Abstractions and Paradigms for Programming

Abhiram Ranade

September 2005

Preface

This is a draft-in-progress for a textbook for a course on *Abstractions and Paradigms for Programming*. It is intended as an introductory course or a second course in programming for CS majors. The focus of the course is not on a taxonomic study of the different paradigms, but rather the paradigms are viewed as schools of thought in the broader goal of improving the program development process. To motivate the different paradigms, a number of programming problems are considered, drawn from many areas of computer science. Because of this, the course also plays the role of an *introduction to computer science*. Of course, no area of computer science is covered in great depth; however the coverage is such that the students get some indication of the depth and excitement of the different areas, and can also do a substantial programming assignment.

0.1 Course Overview

The programming language used in the course is Scheme, which because of its flexibility allows many paradigms to be embedded into it. An equally important reason is that the built-in list input-output enables students to work on symbolic, structurally complex data without having to first learn parsing.

The first paradigm studied is the Functional programming paradigm. Even though *pure* functional programming is not very popular, many of its concepts such as applicative (stateless, or global-variable-less) programming, use of high order functions are very common. Recursion is one of the dominant themes. Examples of recursive programs are drawn from graphics, sorting, elementary numerical computation etc. Recursion is also introduced as an algorithm design tool, and its relationship to dynamic programming is briefly considered[1]. Structural recursion is also another dominant theme. Programming examples studied here include the use of lists to represent arithmetic formulae and programs, and the operations on these (recursively structured) representations such as symbolically differentiating the formulae, or rendering them on the screen (i.e. mathematical typesetting), processing programs to implement one program construct in terms of another. Use of higher order functions such as maps, list reduction functions is encouraged. Higher order functions are used to explore the (algebraic) relationship between matrix multiplication and shortest paths. The relationship between the basic list operators like map, filter, reduce and connections to relations and relational operators is also studied.

[1] Although algorithm design is not the main theme, this relationship is almost syntactic, and is therefore of interest in a course focussing on program expression.

Some basic ideas in proving the correctness of functional (recursive) programs are discussed. These are contrasted with the proof techniques for imperative (loop based) programs. For the few examples considered, the functional programs are seen to be easier to understand and prove correct as compared to the imperative counterparts.

The next paradigm to be discussed is the Object Oriented Paradigm. The major ideas discussed here are data abstraction and encapsulation and inheritance. Object oriented program design issues such as construction of the class heirarchy are discussed. The idiom of container classes and delegating operations to the contained objects is discussed. The first motivating applications considered here is the symbolic differentiation program (which was earlier developed using a simple functional programming style). One of the advantages of the object oriented model is seen to be the ability to add new features (new classes of functions that can be differentiated) *without* changing any of the old code. A logic circuit simulator is also developed. This uses the inheritance facility, as well as includes the notion of container classes (more complex components *contain* simple components).

The final paradigm to be discussed is the Constraint Programming paradigm. This consists of a simple language for defining constraints on a set of variables and a general purpose solver which can be used to find satisfying solutions. The language is rich enough to express NP-hard problems, and the solver thus has to be brute-force in some manner. In the course we only consider the *finite domain* version, i.e. in which each variable is constrained to take values from a finite domain. The solver we have is basically based on backtrack search and is used to solve a number of optimization problems relating to scheduling, packing, covering, solving logic puzzles, cryptarithms.

This is more of a special purpose paradigm, but it was considered nevertheless for a number of reasons. First, Constraint solving systems seem to be becoming more and more popular in many communities such as Operations Research, Optimization. Second, studying the paradigm provides a reason to formulate and solve (the solver is good enough for small problems) many interesting (and fun!) problems, which otherwise remain outside the reach of students in an introductory course. For many students just the exercise of precisely formulating the problems abstractly and noticing relationships amongst the formulations is an interesting experience. Third, the paradigm is in some sense halfway to the Logic Programming Paradigm (i.e. it includes backtracking but not Unification). [2] Fourth, the solver algorithm, i.e. backtracking and its variations can be explained to students.[3] Finally, it should be noted that overall the evolution of programming paradigms has been towards the automation of more and more tasks and the consequent reduction of the work needed to be done by the programmer. The Constraint Programming Paradigm is an excellent example of this trend (probably more so than Logic Programming itself). While the Functional programming paradigm and the Object oriented programming paradigm were substantially different from an expressiveness point of view, the execution of the program was substantially under program control. The Constraint programming paradigm exposes the students to some of the issues which arises when the software system being used is substantially intelligent.

The last major topic concerns the design of paradigms itself. How can we implement a special purpose language constructs on top of a base language? The problem of extending

[2] One of the Probably there isnt enough time to cover Logic Programming with much depth. Besides, it is apparently covered substantially in the formal methods course.

[3] Explaining Prologoue unification is much harder, and probably best left for a later course.

the basic graphics features provided in Scheme so as to allow a convenient ability to specify transformations (preferably incrementally/recursively) is considered. A number of ways of providing this functionality are considered: the simplest being a set of library functions, to a complex implementation based on Scheme macros. The implementation based on macros is seen to be most convenient to use and least error prone. Traditionalists may consider macros to be hacks, but Scheme macros are reasonably clean and can be used to bring out execution order issues. Macros are also a necessary complement to the symbolic processing mentioned earlier. The symbolic processing ideas studied earlier allow students to write code which, for example, translate a fortran style do loop into a recursive function. By calling the code through a macro, the do loop can actually be executed.

A note on what is not covered and perhaps ought to be. (Add your suggestions here!) Since the language is Scheme, there is no static type checking. So the students do not get familiar with this aspect. Lazy evaluation as in functional programming is not covered. My rationale is that this is a slightly advanced feature, which is a bit difficult to motivate especially since it is not available in any standard language such as C++/Java.[4]

[4]I am aware of some of the elegant uses of lazy evaluation as *glue* for composing programs, however, in some ways I consider that material a bit heavy for first year students.

Contents

0.1 Course Overview 2

1 Abstractions and Paradigms: Course Overview **10**
1.1 The program development process 11
1.1.1 Modelling or Representation 12
1.1.2 Issues in developing large programs: Problem Decomposition 12
1.1.3 Issues in developing small programs 13
1.1.4 Identifying, representing and exploiting patterns 13
1.1.5 Program Readability 14
1.2 The Paradigms 14
1.3 Course Overview 16

2 Introduction to Scheme **17**
2.1 Simple Scheme Session 17
2.2 Names and values 19
2.3 Commands and values 19
2.3.1 Arithmetic and Logic commands 19
2.3.2 Input-Output Commands 19
2.3.3 Graphics Commands 20
2.3.4 The `if` command 20
2.3.5 Examples 21
2.3.6 Nesting of commands: 22
2.4 General Remarks on the Scheme Interpreter 22
2.5 Exercises 23

3 Functional Programming **25**

4 Recursion **27**
4.1 How Recursive Programs Execute 28
4.2 Recursion vs. loops 29
4.3 Recursion in Graphics 29
4.4 Clever Recursion 29
4.4.1 Computation of the sine of an angle 30
4.4.2 Square root computation 30
4.4.3 Coin Change Problem 31
4.5 Efficiency and Recursion 32
4.6 *Tracing* Scheme procedures 33

4.7 Exercises . . . 34

5 Proving Program Correctness **35**
5.1 Basic ideas in program proving . . . 35
5.1.1 Form of the proof . . . 36
5.2 Proving correctness of functional programs . . . 36
5.3 Summary of the argument . . . 37
5.3.1 Euclid's Algorithm . . . 37
5.4 Proof Strategy for Imperative programs . . . 38
5.4.1 Structure of proofs of Imperative Programs: . . . 40
5.4.2 Euclid's algorithm for computing GCD . . . 40
5.5 Proving programs involving several functions . . . 41
5.6 Conclusions . . . 42
5.7 Exercises . . . 42

6 More on Scheme Statements **44**
6.1 The "begin" Statement . . . 44
6.2 The "cond" statement . . . 44
6.3 The "let" stetement . . . 45
6.4 Exercises . . . 46

7 Lists **47**
7.1 Program Example . . . 48
7.2 Proving correctness of programs involving lists . . . 49
7.3 More List Operations . . . 49
7.4 Applications of Lists: . . . 51
7.5 Procedures with variable number of arguments . . . 51

8 Higher Order Functions **54**
8.1 Functions as parameters to other functions . . . 54
8.1.1 Generalized Root Finder . . . 54
8.1.2 Animation . . . 55
8.2 Function Expressions . . . 56
8.3 Unnamed Functions . . . 56
8.4 Exercises . . . 57

9 Built-in High Order Functions for lists **59**
9.1 Map . . . 59
9.2 Reduce: . . . 59
9.3 Filters . . . 60
9.4 Application of Filters: Quicksort . . . 61
9.4.1 Applications of Map and Reduce . . . 61

10 Trees **63**
10.1 Representing trees in Scheme . . . 64
10.2 Operations on trees . . . 64
10.3 Exercise . . . 64

11 Analysis of Algorithms **66**
11.1 Selection sort . . . 66
11.2 Analysis of smallest . . . 67
11.3 Analysis of remove . . . 67
11.3.1 Analysis of selsort . . . 68
11.4 Exercises . . . 68

12 Matrices And Shortest Path Problems **69**

13 Symbolic Computing **72**
13.1 Uses of s-expressions . . . 72
13.1.1 Symbolic Differentiation . . . 73
13.1.2 Processing Programs . . . 74

14 More on Higher Order Functions **75**
14.1 Functions returning functions . . . 75
14.1.1 Simulation of Data Structures . . . 75
14.1.2 Simulating let command using functions . . . 76
14.1.3 Simulating do commands using functions . . . 76
14.2 Unnamed recursive functions . . . 77
14.2.1 Exercise . . . 78

15 Relations **79**
15.1 Selection . . . 79
15.2 Projection . . . 79
15.3 Count . . . 80
15.4 Cross Product . . . 80
15.4.1 Applications of the cross-product . . . 80
15.5 The JOIN function . . . 81
15.6 Examples . . . 81

16 Memoization **82**

17 Embedded Languages **84**
17.1 Macros . . . 85
17.1.1 Convenient notation for writing macros . . . 85
17.1.2 Nesting and recursion . . . 87

18 Embedding Advanced Graphics into Scheme **90**
18.1 Algorithmic Issues . . . 90
18.2 User Interface . . . 91
18.2.1 User Interface 1 . . . 92

18.2.2 User Interface 2 . . . 93
18.2.3 User Interface 3 . . . 94

19 Object Oriented Programming **95**
19.1 Aggregation . . . 95
19.2 Data Abstraction . . . 96
19.3 Function overloading . . . 97
19.4 Code evolution . . . 97
19.4.1 Example 1: Library information system . . . 97
19.5 Example 2: Extensions to Deriv . . . 98
19.6 Basics of OOP . . . 98
19.6.1 Classes . . . 98
19.6.2 Inheritance . . . 99
19.6.3 Defining Classes . . . 99
19.6.4 Creating Instances . . . 100
19.6.5 Defining Methods . . . 100
19.6.6 Remark: Incremental specification of functions . . . 102
19.7 Utility Functions . . . 102
19.8 Exercises . . . 103

20 Object Oriented Design **104**
20.1 A Biology Tutor . . . 104
20.1.1 Design 1 . . . 104
20.1.2 Design 2 . . . 105
20.1.3 General Principles . . . 106
20.2 Simulation . . . 107
20.3 Exercises . . . 109

21 Constraint Logic Programming (CLP) **111**
21.1 Finite Domain CLP . . . 112
21.2 Initializing the CLP Solver . . . 113
21.3 Definition of domain variables . . . 113
21.3.1 Quadratic Equation Problem: . . . 113
21.3.2 Magic Square Problem: . . . 113
21.4 Specification of constraints . . . 114
21.4.1 Quadratic Equation Problem: . . . 114
21.4.2 Magic Square Problem: . . . 114
21.5 Invoking the solver . . . 115
21.5.1 Printing the solutions: . . . 116
21.5.2 Quadratic Equation Problem: . . . 116
21.5.3 Magic Square Problem: . . . 116
21.6 Finding a solution vs. finding an optimal solution . . . 117
21.7 Exercises . . . 117

22 Implementation of Our CLP Solver **119**
22.1 A Brute Force Strategy 120
22.1.1 Refinement 1 120
22.1.2 Refinement 2 121
22.1.3 Other Refinements 122
22.1.4 Recursive formulation 122
22.2 Design of solvers 123
22.2.1 Our solver 123
22.3 On writing CLP programs 124
22.3.1 Guidelines for our solver 124
22.4 Guidelines vs. rigid dictums 125
22.5 Exercises 125

23 Advanced topics in CLP **127**
23.1 Knapsack Problem 127
23.1.1 Solving for a fixed V 128

Chapter 1

Abstractions and Paradigms: Course Overview

What is a *good* computer program? Obviously, a good program must correctly produce the desired answers, but are there other features that it must have? For example, you could insist that a program is good if it runs fast, or if it is easy to use. Or, if you are a programmer, you could say that a program is good if it is possible to use it to develop more complex programs later. As you can see, the answer depends upon the viewpoint, or the *paradigm*. Once we take a position on what is "goodness", it is natural to ask, how do we go about writing good programs. Over time a variety of definitions of goodness have emerged, and for each definition a set of program development strategies that help write good programs have evolved. Each notion of what constitutes a good program has led to a distinct theory of how to write programs. Which theory should an ordinary programmer follow? Would it not be nice if we could develop a single *best* theory? Unfortunately, no single universally suitable theory has been found, and thus it is necessary to study different theories, each of which might be appropriate under different circumstances. Multiple paradigms or schools of thought are common in other domains as well, e.g. literature, painting, designing a building and so on. So for example in painting there are paradigms such as *realism*, *Impressionism* and *cubism*. In literature there are paradigms such as *Magical Realism* (popularized in the work of Gabriel Garcia Marquez or Salman Rushdie).

