive_object;

   X : number;
   ___________________
   X : primitive_object;
```

🗀 Load Qi Programs/G/interp.syntax and Qi Programs/G/interp.semantics.

```
      X : character;
       ____________
       X : primitive_object;

       __________________
       [ ] : primitive_object;)

 (datatype pattern

    X : primitive_object;
    ____________
    X : pattern;

    P1 : pattern; P2 : pattern;
    ===================
    [cons P1 P2] : pattern;

    P1 : pattern; P2 : pattern;
    ===================
    [@p P1 P2] : pattern;)

  (datatype variable

   if (variable? X)
   _________________
   X : variable;)

   (datatype l_formula

    X : pattern;
    ___________
    X : l_formula;

    X : l_formula; Y : l_formula; Z : l_formula;
    ================================
    [if X Y Z] : l_formula;

    X : variable; Y : l_formula; Z : l_formula;
    ===============================
    [let X Y Z] : l_formula;

     X : l_formula; Y : l_formula;
    ====================
    [cons X Y] : l_formula;
```

```
X : l_formula; Y : l_formula;
=====================
[@p X Y] : l_formula;

X : l_formula; Y : l_formula;
=====================
[where X Y] : l_formula;

X : l_formula; Y : l_formula;
=====================
[= X Y] : l_formula;

X : l_formula; Y : l_formula;
=====================
[X Y] : l_formula;

Xn : (list l_formula);
================
[cases | Xn] : l_formula;

P : pattern; X : l_formula;
====================
[/. P X] : l_formula;)
```

Part of the task of the $\mathcal{L}$ interpreter is to perform addition and subtraction. Initially I wrote these lines of code as part of the interpreter:

```
[++ X] -> (+ 1 (normal_form X))
[-- X] -> (- (normal_form X) 1)
```

But this caused a type error message. The error is that + and - operate on numbers, and the rules for $\mathcal{L}$ formulae do not allow the inference of **X : number** from **[- - X] : l_formula**.

There are two solutions to the problem. The first is to represent numbers using a successor notation (so **3** would be **(succ (succ (succ 0))))**; this is the sort of solution which one would use in SML. The drawback is that it is obviously a clumsy representation. The chosen solution was to create special functions that add or subtract 1 from the input, if the input is a number and return the error output if it is not. This is neat. Here they are:

```
(define successor
  {A - -> l_formula}
  X -> (+ 1 X)     where (number? X)
  _ -> "error!")

(define predecessor
  {A - -> l_formula}
  X -> (- X 1)     where (number? X)
  _ -> "error!")
```

To typecheck these functions we add an extra rule

```
(datatype number

 ________________________________
  (number? X) : verified >> X : number;)
```

The code for the $\mathcal{L}$ interpreter now follows.

```
(if-without-checking "switch on the typechecker first!~%")

(define l_interpreter
  {A - -> B}
  _ -> (read_eval_print_loop
       (output "~%L interpreter ~%~%~%~%l-interp - -> ")
       (output "~A~%" (normal_form (input+ : l_formula)))))

(define read_eval_print_loop
  {string - -> string - -> A}
  _ _ ->  (read_eval_print_loop
            (output "l-interp - -> ")
             (output "~A~%"
               (normal_form (input+ : l_formula)))))

(define ==>
   {l_formula - -> l_formula}
   [= X Y] -> (let X* (normal_form X)
             (let Y* (normal_form Y)
               (if (or (eval_error? X*) (eval_error? Y*))
                     "error!"  (if (= X* Y*) true false))))
   [[/. P X] Y] -> (let Match (match P (normal_form Y))
                     (if (no_match? Match)
                        "no match"  (sub Match X)))
   [if X Y Z] -> (let X* (normal_form X)
                    (if (= X* true)  Y
                       (if (= X* false)  Z  "error!")))
  [let X Y Z] -> [[/. X Z] Y]
  [@p X Y] -> (let X* (normal_form X)
              (let Y* (normal_form Y)
                (if (or (eval_error? X*) (eval_error? Y*))
                         "error!"  [@p X* Y*])))
  [cons X Y] -> (let X* (normal_form X)
               (let Y* (normal_form Y)
                (if (or (eval_error? X*) (eval_error? Y*))
                           "error!"
                           [cons X* Y*])))
  [++ X] -> (successor (normal_form X))
  [- - X] -> (predecessor (normal_form X))
  [cases X1 | Xn] -> (let Case1 (normal_form X1)
                               (if (= Case1 "no match")  [cases | Xn] Case1))
  [cases] -> "error!"
  [where X Y] -> [if X Y "no match"]
  [y-combinator [/. X Y]] -> (replace X [y-combinator [/. X Y]] Y)
  [X Y] -> (let X* (normal_form X)  (let Y* (normal_form Y)
               (if (or (eval_error? X*) (eval_error? Y*))
                   "error!"
                   [X* Y*])))
   X -> X)