Programming paradigms have evolved substantially over the years. Fifty years ago, computers were very expensive and very slow, and most often the program was used by its own author and few others. Good programs were those that executed as fast as possible. Today, the circumstances are very different: computers are very fast and very cheap. Further, computers are pervasive – millions of people (non-specialist users who do not know much programming) use them for a very wide range of purposes. Thus, the program development process of today gives less importance to speed of execution and more importance to reliability and ease of use of programs. Since programmer effort has become more expensive (in relative terms), a programming process that improves programmer productivity (possibly at a cost of execution speed) is becoming more important.

To understand what is a good programming paradigm, it is also necessary to have seen a large number of relatively large, complex programs of different kinds. So our course will do that as well. We will write programs to solve a number of problems of varying hardness from

a number of diverse subjects. While studying these problems, we will consider the question, *what is the best way to write these programs?*. So we will get a general introduction to copmputer science as well as programming paradigms.

We begin by considering some of the basic questions that any programing paradigm must address. We then give a overview of the programming paradigms studied in the course. We conclude this chapter by giving an overview of the other aspects of this course.

1.1 The program development process

What paradigm to use for developing programs depend upon what we wish to optimize (e.g. speed, ease of understanding, ease of developing the program) as well as the kind of programs we are to write, e.g. a program with 20 lines versus a program with thousands of lines. The paradigm must thus cater to a variety of demands.

To make the discussion more concrete, consider the following program development problems.

1. Find the greatest common divisor of given numbers.

2. The programming language you use does not have primitives for manipulating complex numbers. Develop a library using which complex numbers can be added, subtracted etc.

3. An electrical engineer gives you the description of a circuit, and the values to be fed to its inputs. He has designed the circuit so that it will produce a certain set of output values. He would like to have a program which verifies if those values will indeed be produced, i.e. to check if his circuit design is correct.

4. A tax consultant keeps records of his clients incomes etc., and also the government rules on tax computation. He would like to have a program which reads in the records and the rules and computes the tax. The computed tax values must be stored back into his computer for future reference, as well as printed out.

5. A textbook is to be developed. It contains complex mathematical formulae, say something like:

$$Q_{lm} = \sqrt{\frac{4\pi}{2l+1}} \int \rho(\mathbf{r}') r'^{l} Y_{lm}(\Omega') d^3\mathbf{r}'$$

   You are to write a program which allows the author to typeout the formula in a suitable notation (which you must devise) and then select the proper character sizes and their positions and draw out the formula on paper.

6. You are to write a program to solve the following puzzle. For each of the letters in the "sum" given below, the program must find a unique digit so that we have a valid arithmetic expression.

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \\ \hline \end{array}$$

As you can see, programs can be of very different types. Nevertheless, we can make some general remarks about how to go about developing them.

1.1.1 Modelling or Representation

The first question in designing a program is, how do we represent the given information on a computer? So for example, how do we represent a complex algebraic formula like the one given above? How do we represent the information given in the puzzle?

Sometimes it might seem that there is an obvious way to represent the given information on a computer. For example, a complex number could be represented as a pair of numbers: one representing the real part, and the other the imaginary part. However, in this case, there is another representation as well: we could represent the number in the polar representation, i.e. by giving its distance from the origin and the angle. Which represenation is the right one? That depends upon what you need to do with the representation: for complex numbers the polar representation is favoured if you want to do multiplication; the cartesian one for addition.

Another example: the n-queens problem. We could represent the board "naturally" as a 8×8 array. Or, noting that we can only place one queen in each row, we could just have a single dimensional array of length 8, the ith element of the array indicating the column position of the queen in the ith row.

For many problems, it is first important to understand the underlying *mathematical* structure before determining what the right computer model is. For example, suppose we are given registration lists for a number of courses and we need to construct a timetable for the courses. Then it might be useful to realize that this problem is similar to the so called graph colouring problem.[1] By writing down a problem in mathematical notation, we can be sure that we have understood all its intricacies; also the process might reveal that our problem is similar to another problem that has been solved earlier.

1.1.2 Issues in developing large programs: Problem Decomposition

Any reasonably large problem is solved only by breaking it into smaller problems. The hope is that the smaller will be simple enough so that it is clear how to solve them. Otherwise we might have to break apart the smaller problems themselves. There are many ways to do this.

One idea is to decompose functionally. For example, in the circuit simulation problem, one could conceive of the entire program as consisting of 3 steps:

1. Read in the information regarding the circuit and the input values.

2. Devise a program which computes the next state of the circuit given the current state, and the current inputs. Execute this for as many steps as needed.

[1] A graph colouring problem is like a map colouring problem. In map colouring countries which are geographically adjacent must be assigned distinct colours. In timetabling, define two courses to be adjacent if they share a student. So all we need to do now is give distinct time slots to "adjacent" courses.

3. Print out the required output values.

The computation of the next state is itself a complex step, especially if the values have to be shown on the screen as they change. So this might have to be decomposed further. Here the decomposition could be according to the type of the component: so we might have a `calculate-next-state` function for an AND gate, another for a NOR gate, and so on.

When we decompose a problem (say P) into subproblems (say P1 and P2), it is important that we clearly write down how the subproblems interact, i.e. what is the *interface* between the problems. The interface can be thought of as a set of guarantees and expectations; what P1 provides to P2 and expects from P2, and vice versa. Once we decide what the interface is, we can work on the two parts independently. Now, while building P1 we do not worry about how P2 is built; we simply take it for granted that however P2 is built, it will adhere to the stipulations in the interface. Likewise, when we build P1, we make sure that it does provide to P2 the services mentioned in the interface.

A clearly defined interface has other advantages also. Suppose tomorrow we want to change the implementation of to P1 (say because we have found a better implementation) then can we do it without affecting the implementation of P2? If our implementations always adhere to the interface, then we should be able to change P1 (such that the new implementation adheres to the interface), *without changing P2 at all!*

Some ways of decomposing a problem are very clever– the study of this belongs to the course on Algorithm Design. Our concern here is, suppose we know how to decompose a problem, can we express the decomposition of our choice using our programming language? Clearly a programming paradigm/language which allows more kinds of problem decompositions to be expressed conveniently is better. This could also be stated as: a paradigm which allows more ways to glue together small programs to make a larger program is better.

This general idea of solving problems by decomposing them into smaller problems with well-defined interfaces is used in other disciplines as well and is often called *modular design*.

1.1.3 Issues in developing small programs

The main concerns in developing small programs (i.e. programs which are short to begin with, or those arising after partitioning large problems) are which statements to use, how do we make sure that the idea used in the program is expressed lucidly.

For example, considering the problem of computing the gcd of two numbers. This could be written out using a single loop, or it could be written recursively. Each of these expressions might be natural in different programming paradigms; and both have certain advantages. For example the loop based program might perhaps run fast, the recursive program might be very easy to understand, or to certify bug-free.

1.1.4 Identifying, representing and exploiting patterns

Most problems contain patterns – say in the actions to be performed (e.g. same action several times) or in the data (e.g. a circuit can be thought of as made of smaller circuits, each of which of even smaller circuits) and so on.

Understanding these patterns and representing them in the program is extremely important. The benefit of observing that the same action is repeated is obvious: we could write

the code for the action once, and then simply reuse it as many times as needed. Patterns in data are also valuable; in this case we can use a *recursive* strategy which will be discussed shortly. More simply, if we observe that two kinds of objects, say K1 and K2 constituting a program are very similar (but not identical) it might be useful to divide the code into 3 kinds, the one that works on both K1 and K2, the one that work only for K1, and the one that works only for K2. This is preferable to developing completely separate code for K1 and for K2.[2]

Observing patterns is profitable also across different programs. For example, suppose the program you are currently writing uses ideas similar to the ones you used earlier, e.g. both programs use complex numbers. Then you should consider paying more attention to those ideas, and maybe reusing the code you developed there. Or if reuse is not directly possible, rewrite the code so that it works for both the programs, i.e. develop a proper library of subroutines for working on complex arithmetic. Complex arithmetic is only a small example. As another example, suppose you have already written code that uses Newton's method to find $\sqrt{n}$. Now in a new program you need to use Newton's method to solve equations such as $\tan(x) = x$. It would be good if you can plug in the new equation instead of the old (i.e. $x^2 - n = 0$) and make the code work.

Do note that the main point in using old tested code is saving effort in writing, as well as testing.

1.1.5 Program Readability

If a program is written so that it is easy to understand, then it is easier to modify if necessary. Suppose we need to fix bugs, or accomodate small changes in the functionality of the program. Then if the program is easy to understand it might be possible to modify it rather than having to rewrite it.

Some programming paradigms/styles make for more readable programs, and hence are more desirable in this age when programmers are scarce.

The term "readability" is usually used in an informal sense. However, we could make the issue more formal and also consider the question of whether one paradigm style of programming leads to programs that are easier to *prove* correct.

1.2 The Paradigms

We will study four programming paradigms, or approaches to developing programs, in this course. These are Imperative Programming, Functional Programming, Object Oriented Programming, and Logic Programming. We will mostly concentrate on the last three (since the first is fairly familiar through earlier courses), and even in these we will focus only on some of the major ideas. Many, many details (even some important ones) will be left undiscussed.

[2]Say you found a bug in the code for K1. Since K1 and K2 are similar, it is wise for you to check if the bug is present in K2 as well. If your code somehow mentions that K1 and K2 are similar, you are more likely to make the check; if the bug happens to be in the common portion, you will need to make only one change.

Imperative Programming:

In this, a program is thought of as *a sequence of instructions to be given to the computer*. This paradigm inspired the development of languages such as Fortran and C, although more recent versions of these languages support non-imperative programming styles to some extent. The main virtue of the paradigm is speed. Since the programmer more directly influences the actions of the computer (as compared to the other paradigms), a good programmer can write programs that execute fast. This is also a disadvantage – the programmer is expected to worry a lot about managing the computer in addition to modelling the application that is being programmed.

Functional Programming:

In this paradigm program execution is thought of as function application in mathematics. A program is simply a map (in the sense of a mathematical function) from the set of possible input instance to the desired output values. The task of programming is simply to define these functions, and this is accomplished by defining functions from scratch or composing predefined functions into new functions. Typically, programs written in a functional programming style are slower for the same job than their imperative counterparts; however they are often immensely more readable, elegant and easier to understand. The improved understandability is very important today.

Object-oriented programming:

This is really a sub-paradigm of the Imperative paradigm. One of the major ideas in it concerns program organization. In conventional programming languages it is customary to think of a program as consisting of data structure declarations and code. The OO paradigm encourages programmers to treat each data structure together with code for accessing it as a separate unit. Each such unit is called an *object*, and hence the name. OO programs are considered easier to modify and extend; this is because typically a modification will need changes to only one-two objects in the entire program. It is believed that OO programs are easier to design because the objects constituting a program usually have a natural correspondence with the objects constituting the real-life system that is being modelled in the program. For example, in a program concerned with circuits, there may be an object for each transistor.

Logic Programming:

This paradigm is becoming popular for solving problems in Artificial Intelligence, or optimization problems. We will study a subparadigm called *Constraint Logic Programming*, CLP.

In CLP the user only needs to state formally and to full detail the constraints that the values of the variables constituting the program must satisfy. The programmer does not need to worry about how such a satisfying set of values is to be found. The CLP system (which is some kind of an expert system) attempts to find a solution the problem using general techniques. A CLP system may attempt to reduce the user specified problem to problems from some class that it knows how to solve; or it might use brute force methods (e.g. trial

and error) to solve the problem. With advances in theoretical computer science, more and more powerful logic programming systems are being developed. For the foreseeable future, it is expected that programs written in this style will be substantially slower in execution than those in the other styles. However, it will also be much easier for the programmer to write programs in this style.

1.3 Course Overview

This course will study the paradigms described above. Each paradigm is the basis for several programming languages, e.g. the language ML is said to be based on the functional paradigm, the language Prolog on the logic programming paradigm, the languages C++ and Java on the Object Oriented Paradigm, and languages such as Fortran and C on the Imperative (non object oriented) paradigm. The most obvious way to study the paradigms is to study these languages.

Instead, in this course we will study the Scheme language in which all the four paradigms can be *embedded*. This is because Scheme is an unusually flexible language which can take on the features of different paradigms as needed. Scheme also has several other advantages including its ability to easily process non-numerical data. Learning the Scheme language (and how it can be extended as needed) will also be a part of the course.