(define eval_error?
 {l_formula - -> boolean}
  "error!" -> true
  "no match" -> true
  _ -> false)

(define normal_form
 {l_formula - -> l_formula}
  X -> (fix ==> X))

(define successor
 {A - -> l_formula}
 X -> (+ X 1) where (number? X)
 _ -> "error!")

(define predecessor
 {A - -> l_formula}
 X -> (- X 1) where (number? X)
 _ -> "error!")
```

```
(define sub
 {[(pattern * l_formula)] - -> l_formula - -> l_formula}
  [] X -> X
  [(@p Var Val) | Assoc] X -> (sub Assoc (replace Var Val X)))

(define match
  {pattern - -> l_formula - -> [(pattern * l_formula)]}
   P X -> []                       where (== P X)
   P X -> [(@p P X)]      where (variable? P)
   [cons P1 P2] [cons X Y]
          -> (let Match1 (match P1 X)
           (if (no_match? Match1)
               Match1
               (let Match2 (match P2 Y)
                 (if (no_match? Match2)
                       Match2
                       (append Match1 Match2)))))
   [@p P1 P2] [@p X Y] -> (let Match1 (match P1 X)
                         (if (no_match? Match1)
                           Match1
                           (let Match2 (match P2 Y)
                            (if (no_match? Match2)
                                Match2
                                   (append Match1 Match2)))))
   _ _ -> [(@p no matching)])

(define no_match?
 {[(pattern * l_formula)] - -> boolean}
  [(@p no matching)] -> true
  _ -> false)

(define replace
 {pattern - -> l_formula - -> l_formula - -> l_formula}
  V W [let V X Y] -> [let V X Y]
  X Y X -> Y
  V W [= X Y] -> [= (replace V W X) (replace V W Y)]
  V W [/. P X] -> [/. P (replace V W X)]  where (free? V P)
  V W [if X Y Z]
  -> [if (replace V W X) (replace V W Y) (replace V W Z)]
  V W [let X Y Z] -> [let X (replace V W Y) (replace V W Z)]
  V W [@p X Y] -> [@p (replace V W X) (replace V W Y)]
  V W [cons X Y] -> [cons (replace V W X) (replace V W Y)]
  V W [cases | Xn] -> [cases | (map (/. Xi (replace V W Xi)) Xn)]
  V W [where X Y] -> [where (replace V W X) (replace V W Y)]
  V W [X Y] -> [(replace V W X) (replace V W Y)]
   _ _ X -> X)

(define free?
  {pattern - -> pattern - -> boolean}
   P P -> false
   P [cons P1 P2] -> (and (free? P P1) (free? P P2))
   P [@p P1 P2] -> (and (free? P P1) (free? P P2))
   _ _ -> true)
```

After some debugging of earlier versions, I ran this program. Here is a script; Qi 6.1 is running under CLisp on a 2.6GHz processor with 0.5 Gb of memory.

```
Qi, Version 6.1
Copyright 2000-2005 Mark Tarver

(0+) (load "Qi Programs/G/interpreter.qi")
ok : symbol
number : unit
primitive_object : unit
pattern : unit
variable : unit
l_formula : unit
l_interpreter : (A --> B)
read_eval_print_loop : (string --> (string --> A))
==> : (l_formula --> l_formula)
eval_error? : (l_formula --> boolean)
normal_form : (l_formula --> l_formula)
successor : (A --> l_formula)
predecessor : (A --> l_formula)
sub : ((list (pattern * l_formula)) --> (l_formula --> l_formula))
match : (pattern --> (l_formula --> (list (pattern * l_formula))))
no_match? : ((list (pattern * l_formula)) --> boolean)
replace : (pattern --> (l_formula --> (l_formula --> l_formula)))
free? : (pattern --> (pattern --> boolean))
typechecked in 719827 inferences.

Real time: 3.3448095 sec.
Run time: 3.3347952 sec.
Space: 109385144 Bytes
GC: 173, GC time: 0.7410656 sec.
loaded : symbol
```

```
(2+) (l_interpreter start)
L interpreter

l-interp - -> [[/. 3 5] 3]
5
```

Here is the combinator definition of the addition function from the chapter on lambda calculus.

```
l-interp - -> [[[y-combinator
            [/. ADD [/. X [/. Y [if [= X 0] Y
                           [[ADD [++ X]] [- - Y]]]]]]]  3] 4]
7
```

Here is a combinator definition of the append function used to append two lists.

```
l-interp  - ->  [[[y-combinator [/. APPEND [/. X [/. Y
                    [if [= X []] Y
                    [cons [[/. [cons A B] A] X]
                     [[APPEND [[/. [cons A B] B] X]] Y]]]]]]]
                                    [cons 1 []]] [cons 2 []]]
[cons 1 [cons 2 []]]
```

Here is a combinator definition of our mutually recursive even and odd functions from chapter 13, used to test whether 5 is even.

```
l-interp - -> [[[/. [@p X Y] X]
                    [y-combinator  [/. T
                      [@p [/. A [cases [[/. 1 false] A]
               [[/. X [[[/. [@p X Y] Y] T] [- - X]]] A]]]
              [/. A [cases [[/. 1 true] A]
                        [[/. X [[[/. [@p X Y] X] T] [- - X]]]
                              A]]]]]]]  5]
false
```

Appendix H

The Qi Debugging Environment

Qi is a fairly complex system to implement, and so during the course of its development, a number of debugging tools were built which found their way into the final release. This appendix details these tools.

Tracking the Execution of a Qi Function

This tool does the job of a LISP trace package, but the print routines are driven by Qi. **(track f)** will cause a Qi function **f** to be tracked. When a function **f** is tracked, each time **f** is called, the inputs to **f** are printed and so is the normal form of the output **f** returns. **untrack** untracks a function. Figure H1. shows the factorial function being tracked.

```
(62-) (track factorial)
factorial

(63-) (factorial 3)
  <1> Input to factorial
   3 ==>
     <2> Input to factorial
     2  ==>
       <3> Input to factorial
          1 ==>
          <4> Input to factorial
             0 ==>
         <4> Output of factorial
             ==> 1
       <3> Output of factorial
          ==> 1
    <2> Output of factorial
       ==> 2
 <1> Output of factorial
==> 6

(64-) (untrack factorial)
```

Figure H1. Using the Qi *Trace Program*

(step +) typed to the top level will cause the tracking package to pause after each **Inputs to …** and wait for a keystroke from the user. Typing **^** (abort) at this point will abort the computation.

Tracking Type Checking

The type checker can tracked in Qi. The type debugger is enabled by the command **(spy +)** and the debugger shows the result of each type rule successfully applied by the type checker. Figure H2. shows part of a sample script.

```
(1+) (spy +)
true

(define factorial
      {number --> number}
       0 -> 1
       X -> (* X (factorial (- X 1))))
____________________________________________________ 9 inferences
?- 0 : number


>
____________________________________________________ 33 inferences
?- 1 : number

0 : number

>
____________________________________________________ 59 inferences
?- ((* X) (factorial ((- X) 1))) : number

factorial : (number --> number)
X : number

>
____________________________________________________ 81 inferences
?- (* X) : (V499884 --> number)

factorial : (number --> number)
X : number

>
```

```
_______________________________________________ 103 inferences
?- * : (V499887 --> (V499884 --> number))

factorial : (number --> number)
X : number

>
_______________________________________________ 123 inferences
?- X : number

factorial : (number --> number)
X : number

>
_______________________________________________ 136 inferences
?- (factorial ((- X) 1)) : number

factorial : (number --> number)
X : number

>
_______________________________________________ 158 inferences
?- factorial : (V499891 --> number)

factorial : (number --> number)
X : number

>
_______________________________________________ 169 inferences
?- ((- X) 1) : number

factorial : (number --> number)
X : number

>
_______________________________________________ 191 inferences
?- (- X) : (V499894 --> number)

factorial : (number --> number)
X : number

>
```

```
_____________________________________________ 213 inferences
?- - : (V499897 --> (V499894 --> number))

factorial : (number --> number)
X : number

>
_____________________________________________ 233 inferences
?- X : number

factorial : (number --> number)
X : number

>
_____________________________________________ 246 inferences
?- 1 : number

factorial : (number --> number)
X : number

>
_____________________________________________ 274 inferences
?- factorial : Type


>
factorial : (number --> number)
```

Figure H 2. Using spy to view the operation of the type checker

As with the trace package, typing **^** (abort) to the debugger prompt > will abort the typechecking. **(spy -)** on the top level will turn off the type debugger.

Echoing Output

The command **(debug <any argument>)** echoes terminal output to the file **debug.txt**. **(undebug <any argument>)** toggles this off.

Profiling Code

The command **(profile** f**)** profiles the function f. The command **(unprofile** f**)** unprofiles the function f by removing the profiling code. The command **(profile-**

results <any argument>) produces a list of timings for all profiled functions, initializing all counters to zero. Figure H3. is a script of a session that profiles the time taken by the **raa** rule used in the propositional ATP of chapter 14.