During the course we will use examples from a variety of subfields in Computer Science such as Computer Graphics, Databases, Compilers, Circuit Design, Symbolic Algebra, Optimization and Artificial Intelligence. In other words, the course will also be a mini-survey of Computer Science.

Chapter 2

Introduction to Scheme

Scheme is a dialect of the programming language LISP (LISt Processing language). Scheme has been chosen for our course because of a number of reasons. You can embed other languages into it quite easily. It allows easy symbol manipulation. Scheme and Lisp are interpreted languages (ie. there is no intermediate object file).

This chapter will provide a brief introduction to scheme. We will begin with a very simple session with the scheme interpreter. Then we will describe the general structure of the scheme commands. Finally we will describe some commonly used scheme commands.

2.1 Simple Scheme Session

The scheme interpreter can be started by typeing "scheme" to the unix command shell. After that you can typein commands, which are executed by the interpreter. You can end the session by typeing CTRL-d.

After the scheme interpreter is invoked, the user must type commands through the keyboard which the interpreter then executes. The most common command is an *expression* which has the form "(command-name argument ...)". Simplest expressions are formed using the arithmetic operators. Thus you may type

```
(+ 10 35)
```

To which scheme will reply by printing out 45, which is the result of adding 10 to 35. Scheme uses prefix notation for expressions eg 10+35 is written as (+ 10 35). Commands can be nested. For example, you may type

```
(+ 35 (* 2 10))
```

which scheme will interprete as "add the 35 to the result of multiplying 2 and 10. Trigonometric functions can also serve as command names. As in most languages, trigonometric functions expect the arguments in radians. Thus here is a scheme expression to compute the trigonometric sine of 30 degrees:

```
(sin (/ (* 30 3.14 ) 180 ))
```

You can define names as follows:

```
(define pi 3.14159 )
```

This tells the Scheme interpreter to use 3.14 whenever pi is encountered subsequently. So following this definition we could just write

```
(sin (/ (* 30 pi) 180 ))
```

for computing the sine of 30 degrees. Thus pi could be used in place of 3.14 in previous expression. Previously defined names can be used to define new names, e.g.

```
(define twopi (* pi 2 ) )
```

You can also define new commands. Here is a command to calculate the volume of a cylinder:

```
(define (cylinder-volume r h)     ;; procedure definition
  (* pi r r h)                    ;;   procedure body
)                                 ;; end of procedure definition
```

The text following a `;` is a comment, and is ignored by the Scheme interpreter. It is assumed in the above definition that pi has been defined earlier. The above procedure could be invoked as follows:

```
(cylinder-volume 10 20)           ;; procedure invocation
```

The numbers 10, 20 in the invocation are called the arguments, and the corresponding names `r` and `h` in the definition of the procedure `cylinder-volume` are called formal parameters of the procedure. When the invocation is executed, the values of `r` and `h` are set to respectively 10 and 20, and the procedure body is executed. The body will cause the multiplication $3.14 \times 10 \times 10 \times 20$, and the result, 6280, will be returned. This is what Scheme will print as the value of the invocation. Scheme procedures can invoke other procedures, or even themselves, i.e. procedures can be recursive.

Besides expressions, scheme commands can be just references to a name: it is possible to type a name to the scheme interpreter, in this case the interpreter simply returns its value. So for example if you type

```
pi
```

then the scheme interpreter would respond by typing 3.14. If you type

```
cylinder-volume
```

most scheme interpreters would respond with a message saying that `cylinder-volume` is a *procedure*.

2.2 Names and values

In Scheme, names do not have an associated *type* as in Fortran or C; so there are no *type declaration* statements. A name can be associated with any kind of data. This is done using `define` statements. For example, the statement `(define pi 3.14)` causes `pi` to be associated with the number 3.14. It is also customary to say that the value of `pi` is 3.14.

Names can also be associated with procedures, for example, `cylinder-volume` as defined earlier is a name associated with the procedure for computing the volume of a cylinder, as defined using `(define (cylinder-volume r h) (* pi r r h))`. In fact it is acceptable to ask what the value of `cylinder-volume` is– in Scheme it is appropriate to say that the procedure to compute the cylinder volume is the value of `cylinder-volume`[1]

Scheme does have a command that changes the value of a name; we will look at this command later.

2.3 Commands and values

The term *command* suggests a phrase that causes some action to happen. Scheme commands can cause actions to happen, but in addition, many Scheme commands also represent *values*. For example, a command such as `(* 3.14 10 10)` not only causes the numbers 3.14, 10, 10 to be multiplied, but it also represents the resulting value, i.e. the number 314. This means that the entire command can be *nested* inside another command, in any place where a value is allowed. For example we may write `(+ 1 (* 3.14 10 10))`. The value of this command, of course, is 315.

Nesting one command inside another is allowed in most languages. However, Scheme allows much more interesting kinds of nesting. This is because almost every statement in Scheme has a value (see for example, the `if` statement described below), hence it can be written wherever a value is allowed.

We describe some selected commands below.

2.3.1 Arithmetic and Logic commands

Scheme has built-in commands for addition, subtraction, multiplication, division, and a host of other operations. The syntax is prefix, i.e. to compute 35/47, we would write `(/ 35 47)`. Prefix syntax is also used for comparisons. For example `(> a b)` would return true if $a > b$ and false otherwise. This will typically be used while writing an `if` statement as discussed below.

2.3.2 Input-Output Commands

Scheme does have a powerful set of input-output commands (see manual) – but they are often not needed.

Input commands are not commonly needed because whatever input you want to supply can be typed directly into the interpreter; e.g. as seen in the previous examples. To save

[1]Do not mistake this with the value of the expression `(cylinder-volume 10 20)` – the value of this is the number 6280 assuming `pi` is defined as 3.14.

typing you may put commands into a file and just load the file, using the `load` command. For example, `(load "mycommands.scm")` will load in the file `mycomands.scm` from the current directory. You may omit the suffix `.scm` and write `(load "mycommands")` and scheme will supply it for you.

Output commands are also not commonly needed, because the interpreter will automatically print out the *value* of the command you executed, if any. If you need to do any printing in addition, the `print` is useful. This is really not a part of Scheme, but is defined in the file " ranade/public/graphics.scm" which you must first load. After loading you can use it to print any number of arguments with a single command, e.g. `(print x y z)` will print values of `x`, `y` and `z` respectively.

The `print` command does not return a value.

2.3.3 Graphics Commands

Scheme offers users a graphical interface to create and draw pictures. For this you need to execute the following command which loads the necessary file:

```
(load "~ranade/public/graphics.scm")
```

A window containing a screen for graphics should come up after this.

The following commands are available:

1. (gr:reset) : Clears the graphics screen and resets the graphics system.
2. (gr:write x y "string") : Writes the given string at position (x,y)
3. (line $x_1 y_1 x_2 y_2$) : Draws the line between the specified points.
4. (point $x_1 y_1$) : Plots the specified point.
5. (trans dx dy <commands>) : Translates the origin to (dx,dy) and evaluates the commands.
6. (rotate cos(x) sin(x) <commands>) : Rotates axes by an angle x and evaluates the commands.
7. (scale sx sy <commands>) : Multiplies scale of axes by sx, sy respectively.

None of the graphics commands returns a value.

2.3.4 The `if` command

The general form is:

```
(if condition-expression
    then-command
    else-command)
```

The `condition-expression` is first evaluated. If this evaluates to true, then the `then-command` is evaluated and its value (if any) is returned as the result of the `if` statement. Otherwise, the `else-command` is evaluated and its value (if any) is returned as the result. Suppose we wish to set `y` to have the absolute value of `x` then we could write:

```
(define y (if (> x 0)
              x
              (- x))
)
```

This should not be read as "If `x>0` then set `y` to `x`, else set `y` to `-x`". Instead, it should be read as "Set y to the value returned by the `if` statement; the `if` statement evaluates to x if `x>0`, else it evaluates to -x.". The difference between the two should become more obvious if you consider the following statement:

```
(define z (* 2 (if (> x 0)
                   x
                   (- x)))
)
```

You will see that it is not possible to nest `IF` statements of FORTRAN inside arithmetic expressions. Scheme is very flexible in comparison. Other statements e.g. the `do` statement which is analogous to the FORTRAN `DO` statement also is associated with a value, and can hence be nested in contexts where values are expected.

The Scheme `if` can be used in a more conventional manner as well. For example, the following will produce a horizontal line or a vertical line depending upon whether `a` is 0 or not.

```
(if (= a 0)
    (line 0 0 0 1)        ;; vertical line
    (line 0 0 1 0)        ;; horizontal line
    )
```

Note that the `line` commands do not return a value, so the enclosing `if` will not return a value either. So it doesnt make sense to use such `if` statements in places where a value is expected.

2.3.5 Examples

Here is a procedure for drawing a square:

```
(define (square)
   (line 0 0 0 1)
   (line 0 1 1 1)
   (line 1 1 1 0)
   (line 1 0 0 0)
)
```

This can be invoked using `(square)`. It will draw a unit square in the positive quadrant.

This `define` statement used above deserves a comment. First of all, the body of the define contains 4 commands (for drawing the 4 lines), unlike the single command in the procedure `cylinder-volume`. Scheme in fact allows the body to consist of many commands. If the body of a procedure definition consists of many commands, then a natural question is, what is the value returned when the procedure is called? The answer is: the value of the last command (if any).

For the procedure `square`, thus the value of `(line 1 0 0 0)` would be returned. Since `(line 1 0 0 0)` has no value, no value would be returned for square either.

2.3.6 Nesting of commands:

In the definition of `cylinder-volume` we saw that the `*` command was used inside the `define command` – this is called *nesting* the former inside the latter. Such nesting is not only allowed but inevitable. Almost anything can be nested inside anything else, for example:

```
(define p (if (= a 1)
              100
              (if (= a 2)
                  150
                  (+ b c))))
```

This shows one `if` command nested inside another. The meaning of the definition should be clear: `p` is set to 100 if `a` is 1; to 150 if `a` is 2, and to the sum of `b` and `c` if `a` is neither 1 nor 2.

Here is a case where nesting is not allowed in Scheme:

```
(define q (define p 100))
```

This is illegal because the `define` special form expects to evaluate its second argument and discover a *value*. But a command such as `(define p 100)` does not have a value and is thus inappropriate in this place.

2.4 General Remarks on the Scheme Interpreter

The Scheme Interpreter takes commands typed by the user, executes them, and prints the *value* returned by the command on the screen.

The simplest kind of Scheme command is a name. The name represents the value associated with it by preceding statements. If you type a name to the intepreter, then it simply returns its value.

The more complex command is an expression, i.e. anything enclosed in parentheses, say `(f a1 ... an)`. In this, `a1` through `an` could themselves be commands (which are said to *nest* inside the outer command). For expressions, the evaluation process is more complicated. It depends upon whether `f` is a procedure (such as `cylinder-volume`), or a special-form such as `define`.

If `f` is a procedure, then Scheme evaluates the commands `a1` through `an` and then `f` is called with these values as arguments. The result of applying `f` to the values of the arguments

is the value of the entire expression `(f a1 ... an)`. Notice that the values evaluated for commands `a1` through `an` are not printed on the screen, but are simply used as arguments to the function `f`. Only the final value of the command typed in by the user to the interpreter is printed on the screen.

If `f` is not a procedure, then it is said to be a *special form*. A special form should be thought of as a *built-in* language construct, e.g. like the `DO` keyword in Fortran. The commands `define` and `if` are examples of special forms. For special forms the arguments `a1` through `an` may or may not be evaluated depending upon the definition of `f`. Furthermore, the entire command `(f a1 ... an)` may or may not be associated with a value. For example, if the command is `(define a1 a2)` then `a1` is not evaluated; in fact it is the purpose of the `define` command to associate a value with `a1`. Further, the expression `(define a1 a2)` itself has no value. Another example is `if` which is also a special form. When the command `(if c e1 e2)` is evaluated, `c` is first evaluated and then only one out of `e1` and `e2` are evaluated depending upon whether `c` evaluates to true or false. In this case, a value is returned; the value of `(if c e1 e2)` is the value of `e2` if `c` evaluated false and the value of `e1` otherwise.[2]

What makes Scheme exciting is that it provides facilities so that new special forms can also be defined by the user. This is like allowing a user to add new kinds of statements to FORTRAN or C. It is this flexibility that makes it easy to support many programming styles within Scheme.

2.5 Exercises

1. Write a procedure that returns computes the square of a number.

2. Write a procedure that returns a 1 if the given number is positive, 0 if the number is 0, and -1 if the number is negative.

3. Write a procedure that returns temperature in Centigrade given it in Fahrenheit.

4. Get practice using emacs by doing the emacs tutorial and running scheme from inside emacs. Emacs has a *scheme mode* which eases the task of editing scheme files. This mode is automatically invoked when you edit a file which has a ".scm" extension. Inside this mode, the TAB key will take you to the correct indentation. Further the command ctrl-meta-q when typed at the beginning of a function definition will indent the function properly. It is extremely important to be conversant with these indenting features. Your programs will be expected to be indented well.