```
(11+) (profile raa)
raa : (goals - -> goals)

(12+) (time-proof pc_decision_procedure
   [[s <=> [~ [[[p <=> [~ q]] & [[~ p] <=> q]]
         & [[p <=> q] & [[~ p] <=> [~ q]]]]]]]
   [s & s])

Assuming

[s <=> [~ [[[p <=> [~ q]] & [[~ p] <=> q]] & [[p <=> q] & [[~ p] <=> [~ q]]]]]]

Attempting to prove [s & s].

0.0801152 secs, 3271 refinements, 40829 RPS
true : boolean

(13+) (profile-results all)
raa, 0.0100144 secs

profiled : symbol

(14+)  (unprofile raa)
raa : (goals - -> goals)
```

Figure H3. Using the Profiler

Appendix I

Qi/Tk and the IFPE Project

So many fundamental programming techniques have derived from functional programming, that it is both a shame and a wonder that most students and programmers are deprived of any real knowledge of the area. In chapter one, we saw some of the historical reasons why this happened.

The current challenge for functional programming is that modern programming has expanded past the normal perimeters of the read-evaluate-print loop. Graphical interfaces, scripting languages, multi-user databases, event driven programming, sound files, the Internet and virtual conferencing have all sprung up in the last twenty years. Many functional programming environments have nothing to offer in many of these areas.

Tk (short for toolkit) provides a command-driven means for creating widgets (graphical items such as buttons, menus, labels, geometrical shapes and pop-up windows). Qi/Tk augments Tcl/Tk by incorporating a type-secure communication from Qi to Tk, so that GUIs (graphical user interfaces) can be built without type errors. The current platform of Qi/Tk is the Windows operating system, which in terms of distribution is by far the most commonly used OS.

Qi/Tk is currently available from the Lambda Associates website. It is intended that this system will eventually be able to incorporate as much of TCL/Tk as possible. The Lambda Associates website includes online code samples of Qi/Tk and graphics displays. The standard for Qi/Tk is under revision and the manual can be downloaded from the website.

The goal of the IFPE project is to provide an integrated functional programming environment (IFPE) based on Qi/Tk and Qi development tools so that application programs can be written without the use of Java or C++. The final aim is to build a free functional development environment that is second to none. So far I have contributed Qi, Qi/Tk, Qi-YACC and Qi-Prolog and sunry documentation. There is a list on www.lambdassociates.org of ongoing projects for volunteers for IFPE. All Qi systems and IFPE are open source projects running under the GNU licence.

Bibliography

Abelson H. and Sussman G.J.
Structure and Interpretation of Computer Programs, 2nd edition, MIT Press, 1996.

Abramsky S., Gabbay D.M. and Maibaum T.S.
Handbook of Logic in Computer Science, vols. 1-6, Clarendon Press, Oxford, 1993-5.

Adams A. A.
"INDUCT: A Logical Framework for Induction over Natural Numbers and Lists built in SEQUEL", M.Sc. thesis, Leeds Computer Studies 1994.

Aho A.V. and Ullman J.D.
The Theory of Parsing, Translation and Compiling. Vol. 1 Englewood Cliffs, NJ: Prentice Hall, 1972.

Aiken A. and Murphy B.
"Static type inference in a dynamically typed language". In Proc. 18th ACM Symposium on Principles of Programming Languages, 1991.

Aiken A. et al.
"Soft typing with Dependent Types", In Proc. 18th *ACM Symposium on Principles of Programming Languages*, 1994.

Ait-Kaci H. and Podelski A.
"Towards a meaning of LIFE", *Journal of Logic Programming*, 1993.

Allen J.
The Anatomy of Lisp, McGraw-Hill, 1978.

Andrews P.B.
"Theorem Proving via General Matings", JACM 28, 2, 1981.

Aubin R.
"Mechanising Structural Induction", Theoretical Computer Science, 1979.

Baber R.L.
The Spine of Software: Designing Provably Correct Software, *Theory and Practice*, Chichester: Wiley, 1987.

Backhouse R.
"Constructive Type Theory: a perspective from computing science" in Dijkstra (ed.) 1990.

Backus J.
"Can Programming be liberated from the von Neumann style? A functional style and its algebra of programs", *CACM*, 1978.

Baeten J. and J. Bergstra
"Term Rewriting Systems with Priorities", in *Rewriting Techniques and Applications*, LNCS 256, Springer-Verlag, 1986.

Barendregt H.P.
The Lambda Calculus: its syntax and semantics, North Holland, 1984.

Barendregt H.P.
"Lambda Calculus with Types", in Abramsky, Gabbay and Maibaum, vol. 2, 1993.

Basin D.A. & Kaufmann M.
"The Boyer-Moore Prover and NuPrl: An Experimental Comparison", *Logical Frameworks*, CUP, 1991.

Beckert B. and Posegga J.
"leanT^{A}P: Lean Tableau-based Deduction", JAR 1997.

Bibel W.
"On matrices with connections", J. ACM 1981.

Bibel W. and Jorrand P.
Fundamentals of Artificial Intelligence: an advanced course, Springer-Verlag, 1986.

Bird R.S.
"The Promotion and Accumulation Strategies in Transformational Programming", *ACM Transactions on Programming Languages and Systems* 6, 1984.

Bird R. and Wadler P.
Introduction to Functional Programming, Prentice-Hall, 1998.

Blasius K.H. and Burckert H.J.
Deduction Systems in Artificial Intelligence, Ellis Horwood, 1989.

Bobrow D.G. and Winograd T.
"An Overview of KRL", *Cognitive Science* I, 1977.

Boolos G.S., Burgess J.P. and Jeffrey R.C.
Computability and Logic, 4th edition, Cambridge University Press, 2002.

Boyer R. S. and Moore J. S.
A Computational Logic, Academic Press, 1979.

Boyer R. S. and Moore J. S.
A Computational Logic Handbook, Academic Press, 1997.

Bratko I.
Prolog Programming for Artificial Intelligence, Addison-Wesley, 2000.

de Bruijn N.G.
"Lambda Calculus Notation with Nameless Dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem", Indag. Math., 1972.

Bundy A.
The Mechanisation of Mathematical Reasoning, Academic Press, 1983.

Bundy A., Stevens A., Harmelen F., Ireland A., & Smaill A.
"Rippling: A heuristic for guiding inductive proofs", *Artificial Intelligence* 1993.

Burstall R. A and Darlington J.
"A Transformation Program for Developing Recursive Programs", *Journal of the ACM*, 24, 1977.

Cardelli L.
"The Functional Abstract Machine", *Polymorphism*, vol. 1., #1, 1983.

Cartwright R. and Fagan M.
"Soft typing". In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991, 278--292.

Cerrito S. and Kesner D.
"Pattern Matching as Cut Elimination", *Logic in Computer Science*, 1999.

Chang C. and Lee R.
Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

Charniak E. and McDermott D.
Introduction to Artificial Intelligence, Addison Wesley, 1985.

Church A.
"A Note on the Entscheidungsproblem", *Journal of Symbolic Logic*, 1936.

Church A.
The Calculi of Lambda Conversion, Princeton University Press, 1941.

Colby K.
Artificial Paranoia, New York: Pergamon Press, 1975.

Cooke D.J. and Bez H.E.
Computer Mathematics, Cambridge University Press, 1984.

Darlington J., Henderson P., and Turner D.A.
Functional Programming and its Applications, an advanced course, Cambridge University Press, 1982.

Davis M. and Putnam H.
“A Computing Procedure for Quantification Theory”*, Journal of the Association of Computing Machinery*, 1960.

DeMillo R., Lipton R., and Perlis A.
“Social Processes and Proofs of Theorems and Programs”, *Communications of the ACM* 22, 1979.

Dijkstra E.W.
Formal Development of Programs and Proofs, Addison-Wesley, 1990.

Diller A.
Compiling Functional Languages, John Wiley, 1988.

Diller A.
Z: an Introduction to Formal Methods, John Wiley, 1990.

Dreyfus, H. L.
What computers can’t do : a critique of artificial reason, New York: Harper & Row, 1972.

Dreyfus, H. L.
What computers still can’t do: a critique of artificial reason Cambridge, Mass. London: MIT Press, 1992.

Duffy D.
Principles of Automated Theorem Proving, Wiley, 1991.

Eder E.
“An Implementation of a Theorem Prover Based on the Connection Method”, *Artificial Intelligence*, ed. Bible and Petkoff, North-Holland, 1985.