   Get familiar with the commands "ctrl-c ctrl-e" with allow text from the file you are editing in emacs to be submitted to the scheme buffer in emacs. Also the other commands in the scheme mode.

5. You are to find the errors in some Scheme commands given below.

   (a) `((+ 100 200))`

[2]This assumes that `e1` or `e2` respectively do have a value. If not then no value is returned by `(if c e1 e2)` as well.

(b) `(define (pi) 3.1415)`

(c) `(define pi (3.1415))`

(d) `(define (cube n) n*n*n)`

(e) `(define (cube n) n * n * n)`

(f) `(define (cube1 n) (print (* n n n)))`

Hint: Note that In many languages such as FORTRAN brackets are mostly harmless, i.e. it doesnt hurt to put an expression in an additional pair of brackets. This is however not true in Scheme. As mentioned before Scheme thinks of "(" as the beginning of a command, and ")" as the end of one. Likewise if the first argument after a `define` is in parentheses then Scheme interpretes that as a definition of a function. Finally, note that the command `print` does not return a value. Do you understand the implications of this on the definition of `cube1` above? Do note that all of the above commands may not produce errors, however, they probably do not do what you expect.

6. Write a program that draws a square given the center of the square and half the side length. It should return the area of the square as the result, and also print the coordinates of the corners on the square. Thus `(square 5 4 10)` should draw a square of side 20 with center at (5,4) print a message on the screen saying the corners are (-15,-16), (25,-16), (25,26), (-15,26), and return the value 400. Note that `print` can take as argument strings as well, e.g. `(print "Corners:  " ...)`.

Chapter 3

Functional Programming

In some ways Functional Programming is the simplest paradigm of the ones we are going to study. One may consider it to be *minimalist*;[1] indeed, it is possible to write interesting functional programs using only two kinds of language statements, e.g. `define` and `if`. This simplicity is very useful, it often leads to programs that are easy to reason about.

In the first part of our study of functional programming, we will define it by what it *lacks* rather than what it posseses. Functional programming does not have an assignment statement using which the value of a variable can be changed. It does have variables – but the values, once assigned, cannot be changed. Indeed, the term variable is misleading in functional programming; it is much more appropriate to think in terms of *names* that are given to values that arise during the course of program execution.

In a language such as FORTRAN or C, most of program execution happens using loops, of which there are various varieties. In Functional Programming, we use *recursion* to do all the tasks which are done using loops. This might seem complicated, however, you will soon see that often this makes for shorter and more understandable programs. Recursion will be an important topic in our study. It should be mentioned that the importance of recursion goes beyond functional programming; recursion is a very important tool in design of algorithms. We will also see use of recursion in graphics, this will form an instructive and pleasant diversion.

In the second part of our study we will take up the question of *reasoning about programs*. In fact we will present a strategy for proving the correctness of functional programs as well as imperative programs. By applying the strategy to programs in the two paradigms solving the same problem, we will hope to pursuade you that functional programs can indeed be much easier to reason about.

In the third part we will consider the notion of higher level functions. A higher level function is simply a function that operates on functions; in programming terms, a high level function is one that takes another function as argument, or returns a function as result. Many conventional languages such as FORTRAN and C support some of these features; however in functional programming it is especially convenient to program using high level functions.

Our study will not include an important feature of functional programming: *lazy evalu-*

[1] Minimalism is a paradigm in Art that stresses simplicity, use of the fewest and barest essentials or elements be it in painting, music, literature, or design.

ation. This is a very useful feature, which among other advantages, makes it very easy to write programs dealing (implicitly) with infinite structures. Lazy evaluation is not supported in common programming languages such as FORTRAN, C, C++, Java.

In addition, we will also study the *list* data structure, and consider several problems in *symbolic computing*. It will be seen that all this is programmed elegantly in the functional style.

Although our examples will use the Scheme language, most of the ideas we develop will be useful while programming in other languages as well. For example, it should become clear that in the interest of understandablility, it is better to write programs in which the value of a variable does not change, especially that of a global variable. A clear understanding of recursion is of course, central to all of Computer Science. Finally, the use of high level functions is an important abstraction which can be, and is often, used in standard programming languages.

Chapter 4

Recursion

A fundamental idea in design of programs is *problem reduction*. We say that a problem A can be reduced to a problem B, if given the solution to problem B we can (easily) extract (from the solution to problem B) the solution to problem A. This notion is very common in Mathematics, where we might say "using the substitution $y = x^2 + x$ the quartic equation $(x^2+x+5)(x^2+x+9)+7 = 0$ reduces to the quadratic $(y+5)(y+9)+7 = 0$". Of course, a reduction is useful only if the new problem is in some sense easier to solve than the original problem. An interesting case of reduction is the one in which the new problem is of the *same kind* as the original problem. Such a problem reduction procedure is said to be *recursive*.

Recursively reducing a problem may not seem to be very useful. However, not only is it very useful, it is actually very common too. Consider the following rules for differentiation:

$$\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$$

$$\frac{d}{dx}(uv) = v\frac{d}{dx}u + u\frac{d}{dx}v$$

The first rule, for example, states that the problem of differentiating the expression $u + v$ is the same as that of first differentiating u and v separately, and taking their sum. You have probably used these rules without realizing that they are recursive.

Of course, a recursive procedure is not of much use if it only specifies how to reduce problems. Eventually, we must get to problems that can be solved directly, without applying any additional reduction. These problems that are expected to be solved directly are said to be the *base cases* of the recursive procedure. Suppose we wish to compute:

$$\frac{d}{dx}(x\sin x + x)$$

Then using the first rule we would ask to compute

$$\frac{d}{dx}x\sin x + \frac{d}{dx}x$$

Now, the computation of $\frac{d}{dx}x$ is not done by further reduction, i.e. this is a base case for the procedure. So in this case we directly write that $\frac{d}{dx}x = 1$. To compute $\frac{d}{dx}x\sin x$ we could use the product rule given above, and we would need to know the base case $\frac{d}{dx}\sin x = \cos x$.

4.1 How Recursive Programs Execute

We start with the simplest: computation of $i!$ given i. This uses the identity $i! = i \cdot (i-1)!$ which is true if $i > 1$. We also know that $1! = 1$. This is easily coded:

```
(define (factorial i)              ;; i assumed a positive integer
   (if (> i 1)
       (* i (factorial (- i 1)))
       1)
)
```

We will discuss this program at some length to provide to you a rough model for how recursive programs are executed. In some ways, it is fair to say that a computer does not make a distinction between recursive and non recursive programs.

The general mechanism for executing procedures is: suspend whatever program you are executing currently, start up the procedure, wait for it to complete, and then proceed with your program, possibly using the results returned by the procedure. The important ideas are: the program and the procedure interact only twice: at the beginning when arguments are sent to the procedure, and finally, when the results return from the procedure. Indeed, somewhat fancifully, we could think of the main program as *contracting out* the task for executing the procedure to an *external contractor*. The external contractor works independently except for communicating with the main program at the beginning and at the end. This means, for example, that both the main program, and the procedure may have a variable with the same name; the contractor does the work in his own space and uses his own version of `i` which does not interfere with the version of `i` that the main program has.

In a recursive procedure, we may consider that our contractor further *sub-contracts* the work to another contractor, who in turn may contract it out to another contractor and so on. Suppose you have the call `(factorial 3)` in the middle of some code, say some function `f`. Then the first contractor starts working on `(factorial 3)` only to discover that it needs to compute `(factorial 2)` which needs to be subcontracted out. When this subcontractor executes the code of `factorial` with the value of its variable `i` set to 2, it discovers that it needs to have the value of `(factorial 1)`, which is subcontracted out. This new contractor now starts executing its code, but it realizes that it does not need to do any subcontracting: its for its variable `i` the contition `(> i 1)` is false, and it can return 1 directly. Notice that at this point, we have the original program waiting, also waiting are the contractors working on `(factorial 3)` and `(factorial 2)`. The contractor working on `(factorial 1)` returns 1 to the contractor from which it got its work, i.e. the contractor for `(factorial 2)`. When this contractor gets the value, it resumes its work. So it calculates `(* i received-value)` which it returns to the previous contractor; until eventually the main program receives the value 6.

The above schematic model should enable you to see how the base cases in the recursion get used. Whenever you make a recursive call, you are in fact setting up a long series of subcontractors – for this to terminate you need to have a base case.

4.2 Recursion vs. loops

Recursion can be used to effectively accomplish the same thing as loops. Suppose you wish to print the squares of the numbers 1 through n on the screen. Here is how you could do it:

```
(define (print-square n)
        (if (> n 0)
            (begin (pp (* n n))
                   (print-square (- n 1)))))
```

This program uses the command `pp` which stands for "pretty-print". In general, executing `(pp x)` will cause `x` to be printed in as nice a form as the Scheme Interpreter knows.[1]

In this procedure we have used the `if` statement without specifying the else-part. In such cases, if the condition happens to be false, the value returned will be undefined.

4.3 Recursion in Graphics

Recursion is not limited only to computing; we can use it for drawing pictures as well.

Suppose we wish to draw a tree. Fancifully, we may think of a tree as made up of two branches, each with a small tree on top. Thus we might write the code for drawing trees as follows:

```
(define (tree n)
        (if (> n 0)
            (begin (line 0 0  0.5 0.5)
                   (line 0 0 -0.5 0.5)
                   (trans  0.5 0.5 (scale 0.5 0.5 (tree (- n 1))))
                   (trans -0.5 0.5 (scale 0.5 0.5 (tree (- n 1))))
)))
```

The *(begin a1 ... an)* statement is used for grouping statements together. It simply causes the commands `a1` through `an` to be evaluated. The value of `an` is returned as the value of the `begin` statement. In the present case, note that the last command inside the `begin` command shown, `(trans -0.5 ...)` does not have a value. Hence the `begin` statement itself will not have a value. As a result, the `if` also will not return a value.

The basic idea used here is that of *structural recursion*. A tree in nature has a structure whereby its parts are self-similar. This idea is very important, and will be used later quite extensively.

4.4 Clever Recursion

We will now give an examples of recursion as an algorithm design technique. This will be different from the preceding examples in which the algorithm to solve the problem is

[1] For now, you may wonder whether there can be different ways of printing numbers, but later when `x` might be a more complex Scheme structue, this prettiness will be seen to be extremely useful.

obvious (because the statement is itself recursive) and we were only attempting to express it recursively. In the examples of this section, the algorithm to solve the problem is not obvious by any means, and you will see that recursive thinking helps us discover the algorithm. The material of this section is not the focus of the course– you will learn more about it in courses on Algorithm Design. However, the point here is to realize that recursion is a very powerful idea.

The basic idea in most cases is: for some values of the input, the solution is obvious. These values will typically be the base cases. For other (larger) values, say x, we *imagine* that we already know how to obtain the solution for values smaller than x. Can we then use the solution of the smaller values to obtain the solution for the larger values? If we can answer this question positively, we have a recursive algorithm.

4.4.1 Computation of the sine of an angle

Given x, we are to compute $\sin(x)$. Of course, this can only be done approximately in general.

We might reason as follows. For small x, $\sin(x) \approx x$. This then will be our base case. For large x, we simply need a way to construct $\sin(x)$ from $\sin(u)$ where $u < x$. For this we use the identity:

$$\sin(x) = 3\sin(\frac{x}{3}) - 4\sin^3(\frac{x}{3})$$

This gives the following code.

```
(define (polynomial t)
        (- (* 3 t) (* 4 t t t) ) )

(define (sin x)
        (if (< (abs x) 0.001)
            x
            (polynomial (sin (/ x 3) ) ) ) )  }
```

4.4.2 Square root computation

Given x we are to return $\sqrt{x}$. This uses a different strategy than the one given above. The idea here, which is useful for other computations as well, is to start with an approximate square root, say y_0. We then device a strategy which refines our approximation. So given an approximation y_i we use our strategy to get y_{i+1}. This we apply repetitively, till the error becomes small enough.

So the following fact is needed:

Theorem 1 *Let* $y_{i+1} = \frac{1}{2}\left(y_i + \frac{x}{y_i}\right)$. *Then* $y_{i+1}^2 - x \leq \frac{(y_i^2 - x)^2}{4y_i^2}$.

The proof is left as an exercise. Now at least for the case $y_i^2 \geq x \geq 1$ it should be clear that the the program is immediate.

```
(define (internalsqrt x y)
        (if (> 0.001 (abs (- (* y y ) x) ) )
```

```
            y
            (internalsqrt x (/ (+ y (/ x y)) 2) ) ))

(define (sqrt x)
        (internalsqrt x 1.0) )
```

4.4.3 Coin Change Problem

Given an integer M determine the minimum number of coins needed to make up M paise, assuming you have an infinite supply of coins of all the denominations 1, 2, 5, 10, 20, 25 and 50 (i.e. those available for Indian currency). At first glance you may think that the solution is obvious: keep using the largest possible coin until the amount drops down to 0. For $M = 40$ this idea would suggest coins of 25, 10 and 5; the best solution uses just 2 coins of 20.