Eisinger N. and Nonnengart A.
“Term Rewriting Systems”, in Blasius and Burckert, 1989.

Faé M. I. and Tarver M.
“Wardrop’s Principle Revisited: a multiagent approach”, *ANPET 2002*.

Ferber J.
Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence, Addison Wesley, 1999

Fetzer J.H.
Communications of the ACM 31, 1988.
Artificial Intelligence: its Scope and Limits, Kluwer Academic Publishers, 1990.

Field A. J. and Harrison P. G.
Functional Programming, Addison-Wesley, 1988.

Fitting M.
Proof Methods for Modal and Intuitionistic Logics, Synthese library, 1983.

Futumara Y.
"Partial Computation of Programs", in Goto ed. *RIMS Symposia on Software Engineering and Science*, LNCS, Springer-Verlag 1983.

Gabbay D.M. and Robinson J.A.
Handbook of Logic in Artificial Intelligence and Logic Programming Hogger C.J. Vols 1-6, Clarendon Press, Oxford, 1993.

Gallier J.N., Plaisted D., Raatz S. and Snyder W.
"An algorithm for finding canonical sets of ground rewrite rules in polynomial time", *Journal of the ACM*, 1993.

Gabrial R.P.
"Lisp: Good News, Bad News, How to Win Big"*, European Conference on Practical Applications of Lisp*, 1990.

Gardner M.
Logic Machines and Diagrams, University of Chicago Press, 1983.

Gentzen G.
Investigations into logical deduction (first published 1934) in *The Collected Papers of Gerhard Gentzen* (Szabo, ed.), Amsterdam, North Holland 1969.

Giarratano J.C. and Riley G. D.
Expert Systems: Principles and Programming (3rd edition), PWS, 1998.

Girard J.Y., Lafont Y. and Taylor P.
Proofs and Types, Cambridge University Press, 1989.

Gordon M.J.
Programming Language Theory and its Implementation, Prentice Hall, 1988.

Gordon M.J., Milner R. and Wadsworth P.
Programming in the Edinburgh LCF, Springer-Verlag, 1979

Green J. A.
Sets and Groups, Library of Mathematics, Routledge and Kegan Paul, 1965.

Green C.
Theorem-proving by resolution as a basis for question-answering systems, Machine Intelligence 4, 1969.

Grune D. and Jacobs C.J.H.
Parsing Techniques: a practical guide, Ellis Horwood, 1991.

Gunter C.
The Semantics of Programing Languages: Structures and Techniques, MIT Press, 1992.

Haack S.
Deviant Logic, Cambridge University Press, 1974.

Haack S.
Philosophy of Logics, Cambridge University Press, 1978.

Haynes C.
"Logic Continuations", *Journal of Logic Programming* 4, 1987.

Hendrix G.
"Encoding Knowledge in Partitioned Networks" in *Associative Networks*, ed. N. Findler, Academic Press 1979.

Henglein F.
"Global tagging optimisation by type inference" in ACM Conference on LISP and Functional Programming, 1992.

Henkin L.
"Systems, Formal and Models of", in Edwards P. (chief editor), *The Encyclopaedia of Philosophy*, MacMillan and Free Press, 1967.

Henson M.C.
Elements of Functional Languages, Blackwell, 1987.

Hewitt C.E.
"PLANNER: A Language for Proving Theorems in Robots", *IJCAI* 1969.

Hindley J.R. and Seldin J.P
Introduction to Combinators and Lambda Calculus,. Cambridge University Press, 1986.

Hodges W.
Logic, Pelican, 1977.

Hogger C.J.
Introduction to Logic Programming, London, Academic Press, 1984.

Hogger C.J.
Essentials of Logic Programming, Oxford, Clarendon, 1990.

Holmes B.J.
Introductory Pascal, DP Publications, 1993.

Hughes G. E. and Cresswell M.J.
An Introduction to Modal Logic, Methuen, 1968.

Hughes G. E. and Cresswell M.J.
A Companion to Modal Logic, Methuen, 1984.

Hughes G.E. and Cresswell M.J.,
A New Introduction to Modal Logic, Routledge, 1996

Hughes J.
"Why Functional Programming Matters", in Turner D.A. (ed.), *Research Topics in Functional Programming*, Addison-Wesley, 1990.

Ito T. and Halstead R.H.
Parallel Lisp: Languages and Systems, LNCS 441, Springer-Verlag, 1989.

Jackson P.
Introduction to Expert Systems, International Computer Science Series, 1999.

Jones N.D., Sestoft P. and Sondergaard H.
"An Experiment in Partial Evaluation: The Generation of a Compiler Generator", in Jouannaud 1985.