How then, can we find the minimum number? The recursive idea is to start by assuming that we can solve the problem for all integers between 1 and $M - 1$. By itself, this is not enough; you do need some more exploration of the problem. So let us take an example, to make our thoughts concrete: suppose that you wish to find the change for Rs. 5.37. Let us try to visualize the problem– maybe this will help us find the solution. Imagine therefore, that there is a bag in front of you which contains the requisite coins. Of course, you dont know how many coins are there, nor what their denominations are. There is a policeman sitting near the bag who will not allow you to look inside. Your goal is to guess without looking in how many coins are there. Furthermore, there is a friendly angel sitting next to the policeman. This angel will tell you how many coins are needed for any amount x, provided x is smaller than Rs. 5.37.[2]

Imagine now that you think for a long time, and get nowhere. Then maybe the policeman takes pity on you, and agrees to show you one coin from the bag. So he opens it, and out comes a 50 paise coin. Is this information of any help? Surprisingly, this information is just enough to solve the problem, given our old assumption. If you remove the 50 paise coin, then the bag will contain coins worth Rs. 4.87. Now you ask our angel, "How many coins do I need to make Rs. 4.87". The angel replies "12". Now you can confidently tell the policeman that his bag must have contained 13 coins.

Why should you be sure of your answer? Because the angel said that 12 coins suffice to make Rs. 4.87, you know that 13 coins are sufficient to make Rs. 5.37. So you know that the bag must contain at most 13 coins. But could it contain fewer, say 12? But if so, then after removal of the 50 paise coin, it would have Rs. 4.87 in 11 coins only. But this contradicts our angel who said that the minimum number of coins needed to make Rs. 4.87 is 12. Since our angel never lies, the bag could not have had fewer than 13 coins in the beginning.

So what we have proved can be summarized as "If the bag contains a 50 paise coin, then 13 coins are needed". But now you should realize that there is *nothing* special about the coin being 50 paise. Had the policeman shown you a 25 paise coin, you would ask the angel the number of coins needed to make Rs. 5.12, and so on. We can summarize this as in Figure 4.1.

[2]Note that when we come to writing a program for this, we well simply recurse, whenever we want information from the angel. The angel *is* recursion.

Coin Shown	Question to angel	Angel's Response	Your answer
50	"How many coins for Rs. 4.87?"	"12"	13
25	"How many coins for Rs. 5.12?"	"12"	13
20	"How many coins for Rs. 5.17?"	"13"	14
10	"How many coins for Rs. 5.27?"	"12"	13
5	"How many coins for Rs. 5.32?"	"13"	14
2	"How many coins for Rs. 5.35?"	"12"	13
1	"How many coins for Rs. 5.36?"	"13"	14

Figure 4.1: Possible Scenarios

Now note that one of the answers in the last column must be correct. This is because you have considered all possible coins that the policeman could show you. You further know that it is indeed possible to make up Rs. 5.37 using coins equal in number to any entry in the last column. Hence, you simply take the minimum entry, and declare that as your answer, i.e. 13.

Now note the important point: the policeman does not really need to tell us anything!! We considered his all possible actions anyway.

The Scheme program that implements the above ideas is left as an exercise.

4.5 Efficiency and Recursion

Sometimes it might be said that a recursive program is inefficient, as compared to one which uses loops. For example, the factorial program given earlier could be considered inefficient because it needs to create several copies of the same variable, viz. variable `i`, as discussed in Section 4.1. A program which simply uses a loop and multiplies numbers 1 through n to compute $n!$ is very likely more efficient. However, do note that there are many problems, including some that we have discussed above, for which there are no obvious non-recursive implementations. Hence, it should be clear that recursion is an extremely valuable programming and algorithm design technique. Since our emphasis is on understanding this technique, we will ignore some level of inefficiency.

However, occasionally some *natural* recursive algorithms are far too inefficient. In that case, the right approach is to use a recursive algorithm/program as a starting point, which is subsequently refined to get a fast algorithm. Here is a simple example.

The ith Fibonacci number is defined to be the sum of the $i-1$th and the $i-2$th Fibonacci numbers, with the first and second Fibonacci numbers being 1. More specifically, if F_i denotes the ith number, we have, $F_i = F_{i-1} + F_{i-2}$ and $F_1 = F_2 = 1$. This directly translates to a recursive program for computing F_i.

```
(define (fib i)
        (if (<= i 2)
            1
            (+ (fib (- i 1) ) (fib (- i 2 )))
        )
```

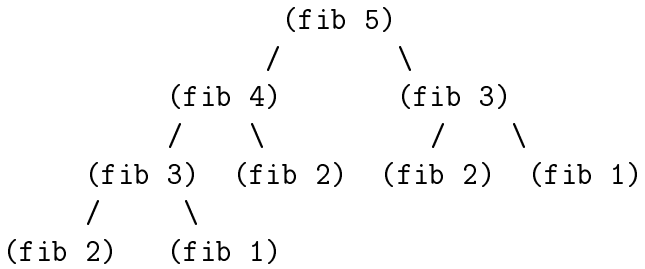

Figure 4.2: Call Tree for `(fib 5)`

)

Trees (drawn growing downwards) are often used to visualize execution of recursive programs, like `fib`. For example, the recursive call `(fib 5)` will generate a call to `(fib 4)` and to `(fib 3)` each of which will generate more calls. The execution of this program for the call (fib 5) can be represented by a tree as shown in Figure 4.2. The figure shows all the calls made during the execution of `(fib 5)`. Observe that this program is not very efficient; it calls `(fib 3)` 2 times and thus repeats work. In fact, if you draw out the tree for `(fib 6)` you will see a lot more work repeated (many computations, many times). By the way, this kind of inefficiency is present also in the coin change problem solution we gave earlier.

We will see later how such programs can be made more efficient. For now, our focus is on the clarity of the programs that we have developed. It is hoped that you will agree that all the programs appear very simple, and are almost direct representation of the mathematical definition of the problem.

4.6 *Tracing* Scheme procedures

Scheme provides a `trace` facility which is very convenient for debugging recursive programs.

```
(trace procedure)
```

This asks scheme to display a message everytime `procedure` is called, showing its arguments, and also when the function returns, along with the value returned. During debugging, it is expected that this command will be issued after `procedure` is defined but before it is invoked. At any point in time, as many functions as you want can be traced.

For example, before invoking `(sin 0.15)` but after `sin` has been defined, if you type `(trace sin)`, Scheme will show you what happens at each recursive call to `sin`.

The command:

```
(untrace procedure)
```

asks scheme to stop displaying messages i.e. it undoes the effect of `(trace procedure)`.

4.7 Exercises

In the problems below that ask you to write functions, your solution may include additional functions that might get called internally.

1. What will the value of `a` be after the following code is executed?

```
(define (print-square n)
        (if (> n 0)
            (block (pp (* n n))
                   (print-square (- n 1))
                   n)
))

(define a (print-square 10))
```

2. Write a function to draw a regular polygon of unit side-length with center at the origin. It should take the number of sides as the argument.

3. The call `(random x)` will return a random integer in the range 0 through x-1, if x is an integer, or from the interval 0 to x if x is a real number. Use randomness to determine the number of branches, their length, or their angle (this can be changed by scaling the x and y coordinates separately) while drawing trees. Try to make your trees resemble some known tree!

4. Write a function which computes the value of e using n terms of the following series, with n being taken as an input:

$$e = 1 + \frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \ldots$$

5. Write a function to compute the minimum number of coins to make up an amount M. Assume Indian currency coins, i.e. denominations 1,2,5,10,20,25,50.

   (Your program might take a very long time on even slightly large M. Later on we will see how to fix this.)

6. Write a function that takes an integer number as argument and determines whether the number is prime.

   Use a simple idea of checking whether i is a factor, for $2 \leq i < n$ where n is the input number. Note that the Scheme function `(remainder m n)` returns the remainder when `m` is divided by `n`.

7. Modify the `fib` function so that it not only returns the nth Fibonacci number, but also prints a tree on the screen showing all the calls made during execution, i.e. similar to Figure 4.2, but for all n. It might be useful to know that `(number->string n)` returns a string representation of the integer n.

Chapter 5

Proving Program Correctness

Programming errors, or bugs, can cause and already have caused many, many disasters, such as failures in satellite launches, and car accidents (bugs in the program that performs car control functions). We are all familiar with operating system or other program releases that "fix bugs", and which keep appearing from time to time. If we could some how ensure that a program will function correctly, before it is released for use, we will avoid much expense, much wasted effort, and many accidents.

How can we be sure that a program is correct? One way is to test it on several examples: this is obviously insufficient; however much testing we do we still cannot be sure that the program will not fail on some instance that we have not tested. A better way is to *prove* the program to be correct. A proof of the correctness of a program is in spirit similar to the proof of a theorem in Euclidean Geometry. In both cases, the proofs consist of a sequence of assertions, where the assertions must either be postulates, or be implied by preceding assertions using the so called *inference rules*. For example, if you know that "A implies B" and that "A is true", then you can conclude that "B is true". By properly employing such rules of inference if you prove that "The bisectors of the angles of a triangle meet in a point", then most people accept this as true. Proofs of correctness of programs are expected to have similar credibility.

The proof process is substantially different for imperative and functional programming styles. It will be seen in many examples that the proof of correctness is much easier to write for functional programs than imperative style programs. The rest of the chapter describes the proof process with examples.

5.1 Basic ideas in program proving

The first step in proving the correctness of a program is to formulate the problem precisely. We need to precisely characterize the input to the program, and the output required from the program. For example, for a program computing the greatest common divisor, the input characterization would say that the inputs must be positive integers (otherwise the gcd is undefined), and the output is the largest integer that divides them both without leaving a remainder. Another example is a sorting program. For this the input could be an array A of numbers, and the output would be another array B of the same length containing a permutation of the input such that B[i]$\leq$B[j] if $i < j$.

Given that the input to the program satisfies the input characterization (also called *preconditions*) the proof must establish (explicitly or implicitly) the following progressively harder claims:

1. The program terminates, e.g. does not go into any infinite loop.
2. The program terminates but not because of an erroneous operation, e.g. division by zero, or accessing an array using a subscript that is larger than the length of an array.
3. The value returned by the program is the correct value.

5.1.1 Form of the proof

In general, a proof consists of a sequence of statements, with a *justification* for each statement stating how that statement is deduced. Proofs of programs are similar. The statements in such a proof must however characterize the behaviour of the program. This characterization will be of the form:

When control arrives at point P in the program, the variable V has value v.

If there are no loops or function calls from the beginning of the program to point P, then we can reason about the effect of every statement from the beginning to point P and derive a characterization of the value of V in terms of the values of the program variables at the beginning. We will call this *hand execution*, or *symbolic execution*. Note however, that such a justification is reliable if we hand execute only over a small number of statements. For large number of statements, we are likely to make mistakes as we trace the execution.

Thus it would seem at first glance that the idea of hand execution would be useless when there are loops or function calls between the beginning of the program and the point P, or when P is itself inside a loop. However, as will be seen below, with some additional machinery (induction) we will be able to use hand execution in our proofs.

5.2 Proving correctness of functional programs

We begin with an example. Consider the program for evaluating the factorial of number.

```
(define (fac n)
   (if (= n 0)
       1
       (* n (fac (- n 1)))
   )
)
```

Program Specification: The function correctly returns n! for all non-negative integers n.

Proof of correctness: The proof is by induction. The induction hypothesis is:

IH(n) : The call (fac n) returns n! provided n is a non-negative integer.

The base case is n=0, i.e. we are required to prove : IH(0) is true. We do this by hand-executing the program. When n has the value 0 then the conditional test (= n 0) succeeds, and the procedure returns 1, which in fact is the correct value 0!

Next we establish the induction. We will assume IH(n-1), i.e. that the program correctly computes n-1! if n-1 is supplied as the argument. We are then required to prove IH(n), with $n > 0$. Again we employ hand-execution. Now in the execution the first check (= n 0) fails, since we know $n > 0$. Thus the program now computes (fac (- n 1)) and returns the result after multiplying it by n. Now, (- n 1) is just n-1, and this must be at least 0 since $n > 0$. But IH(n-1) guarantees that factorial of n-1 is correctly computed if n-1 is a nonnegative integer. Thus the call (fac (- n 1)) returns n-1! – but then the value returned is n * n-1! which is n! by definition. Thus the correct value is returned, establishing the induction.

It should be noted that in all cases, we hand executed the program only for a small number of steps.

5.3 Summary of the argument

Proving the correctness of functional programs consists of the following:

Write down the specifications. These consists of *preconditions* which the inputs to the program must satisfy, and a *claim* regarding the output that will be produced. In the factorial example, the precondition was that the input n being supplied was a nonnegative integer. The claim then was that the value of n! would be correctly returned. Notice that the specifications may leave unsaid what happens if the program is executed without the input specifying the preconditions.

Use induction to prove the specifications. For this, it is necessary to precisely write down an induction hypothesis (this is usually nothing else but the specifications themselves) and identifying a variable or variables for the induction. This variable *over* which the induction is supposed to be is usually the something representative of the size of the problem being solved. In the case of factorial, the variable was the input n.