Jorrand P.
"Term Rewriting as a Basis for the Design of a Functional and Parallel Programming Language", in W.Bibel and Jorrand P. (ed.) 1986.

Jouannaud J.P.
Rewriting Techniques and Applications, LNCS, vol. 202, Springer-Verlag 1985.

Kac M. and Ulam S.
Mathematics and Logic, Pelican, 1971.

Keene S.E.
Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS, Addison-Wesley, 1989.

Kernighan B.W. and Plauger P. J.
"Ratfor – a Preprocessor for a Rational Fortran", *Software, Practice and Experience*, 1975.

Kleene S.C.
Introduction to Metamathematics, North Holland, 1952.

Knight K.
"Unification: a multidisciplinary survey", in *ACM Computing Surveys*, 1989.

Knuth D.E. and Bendix P.B.
"Simple Word Problems in Universal Algebras", in Leech (ed.), *Computational Problems in Abstract Algebras*, Pergamon Press, 1970.

Knuth D.E.
The Art of Computer Programming, *Sorting and Searching*, Addison-Wesley, Reading, Massachussetts, 1998

Kounalis E. and Rusinowitch M.
"Mechanising Inductive Reasoning", Bulletin of the European Association for Theoretical Computer Science.

Kowalski R.
Logic for Problem Solving, North Holland, 1979.

Kripke S.
"Identity and Necessity", in Munitz M. (ed.), *Identity and Individuation*, New York 1971.

Lajos G.
"Language Directed Programming in Meta-Lisp", *First European Conference on Practical Applications of Lisp*, 1990.

Landin P.
"The Mechanical Evaluation Of Expressions", *Computer Journal*, 6, 1964.

Lemmon E. J.
Beginning Logic, Hackett Publishing Co., 1978.

Lenat D. B. and Guha R. V.
"The Evolution of CYCL, the CYC Representation Language", *SIGART Bulletin*, 1991.

Linsky L. (ed.)
Reference and Modality , Oxford University Press, 1975.

Liu Y. and Staples J.
"btC: an Extension of C with Backtracking", *Proceedings of the Sixth International Conference on Symbolic and Logical Computing, Dakota State University*, 1993.

Lam C.W.H.
"How Reliable is a Computer Based Proof?" *Mathematical Intelligencer* 12, 1,1990.

Lipson J.D.
Elements of Algebra and Algebraic Computing, Benjamin Cummings, 1984.

Lloyd J.
Foundations of Logic Programming, Springer-Verlag, 1990.

Lopes R.H.C.
"Inductive Generalisation of Proof Search Strategies from Examples", Ph.D. thesis, School of Computer Studies, University of Leeds, 1998.

Loveland D. W. (ed)
Automatic Theorem Proving: After 25 years, American Mathematical Society, 1984.

Maier D. and Warren D.S.
Computing with Logic, Benjamin/Cummings, 1988.

Martelli A. and Montenari U.
"Unification in Linear Time and Space", Internal Report B76-16, Insituto di Elaborazione di Informatione, Pisa, Italy, 1973.

McCarthy J.
"Recursive Functions, Symbolic Expressions and their Computation by Machine", *Comm. of the ACM*, 1960.

McCarthy, J.
"History of Lisp". *ACM Sigplan Not.* 13,8 (Aug. 1978), 217-223.

McCarthy J.
"Circumscription: a form of non-monotonic reasoning", *Artificial Intelligence* 13, 1980.

McDermott D. and Doyle J.
"Non-monotonic logic", *Artificial Intelligence* 13, 1980.

McDermott D.
"A Critique of Pure Reason"*, Computational Intelligence 3*, 1987.

Mendelson E.
Introduction to Mathematical Logic, Kluwer Academic Publishers, 1987.

Minsky M.
"A Framework for Representing Knowledge", in Winston P. (ed.), *The Psychology of Computer Vision*, McGraw-Hill, 1975.

Murch R. and Johnson T.
Intelligent Software Agents, Prentice Hall, 1998.

O'Donnell M.J.
Equational Logic as a Programming Language, MIT Press, 1985.

Paulson L.
ML for the Working Programmer, Cambridge University Press 1996.

Peirce B.
Types and Programming Languages, MIT Press, 2002.

Pettorossi A.
"A Powerful Strategy for Deriving Efficient Programs by Transformation" *ACM Symposium on LISP and Functional Programming*, 1984.

Peyton-Jones S. and Lester D.
Implementing Functional Languages, Prentice-Hall, 1987.

Plaisted D.
"Equational Reasoning and Term Rewriting Systems" in Gabbay, Hogger and Robinson, vol. 1, 1993.