1. Identify and prove the base case. The base case is often the smallest problem size possible. In case of factorial, we used the base case n=0. The base case is established by "hand executing" the program with the input fixed to the base case.

2. Establish the induction. For this we assume that the induction hypothesis holds for a certain problem size n and prove it holds for the next larger problem size n+1. We may in fact assume that the induction hypothesis actually holds for all problem sizes no larger than n, and then prove it holds for problem size n+1. The induction is established by hand execution as well.

5.3.1 Euclid's Algorithm

Theorem 2 *If $u, v > 0$ and u, v are integers then if v divides u then* $\gcd(u, v) = v$, *else* $\gcd(u, v) = \gcd(v, u \bmod v)$

Our program is based on the above theorem and is as follows:

```
(define (gcd u v)
    (if (= (remainder u v) 0)
        v
        (gcd v (remainder u v))
    )
)
```

We will use gcd(u, v) to denote the gcd of u and v, and (gcd u v) the value returned by program.

Specification: Precondition: inputs u,v must be positive integers. Claim: (gcd u v) = gcd(u,v).

Inductive Hypothesis: The input size can be thought of as the magnitudes of the input numbers. The program can be proved correct by considering an induction over the values taken by either u or v; it is more convenient to use the values taken by v, and so we use that here. The inductive hypothesis is:

IH(n) : (gcd u v) = gcd(u, v) for u having any positive integer as value and v having the value n.

Base Case: n = 1. Thus v takes the value 1. By hand execution of the program, (gcd u 1) = 1 = gcd(u,1) for all u.

Induction: We will assume IH(1),IH(2),...,IH(n-1) and prove IH(n). So consider an execution in which v takes the value n. Let x = (remainder u n). Clearly $x < n$. There are two cases to consider.

CASE 1: x = 0. In this case the function returns n, which is in fact gcd(u, n) for any u.

CASE 2: x > 0. Then the function calls (gcd n x). Since $x > 0$, the precondition of IH(x) is satisfied. Since $x < n$, IH(x) is a part of what we have assumed. Thus we know that this call will return gcd(n,x) = gcd(n, (remainder u n)). But by the theorem above we know gcd(n, (remainder u n)) = gcd(u,n). But this is in fact what we want to return. Hence proved.

5.4 Proof Strategy for Imperative programs

We again begin with the example of computing factorials. Here is a simple program

```
1.      integer function fac(n)

2.      fac = 1
3.      do i=1,n
4.         fac = fac*i
5.      end do

6.      return fac
```

The specification for this program is the same as that for the functional version. The structure of the proof is very different, however. For imperative programs, the proof proceeds by characterizing the values taken by the program variables as the program executes. If the program does not have loops, then this is very easy. In case of programs with loops, we need to characterize what happens to the variables in each iteration. For this we need the notion of a *loop invariant*. This is simply an assertion that holds for every iteration of the loop. The assertion is proved by induction over the number of iterations of the loop. An example is given below. Let x_k denote the value of any variable x after statement 4 has been executed in the kth iteration.

Loop invariant: $fac_k = k!$, $i_k = k$

We could have written that fac=i! instead of the bringing in k; however we do this in order to keep a distinction between variables and the values they take.

Proof of invariant: We use induction over k. The base case is $k = 1$, i.e. the first time statement 4 is encountered. At the beginning of the first iteration the variable fac has the value 1, since that is what it was initialized to in statement 2. In the first iteration the variable i has the value 1. Thus $i_1 = 1$, which is part of what is claimed in the invariant. Further, statement 4 multiplies the current value of i (which is 1) and the current value of fac (which is 1) and sets fac to that. Thus $fac_1 = 1 \times 1 = 1 = 1!$ establishing the base case.

The induction step is as follows. We will assume that $fac_{k-1} = k - 1!, i_{k-1} = k - 1$ and then prove that $fac_k = k!, i_k = k$. Clearly, in the kth iteration $i_k = k$. At the beginning of the kth iteration, fac must have the same value as it did at the end of the $k-1$th iteration, i.e. $fac_{k-1} = k - 1!$. But in statement 4, we set fac to be its current value (i.e. k-1!) times the value of i (i.e. k). Thus after statement 4 of kth iteration, fac must have the value $k - 1! \times k = k!$. Thus $fac_k = k!$, completing the proof.

Proof of correctness: Given the invariant proved above, the function is easily proved to be correct. First, we should observe that the function must terminate. This is because the loop will execute only n times if n is a positive integer, or once if n is zero.[1] Note that we dont need to consider the case of n having a negative value because of the precondition in the specification. Note that since we do not have any operations such as division that can cause erroneous termination, so we can be assured that there are no runtime errors. In summary we know that after a finite number of iterations the program will terminate. The value returned by the program is the value of the variable fac. There are 2 cases depending upon the value of n:

$n = 0$ Loop is executed once, the value of fac at the end is 1, which is returned. This is correct since 0!=1.

$n > 0$ In this case the loop is executed n times, and the value of the fac at the end of the n th iteration, i.e. fac_n is returned. But we know from the invariant that $fac_n = n!$. Thus the current value is returned in this case too.

[1] Here we are not making the distinction between the variable n and its value, because n doesnt change during the program.

5.4.1 Structure of proofs of Imperative Programs:

Let us assume for simplicity that our programs have only a single loop. In this case, the general strategy is to write down a loop invariant that characterizes how the values evolve as the loop executes. The loop invariant is proved by induction over the number of iterations of the loop. To establish the inductive step, we assume that the invariant holds during some $k-1$th iteration, then hand execute the program and keep track of what happens to the variables and show that the values taken by the variables will be such that the invariant is satisfied even in the kth iteration. The base case is established by hand execution from the beginning of the program to the first iteration of the loop.[2]

Proving the invariant is the key step. After this, we need to establish first that the loop executes only a finite number of steps, and there are no errors during execution. In the above program this was easy since the loop executed n times by the definition of the DO statement in Fortran. In other cases, more ingenuinity might be necessary. Given that the program does not go into an "infinite loop" and there are no errors we need to characterize the value that will be returned. This can be done using the invariant for the last loop iteration.

5.4.2 Euclid's algorithm for computing GCD

An imperative style program for computing the gcd is as follows.

```
1.      integer function fgcd(u, v)
2.      integer u,v,temp

3.      while( mod(u,v) != 0)
4.      do
5.         temp = u
6.         u = v
7.         v = mod(temp,v)
8.      end do

9.      return v
```

We will use the expression gcd(u,v) to denote the greatest common divisor of u and v; in contrast fgcd(u,v) is the value (if any) returned by the function above.

The specification is the same as that for the functional program. We will also need the theorem stated there.

Invariant: Let u_k, v_k denote the values of the variables u, v when control reaches statement 3 for the k th time. Then the claims are:

1. $u_k, v_k > 0$
2. $\gcd(u_k, v_k) = \gcd(u_1, v_1)$ i.e. the gcd of the values in u and v is the same as that when the program started execution.

[2] In every case, we hand execute only for 1 iteration, i.e. a small number of steps.

Proof of the invariant: The proof is by induction over k. The base case is $k = 1$. When statement 3 is visited for $k = 1$st time, the values of u,v are u_1, v_1. But because of the precondition we know that $u_1, v_1 > 0$. Since $k = 1$ clearly $\gcd(u_k, v_k) = \gcd(u_1, v_1)$. Thus the base case is established.

Assume that the invariant holds at the time of the $k-1$th visit to statement 3, i.e. $u_{k-1}, v_{k-1} > 0, \gcd(u_{k-1}, v_{k-1}) = \gcd(u_1, v_1)$. Now do a hand execution of the $k-1$th iteration. Since we know that statement 3 is reached for the kth time, the test in statement 3 must have succeeded, i.e. it must be that $u_{k-1} \bmod v_{k-1} > 0$. Continuing the execution we see that when statement 5 is executed, temp gets the value u_{k-1}. When statement 6 is executed variable u gets the value v_{k-1}. Finally in statement 7 the variable v gets the value $u_{k-1} \bmod v_{k-1}$. For this operation to be legal we need that $v_{k-1} > 0$, but we have that from the inductive hypothesis. After this, the control returns to statement 3 for the kth time. The values of u, v at this time are $v_{k-1}, u_{k-1} \bmod v_{k-1}$, and we have established that both of these are positive earlier. Thus $u_k = v_{k-1} > 0$ and $v_k = u_{k-1} \bmod v_{k-1} > 0$. Finally, using the theorem stated earlier $\gcd(u_{k-1}, v_{k-1}) = \gcd(v_{k-1}, u_{k-1} \bmod v_{k-1}) = \gcd(u_k, v_k)$. But from the inductive hypothesis we know that $\gcd(u_{k-1}, v_{k-1}) = \gcd(u_1, v_1)$. Thus we have $\gcd(u_k, v_k) = \gcd(u_1, v_1)$. This completes the proof of the invariant.

Proof Of Proper Termination: Note that because of the invariant, when control reaches statement 3 the variable v is guaranteed to be positive. Thus the mod operation is legal, and so there are no abnormal terminations. Next note that the value of the variable v strictly decreases in each iteration, and the invariant above ensures that it is always positive. Thus the loop must only execute a finite number of steps.

Proof that correct value is returned: Suppose the control reaches the return statement (and we know this happens because of the proof of correctness) after visiting statement 3 k times. At this visit the test in statement 3 must have failed (otherwise the loop must have executed at least one more time). Thus we know that $u_k \bmod v_k = 0$. Thus $v_k = \gcd(u_k, v_k)$. But by the invariant $\gcd(u_k, v_k) = \gcd(u_1, v_1)$. Thus the value v_k returned is indeed the correct value.

5.5 Proving programs involving several functions

Consider the following alternative code for computing the factorial of a number:

```
(define (fact n)
        (fact2 n 1))

(define (fact2 n f)
        (if (>= n 1)
            f
            (fact2 (- n 1) (* f n))))
```

In this case the function `fact` is very simple; clearly, its correctness depends upon what `fact2` does. We must prove that the factorial of `n` is returned for the call `(fact2 n 1)`. If

we can prove this (see Exercises), then that is adequate. Then we can conclude that `(fact n)` must return `n!` since `(fact n)` returns `(fact2 n 1)` which we proved is `n!`.

In general, once we write down and prove the specifications for a given procedure `f`, we can use these specifications in proofs of other procedures which call `f`.

5.6 Conclusions

It should be obvious that the recursive versions are far easier to prove correct. One of the reasons for this simplicity is that each variable has only one value in a functional program. Thus we dont have to worry about "value of variable at time t", or "how the variable decreases". Second, the functional programs are also more compact and pretty much follow the statement of the mathematical ideas on which they are based; notice that the functional program for gcd is very similar to the statement of Theorem 2. There is a reason for this: induction is a very fundamental mathematical proof techniue, and because recursion is very related to induction, it turns out that recursive programs are usually easier to write and prove correct.

On the debit side, functional (recursive) programs often require more memory and run slightly slower than their imperative counterparts.

In these days when program readability and correctness are very important, it is recommended that functional programming (recursion, no reassignment to values of variables) be used wherever possible.

5.7 Exercises

1. Write a function (digits n) which returns the number of digits in a number n. Write a proof of correctness. Clearly write down the mathematical property on which your program is based.

2. Write the invariant necessary to prove the correctness of the following gcd program.

```
integer function gcd1(u, v)
integer u,v,w;

w=mod(u,v)
while(w > 0)
  u=v
  v=w
  w=mod(u,v)
end do

return v
```

   Prove the program correct.

3. Consider the code given below computing ${}^nC_k$:

```
integer function choose(n,k)
integer n,k,c,i
c=1
do i=1,k
  c=c*(n-i+1)/i
end do
return c
```

Write an invariant for the do loop, i.e. state what you know about the different variables at the end of the jth iteration of the loop.

4. Write down the specifications for the function `fact2` of Section 5.5.

   Verify that the specification implies that `(fact2 n 1)` correctly computes `n!`. Prove that the call `(fact2 n 1)` correctly computes `n!`. (Hint: Although the first call will always have the second argument to be 1, this may not be the case in the subsequent recursive calls. Hence the proof will actually involve proving the entire specification, i.e. for arbitrary value of the second argument as well.)

5. Write a scheme function `(isqrt n)` that returns $\lceil \sqrt{n} \rceil$. Use a simple algorithms such as trying out the numbers $1, 2, \ldots$ as candidate square roots and checking until the right conditions are satisfied. You may use additional functions if necessary. State the mathematical property on which your program is based. Write a proof of correctness (include a proof for the additional functions you used, if any).

Chapter 6

More on Scheme Statements

6.1 The "begin" Statement

If a number of statements have to be grouped together then they are put in a block. This feature is usually used to group statements causing side-effects. A block is constructed using the "begin" statement.

```
Syntax:
(begin
      (stmt 1)
      (stmt 2)
        .
        .
        .
      (stmt n)
      )
Example:
(if (< n 1)
    (begin
      (line 0 0 0.5 0.5)
      (line 0 0 0 0.5)
      (line 0 0 0.5 0)
      )
    )
```

The begin statement returns the value of the last statement in the block.

6.2 The "cond" statement

The if statement is basically a two way branch. To implement multi-way branching one can use nested if statements but Scheme provides a more elegant way of doing the same - the "cond" statement.

```
Syntax:
```

```
(cond (t1 stmt-11 stmt-12 stmt-13 ....)
      (t2 stmt-21 stmt-22 stmt-23 ....)
                .
                .
                .
      (tn stmt-n1 stmt-n2 stmt-n3 ....)
      (else stmt-e1 stmt-e2 stmt-e3 ....)
      )
```

The evaluation proceeds stepwise:- First t1 is evaluated, and if true stmt11....stmt1m are evaluated, and if false , only then t2 is evaluated and so on. The else statement is a catch-all which is true even if t1....tn are false. Note that the else statement must be provided when the cond statement is used to return a value but it may be omitted when side-effects are to be caused.

6.3 The "let" stetement

Consider the sine function:-

```
(define (sine x)
  (if (< x 0.0001)
      x
      (- (* 3 (sine (/ x 3))) (* 4 (sine (/ x 3)) (sine (/ x 3)) (sine (/ x 3))))
      )
  )
```

Notice that not only is the expression clumsy but is also wasteful, since the same statement (sine (/ x 3)) is evaluated 4 times. To deal with such situations Scheme provides the "let" statement.

```
Syntax:
(let ((v1 value exp.)
      (v2 value exp.)
        .
        .
        .
      (vn value exp.)
     )
     (stmt 1)
     (stmt 2)
        .
        .
        .
     (stmt m)
     )
```

The value expressions are evaluated only once and the names v1....vn are associated with the value expressions. Now these names can be used in stmt1....stmtm. For example, the sine program can be modified to

```
(define (sine x)
  (if (< x 0.0001)
      x
      (let ((s (sine (/ x 3))))
           (- (* 3 s) (* 4 s s s))
           )
      )
  )
```

As is usual in Scheme, the value of the last statement evaluated in the block is returned as the value returned by let. A point to be noted is that v1....vn are available only in the let block. They are undefined outside.

6.4 Exercises

1. Write a function that given two numbers x and n computes x^n. This should be done using the following idea:

   (a) n is even: Then $x^n = (x^{n/2})^2$.

   (b) n is odd: Then $x^n = (x^{(n-1)/2})^2$.

   Write the program such that recomputation is avoided by using `let` statements as much as possible.

   How many multiplications does your program do in evaluating 6.43^{32}? In evaluating 3.79^{73}? How many multiplications would the obvious algorithm do in each case?

Chapter 7

Lists

A lists is an ordered collection of objects. You can construct a list by using the function list that takes the objects you want put into the list. For example, (list 1 2 3 4 5) will return the a list of the numbers 1,2,3,4 and 5 in that order. This list can be assigned to a variable, as in:

```
(define a (list 1 2 3 4 5))
```

`list` is a function, and it can be given more complicated arguments. For example, the following code will make a list of the first 3 multiples of pi:

```
(define pi 3.14)
(define b (list pi (* 2 pi) (* 3 pi)))
```

This would cause b to become the list containing 3.14, 6.28 and 9.42 in that order.

The various elements of the list can be accessed using the functions `car` and `cdr` (pronounced "could er"). Car of a list gives the first element of the list, while cdr gives the whole list except the first element.

```
(car a)  => 1
(cdr a)  => (list 2 3 4 5)
```

Scheme allows us to write cadr which means the car of the cdr, and so on. Thus

```
(cadr a )
=>2
(caddr a)
=>3
```

The function cons (short for "construct") takes an element and inserts it at the front of the list, and returns the list thus formed. The function append appends the given lists together. Thus

```
(cons 0 a)    => (list 0 1 2 3 4 5) ;; a is not changed however.
(append a a)  => (list 1 2 3 4 5 1 2 3 4 5)
(define c (append b a))
```

The last command would set c to be the list with the first 3 elements being the multiples of pi, and the remaining 5 being the numbers 1 through 5. In any case, the values of a and b will not change.

`(list? expr)` returns true if expr evaluates to a list, and false otherwise.

`(null? expr)` returns true if expr evaluates to the empty list (written as '()), and true otherwise.

Quote notation

A shortform has been provided for constructing lists out of constants. Thus the definition of a could be written as:

```
(define a '(1 2 3 4 5))
```

This has the same effect as the earlier definition. Scheme recognizes the character (') to mean that the expression following it should not be evaluated, but be treated as a list. It is acceptable to write

```
(define d '(1 (2 3) 4))
```

This is equivalent to saying (define d (list 1 (list 2 3) 4)). When using the quoted form note that the single quote at the beginning is sufficient, while the corresponding expression using the list function needs the function to be specified more often as above.

There is another important difference between using quotes and the function list. For example, if we write

```
(define pi 3.14)
(define e '(pi (* 2 pi) (* 3 pi)))
```

This would not set e to be a list containing 3.14, 6.28 and 9.42; instead e would become a list containing *symbols*; this will be explained in the chapter on symbols.

Note of course that a command such as

```
(define f (1 2 3 4 5))
```

is erroneous – scheme expects the first term after a "(" to be a function, and it will give an error because 1 is not a function.

7.1 Program Example

```
(define (length l)
   (if (null? l)
       0
       (+ 1 (length (cdr l)))))
```

The above program gives the length of the list l.

The program below takes two lists of numbers as the input and returns a list whose ith element is the sum of the ith elements of the input lists. It is necessary that the input lists be of the same length.

```
(define (sum  a b)
   (if (null? a )
       '()                    ;; the empty list.
       (cons (+ (car a) (car b))
             (sum (cdr a) (cdr b)))))
```

It will be seen that both the programs are based on the idea that a list is *recursively structured*. Thus a list may be thought of as the first element followed by a list of the remaining elements. Thus procedures operating on lists are often recursive; they explicitly operate on the first element and recurse on the remaining elements (the `cdr` of the list). The base case for the recursion is the empty list.

7.2 Proving correctness of programs involving lists

As an example, we will give a proof of correctness of the function `sum`.

Specification: Let `a, b` be two lists of numbers, both having the same length n. Let the elements be denoted by $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. Then the call `(sum a b)` returns a list with elements $a_1 + b_1, \ldots, a_n + b_n$. If $n = 0$ then the result should be the empty list.

Note that the specification includes the preconditions that the lists have the same length and contain only numbers. No claims are made if `sum` is called with unequally long lists.

Inductive Hypothesis IH(n): `(sum a b)` works correctly for lists of length n, assuming the preconditions are satisfied.

Proof of base case IH(0): If the lists are empty, then by hand execution it is clear that the empty list is returned. But this is what is expected, hence proved.

Proof of IH(n+1) assuming IH(n): Consider `a` and `b` of length $n + 1$. Then hand execution shows that the value returned is `(cons (+ (car a) (car b)) (sum (cdr a) (cdr b)))`. Now if `a` is $a_1, \ldots, a_{n+1}$ and similarly `b`, then `(+ (car a) (car b))` is just $a_1 + b_1$. Also `(cdr a)` and `(cdr b)` must have length n since `a,b` had length $n + 1$, and must respectively contain $a_2, \ldots, a_{n+1}$ and $b_2, \ldots, b_{n+1}$. Thus the call `(sum (cdr a) (cdr b))` satisfies IH(n) and the preconditions in the specification, and hence a list containing $a_2 + b_2, \ldots, a_{n+1} + b_{n+1}$ is returned. Thus what is returned is the `cons` of $a_1 + b_1$ with this list, i.e. the list $a_1 + b_1, \ldots, a_{n+1} + b_{n+1}$. But this is precisely what is expected, hence proved.

7.3 More List Operations

With the basic operations car, cdr and cons defined on lists, we can now implement more complex list manipulations. Some of these are predefined in Scheme.

1. Write a function to access the ith element of a list.

   Solution :

```
(define (list-ref list i)
        (if (null? list)
            ()
            (if (= i 0)
                (car list)
                (list-ref (cdr list) (- i 1)))))
```

2. Write a function to return only the even-indexed elements of a list.

   Solution :-

```
(define (even-list list)
        (if (or (null? list) (null? (cdr list)))
            list
            (cons (car list) (even-list (cddr list)))))
```

3. Write a function to return the sum of the elements of a list

   Solution :-

```
(define (list-sum list)
        (if (null? list)
            0
            (+ (car list) (list-sum (cdr list)))))
```

4. Write a function to filter out an element i from a list, if it exists. Here, the objective is to return a list identical to the original in all respects except that all occurences of the element i will be missing.

   Solution :-

```
(define (filter list i)
        (if (= (car list) i)
            (filter (cdr list) i)
            (cons (car list) (filter (cdr list) i))))
```

5. Write a function to merge two lists, given that the two lists are arranged in ascending order.

   Solution :-

```
(define (merge list1 list2)
        (cond ((null? list1) list2)
              ((null? list2) list1)
              ((> (car list1) (car list2))
               (cons (car list1) (merge (cdr list1) list2)))
              (else
               (cons (car list2) (merge list1 (cdr list2))))))
```

7.4 Applications of Lists:

1. Lists can represent sets of objects. Thus, we can use lists to represent a set of numbers, or the students of a class.

2. Lists can be used to represent vectors, with the ith element of the list representing the ith element of the vector. By making lists of lists, you can represent multidimensional vectors also.

3. Lists can be used to represent mathematical and symbolic expressions. eg The expression '(+ b c (* d e)) can be represented as a list containing the elements '+ 'b 'c and the list '(* d e). See the chapter on symbols.

7.5 Procedures with variable number of arguments

Scheme allows procedures to take a variable number of arguments. Here is an example of such a procedure and how it can be called:

```
(define (print a . args)
  (pp a)
  (pp args))

(print 1 2 3)
(print 4 5 6 7)
```

In general, the procedure definition must have a fixed number of parameters, except that the last parameter can be declared as a list parameter, by putting a . before it. Suppose we define a function with n formal parameters, of which the last is a list parameter. Then the call to such a procedure must contain at least n arguments. The first $n-1$ arguments will be bound to the first $n-1$ parameters in the usual manner. All the remaining arguments will be put into a list and the list will be bound to the last parameter. So in the above the parameters `a` and `args` will have the values 1 and (2 3) when called as `(print 1 2 3)` and the values 4 and (5 6 7) when called as `(print 4 5 6 7)`.

Exercises

The first few problems deal with matrices, which are represented as a list of lists, the ith list storing the ith row of the matrix. For example a matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

will be stored as a list ((a b c)(d e f)).

1. Write a function (firstcol mat) which returns the first column of mat, stored in the matrix representation. Thus if mat is the matrix given above, ((a) (d)) should be returned.

2. Write a function (lastcols mat) which returns a matrix which has all columns of mat except the first column. For the matrix above the result should be ((b c) (e f)).

3. Write a function (matappend mat1 mat2) which takes two matrices having the same number of rows and returns a matrix in which each row is the concatenation of the corresponding rows of mat1 and mat2. Thus if mat1 and mat2 are both the matrix given above, we should get ((a b c a b c) (d e f d e f)).

4. Write a function (mmult mat1 mat2) which returns a matrix that is the multiplication of mat1 and mat2. Use the functions defined above if appropriate.

5. Write a function to return the transpose of a given matrix. For example (transpose '((1 2 3) (4 5 6))) should return '((1 4) (2 5) (3 6)). Use the functions given above as appropriate.

6. The cross product of sets $\{s_1, \ldots, s_n\}$ and $\{t_1, \ldots, t_m\}$ is defined as the set of all ordered pairs (s_i, t_j). Write a program that takes as input two lists representing sets, and returns a list representing their cross-product. For example (cross '(1 2 3) '(4 5)) should return the list ((1 4) (1 5) (2 4) (2 5) (3 4) (3 5)). Since the outer list represents a set, note that the elements in it may be present in any order; the inner lists represent ordered pairs, these must be present in the order shown.

7. Write a function DEL that takes a number and a sorted list and returns the result of deleting the number from the list. If the number is absent then the original list should be returned. For example, the call (DEL 5 '(1 2 5 7 8 9)) should return '(1 2 7 8 9). Your function should exploit the fact that the list is sorted.

   Argue as carefully as you can that your function is correct. Your argument must include a precisely written induction hypothesis, including a statement of what the induction is over.

8. Write an imperative style function that returns the smallest number from an array of numbers. Prove its correctness.

9. Consider the following program for reversing a list.

```
(define (reverse L answer)
  (if (null? L)
      answer
      (reverse (cdr L) (cons (car L) answer))))
```

   To reverse a list `L`, the call required is `(reverse L '())`. Thus `(reverse '(1 2 3) '())` returns `'(3 2 1)`.

   (a) Write a precise specification for the function. Write it as formally as possible.

   (b) To prove correctness, what induction hypothesis will you use? The complete proof is not expected; though you should clearly state what the induction is over.