Post E.
"Formal Reductions of the General Combinatorial Decision Problem", American Journal of Mathematics, vol. 65, 1943.

Quillian M.R.
"Semantic Memory", in Minsky M. (ed.), *Semantic Information Processing*, MIT Press, 1968.

Reeves S. and Clarke M.
Logic in Computer Science, Prentice-Hall, 1990.

Reiter R.
"A Logic for Default Reasoning", *Artificial Intelligence* 13, 1980.

Robinson J.
"A Machine-Oriented Logic Based on the Resolution Principle", *Journal of the ACM*, 1965

Robinson J.A. and Sibert E.E.
"Logic Programming in Lisp", research report, School of Computer and Information Science, Syracuse University, New York, 1980.

Russell S. and Norvig P.
Artificial Intelligence: a modern approach, Prentice Hall Series in Artificial Intelligence, first edition 1995, second edition 2002.

Schwartz J.T., Dewar R.B.K., Dubinsky, E., and Schonberg, E
Programming with Sets: an introduction to SETL, Springer-Verlag, 1986.

Shrobe H.
"Symbolic Computing Architectures", *Exploring Artificial Intelligence*, ed. Shrobe and AAAI, Morgan Kaufmann, 1988.

Simon H.
Models of my Life, New York, Basic Books, 1991.

Sitaram D.
"Handling Control", Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1993.

Sloman A.
"POPLOG: a multi-purpose multi-language program development environment", *Artificial Intelligence - Industrial and Commercial Applications. First International Conference*. 1985

Smullyan R.
First Order Logic, Springer-Verlag, 1968.

Smullyan R.
Theory of Formal Systems, Princeton University Press, 1960.

Sterling L. and Shapiro E.
The Art of Prolog, MIT Press, 1994.

Stickel M.E.
"A PROLOG Technology Theorem-Prover" in *New Generation Computing 2*, 1984.

Stickel M.E.
"A Prolog Technology Theorem Prover: Implementation by an extended Prolog Compiler", *CADE 8*, LNCS vol 230, Springer-Verlag, 1986.

Stickel M.E.
"An Introduction to Automated Deduction" in W. Bibel and P. Jorrand (ed.), 1986.

Tarver M.
"Towards a Unified Theorem Proving Language", M.Sc. thesis, *University of Leeds, School of Computer Studies*, 1989.

Tarver M.
"The World's Smallest Compiler-Compiler", *European Conference on Practical Applications of Lisp*, 1990.

Tarver M.
"A Rationalisation of Lisp", *University of Leeds, School of Computer Studies Report 90.19*, 1990.

Tarver M.
"An Algorithm for Inducing Tactics from Proofs", *University of Leeds, School of Computer Studies Internal Report,* 1992.

Tarver M.
"A Language for Implementing Arbitrary Logics", *IJCAI*, 1993.

Tarver M.
"Representing the Types of Polyadic Lisp Functions", in *International Lisp Conference* 2002.

Tecuci G.
Building Intelligent Agents, Academic Press, 1998.

Thompson S.
Miranda: the Craft of Functional Programming (second Edition), Addison-Wesley, 1995.

Thompson S.
Haskell: the Craft of Functional Programming (second Edition), Addison-Wesley, 1999.

Turner D. A.
SASL Language Manual, University of St Andrews Dept. of Computer Science Report, 1976.

Turner D. A.
"A New Implementation Technique for Applicative Languages", in *Software Practice and Experience*, 1979.

Turner D. A.
"Recursion Equations as a Programming Language", in Darlington, Henderson and Turner, 1982.

Turner D. A.
"An Overview of Miranda" in Turner (ed.) 1990.

Turner D. A. (ed.)
Research Topics in Functional Programming, Addison-Wesley, 1990.

Walther C.
"Many-sorted unification", *Journal of the ACM*, 1988.

Wand M.
"Continuation Based Program Transformation Strategies", *Journal of the ACM*, 27, 1980.

Wardrop, J.G.
"Some Theoretical Aspects of Road Traffic Research", in *Proceedings of the Institute of Civil Engineers*, Part II, 1952.

Wikstrom A.
Functional Programming Using Standard ML, Prentice-Hall, 1987.

Winograd T.
"Understanding Natural Language", *Cognitive Psychology* 1972.

Wright A. and Cartwright R.
"A practical soft type system for Scheme", *1994 ACM Conference on LISP and Functional Programming*, 1994.

Index