10. Write a function to sort a list of numbers, i.e. rearrange the numbers within the list so that they are in non-decreasing order. For this use the following idea: split the list into its even part (use the function `even-list` defined earlier) and the odd part, recursively sort the two parts, and then merge them together using the `merge` function defined earlier.

Chapter 8

Higher Order Functions

One of the most powerful ideas in functional programming is that programmers should be allowed to manipulate functions just as they manipulate data. For example, functional programming languages allow functions to be passed to other functions as arguments, or be returned as arguments. A function which manipulates other functions in this manner is said to be a *higher order function.*

Higher order functions turn out to be extremely powerful in many ways:

- *Help in writing programs in a modular manner.* An important question in programming is how to develop pieces of a program as independent parts or *modules* which can then be combined conveniently to make full programs.
- *Make programs more intuitive and concise.*
- *Can be used to build data structures.* This may sound strange, but is actually true! Read on.

8.1 Functions as parameters to other functions

The ability to pass functions as parameters to other functions can be used to make programs modular. We will describe two examples of this.

8.1.1 Generalized Root Finder

Consider first, a program that finds the square root of 2. This parameters `xneg` and `xpos` denote the upper and lower ends of an interval in which the square root of two must be guaranteed to lie. Then the program finds the midpoint `xmid` of the interval `[xneg, xpos]` and simply determines in which of the two subintervals `[xneg, xmid]` and `[xmid, xpos]` the square root must lie. Then the program recurses with that interval. The recursion stops when the size of the interval becomes small enough (smaller than 0.01 in particular).

```
(define (sqrt2 xneg xpos)
  (if (< (abs (- xneg xpos)) 0.01)
      xneg
```

```
      (let ((xmid (/ (+ xneg xpos) 2)))
        (if (< (* xmid xmid) 2)
            (sqrt2 xmid xpos)
            (sqrt2 xneg xmid)))))
```

This program will yield the square root of 2 for the call `(sqrt2 1.0 2.0)`, for example.

You may observe, that the basic logic of bisecting the interval and recursing on the appropriate subinterval is useful for many other calculations. In particular, for writing a general program for finding the roots of any function `f`. By passing `f` as a parameter, we may write a general root finder as follows.

```
(define (root f xneg xpos)
  (if (< (abs (- xneg xpos)) 0.01)
      xneg
      (let ((xmid (/ (+ xneg xpos) 2)))
        (if (> 0 (f xmid))
            (root f xmid xpos)
            (root f xneg xmid)))))
```

Once we have such a function root defined, we may use it to find the square root of 2 by writing:

```
(define (f x) (- (* x x) 2.0))
(root f 1.0 2.0)
```

Of course, we can use it to get an approximate root of any function provided we know an interval in which a root lies. For example, the function $g(x) = x - \cos(x)$ has $g(0) = -1$ and $g(1) > 0$. Thus we can find its root by writing:

```
(define (g x) (- x (cos x)))
(root g 0.0 1.0)
```

8.1.2 Animation

Even in our primitive graphics system, it is possible to make objects *move*. Well, they dont actually move, but we can create this illusion by first drawing them in one position, erasing them, and then drawing them in the next position. Erasing does not need anything special – we simply draw the object in the same colour as the background.

There are 3 basic tasks in doing our simple animation:

1. Definition of the shape of the object.
2. The basic operation of drawing and erasing the object.
3. Movement of the object, i.e. the description of the trajectory.

It is desirable that the code for each of the tasks above is written as independently as possible, and be combined only when needed. That way, it would be easy to do the animation for many shapes, or along many trajectories. We can do so by using a function for each task,

and by using higher order functions to combine the functions. Here is how it might be done. The main procedure is `move-shape` shown below. It does the work of drawing and erasing the shape; the precise shape to be drawn is passed to it using the function parameter `shape`. The trajectory along which to move the shape is specified parametrically using the function parameters `x` and `y`.

```
(define (move-shape t dt tf shape x y)
  (if (< t tf)
      (begin (trans (x t) (y t) (shape)
                    (colour "white" (shape)))
             (move-shape (+ t dt) dt tf shape x y))
      (trans (x t) (y t) (shape))))
```

To use procedure `move-shape` we must define the functions for the shape and the trajectory, and pass them as parameters. So we could write:

```
(define (square)
  (line -0.02 -0.02 -0.02 0.02)
  (line -0.02 -0.02 0.02 -0.02)
  (line 0.02 0.02 -0.02 0.02)
  (line 0.02 0.02 0.02 -0.02))

(define (xcirc t) (* 0.5 (cos t)))
(define (ycirc t) (* 0.5 (sin t)))

(move-shape -3.14 0.01 1.57 square xcirc ycirc)
```

8.2 Function Expressions

So far we have seen only identifiers in the first position of any expression, Scheme actually allows expressions to be present here too — it is necessary, however, that these expressions evaluate to functions, and then these functions are applied to the remaining arguments. For example, we can write:

```
((if (= x y)
     +
     *)
 10 20)
```

which will either add the numbers 10 and 20 or multiply them depending upon whether or not x was equal to y.

8.3 Unnamed Functions

It is convenient to be able to construct functions without having to give them names. This is done using the general form

```
(lambda arguement-list body)
```

Thus the expression

```
(lambda (x) (* x x ))
```

returns a function that takes a single argument and returns its square. We can use such expressions[1] in any place wherever a function can be used, e.g.

```
(define g (lambda (x) (* x x)))
((lambda (x) (* x x)) 5)                    ;; returns 25.
(root (lambda (x) (- x (cos x))) 0.0 1.0)   ;; root defined earlier

(move-shape -3.14 0.01 1.57                 ;; move-shape defined earlier
        (lambda ()                          ;; square drawing function
          (line -0.02 -0.02 -0.02 0.02)
          (line -0.02 -0.02 0.02 -0.02)
          (line 0.02 0.02 -0.02 0.02)
          (line 0.02 0.02 0.02 -0.02))
        (lambda (t) (* 0.5 (cos t)))        ;; xcirc equivalent
        (lambda (t) (* 0.5 (sin t)))        ;; ycirc equivalent
)
```

The last two commands show how the functions `root` and `move-shape` defined earlier can be called without separately defining the function parameters needed.

It would have been better if the Scheme language had used a more descriptive term like "make-function", instead of the term "lambda". The term "lambda" comes from a system called "lambda calculus" developed by a Alonzo Church, for analysis of functional programs.

8.4 Exercises

1. Our condition for the error being small enough in `root` was $|f(x)| < 0.01$. Some users might want the constant to be something other than 0.01, or might want to ensure for example that $|f(x)^2| < 0.01$. Extend `root` so that it takes an additional function `close-enough` which can be used to determine whether the approximation is good-enough, i.e. assume that `(close-enough x)` is true only when `x` is close enough to the root.

2. Write a function `sum` which when called as `(sum f i j)` will return

$$\sum_{k=i}^{k=j} f(k)$$

3. Use the above function `sum` to compute $\sum_{k=1}^{k=100} k^2$. Define the function for squaring without using a `define`, instead use a `lambda` expression.

[1] The body argument can be several commands, the value of the function is the value of the last command.

4. Write a function `integral` which when called `(integral f y z n)` returns

$$\int_y^z f(x)dx$$

   Use any convenient rule for approximating integrals into sums, dividing the interval from `y` to `z` into `n` parts.

5. Write a function that returns the numerical derivative of a given function. Use a convenient step size.

6. Extend the `move-shape` code to handle the following cases.

   (a) Suppose the shape must not only move, but must also rotate so that it is always at the same angle with its trajectory. This will look nice if the shape is an aeroplane, for example.

   (b) Suppose the shape internally consists of several shapes, e.g. a moon circling the earth.

   (c) Suppose there are several independent shapes to be moved.

   (d) Suppose it is better to erase the shape at $(x(t), y(t))$ only after the drawing the next one, i.e. at $(x(t + dt), y(t + dt))$.

   Other extensions are also possible, e.g. the shape should move in response to a field that could be defined by the user. Or the shape should bounce of walls in the picture.

Chapter 9

Built-in High Order Functions for lists

Three higher order functions are commonly used: map, reduce, and filters. Scheme actually has two kinds of filters respectively called list-transform-positive and list-transform-negative.

9.1 Map

syntax: (map f l1 l2 ... ln)

Here l1,l2,...ln are n lists having the same length and f is a function taking n arguments. If l_{ij} denotes the jth element of list li, then the output is a list in which the ith element is (f l_{i1} l_{i2} ... l_{in}). In other words, the output is the following.

```
(list
  (f (car l1) (car l2) ... )
  (f (cadr l1) (cadr l2) ... )
  ..........................
)
```

Examples

```
(map + '(1 2 3 4) '(10 20 30 40))
=> '(11 22 33 44)

(map abs '(1 -20 3 -4))
=> '(1 20 3 4)
```

9.2 Reduce:

syntax: (reduce f zero l)

The argument f must be a function of two arguments, and l a list. If l is '(), then zero is returned. If l has just one element, then that element is returned. If l has several elements l1,l2,...,ln, then the following is returned:

```
(f (... (f (f l1 l2) l3) ...) ln)
```

Examples:

```
(reduce + 0 '(1 2 3 4 5))  => 15

(reduce list '() '(1 2 3 4 5)) => ((((1 2) 3) 4) 5)
```

The last example shows that reduce is left associative.

9.3 Filters

syntax: (list-transform-positive list predicate)

This is predefined in scheme, but can be implemented as follows.

```
(define (list-transform-positive list predicate)
    (if (null? list)
        '()
        (if (predicate (car list))
            (cons (car list) (list-transform-positive (cdr list) predicate))
            (list-transform-positive (cdr list) predicate))))
```

An analogous function list-transform-negative returns elements that dont match the predicate. Examples:

```
  (list-transform-positive '(1 2 3 4 5) even?)  => '(2 4)

  (list-transform-negative '(1 2 3 4 5) even?)  => '(1 3 5)
```

Note on some Scheme functions:

Map/reduce require you to pass functions – so it is useful to know which scheme commands are function and which are not. Interestingly, `and` and `or` are not functions– instead they are what is called "special forms". The distinction is that the arguments to a function are first evaluated and then the function is applied; whereas in a special form only those arguments that are actually needed are evaluated. For example, in (and x y z) if x is false, then we can return false without evaluating y and z. This is in fact what scheme does, and thus `and` is a special form in scheme. Similarly, `or` can return a value as soon as any non false value is found, and thus `or` is also a special form. If you want to pass `and` /`or` to map/reduce, define logand/logor as functions as follows and pass them:

```
(define (logand x y) (and x y))
(define (logor x y) (or x y))
```

9.4 Application of Filters: Quicksort

This is a famous sorting algorithm developed by C. A. R. Hoare. First we pick an element of the list (in the program below it happens to be the car), which we call splitter – this element is used to split the input list into three smaller lists consisting respectively of elements smaller than, equal to, and greater than the spliter. Now we apply quicksort function to the first and third lists separately and the results are appended. The lists are created using list-transform-positive.

```
(define (quicksort list )
  (if (null? list )
     '()
     (let ((splitter (car list )))
       (append
        (quicksort (list-transform-positive list
                     (lambda (x) (< x splitter ))))
        (list-transform-positive list
                     (lambda (x) (= x splitter )))
        (quicksort  (list-transform-positive list
                      (lambda (x) (> x splitter ))))))))
```

This algorithm has been found to be very efficient in practice. The number of steps required to sort any list mainly depends on the splitter value. It is possible to prove that if the spliter is median of list then we shall require least number of steps to sort the list.

Exercise

Find how many comparisons take place if you are sorting a list of 1023 distinct elements and if at each step the splitter happens to be the median of the list. Do the same for the case in which the splitter happens to be the smallest element of the list.

9.4.1 Applications of Map and Reduce

1. Matrix multiplication
2. Matrix transpose and many more...

Exercises

1. Write a function that compares two lists containing numbers and determines if they are identical.
2. Using the above function or otherwise, write a function that takes two matrices (each a list of lists of numbers) and determines if they are identical. You are expected to use map and reduce in both parts (less credit for writing code which does not use these).

You may note that logand, logor are binary functions that compute logical and, logical or, respectively. For example `(logand #t #f)` evaluates to #f. Also remember that `(= x y)` evaluates #t or #f depending upon whether the numbers x and y are equal.

3. Write a generalized sorting function which when called as `(gensort f l)` orders `l` accorting to `f`, where `f` must be a predicate taking two arguments. Specifically, if `li` and `lj` are elements of `l` then `li` must appear before `lj` in the result if `(f li lj)` is true. You may modify the `quicksort` code given in the chapter.

4. Suppose you are given a list `M` containing elements which are themselves lists and have the form `(rollno marks1 marks2 marks3 ...)` where `marksi` are the marks in the ith lab assignment. The goal is to be able to sort this in ascending order of marks obtained in any specified assignment. So for example `(sortlab M 5)` should return the list `M` but sorted in ascending order of the marks in the 5th lab. Write the function `sortlab`. Use

5. The term *morphing* has been used to denote the technique of smooth transforming a picture of one object into another. This is commonly used in making commercial advertisements, e.g. recently there was an advertisement for a jeep (or maybe engine oil!) which started of as a running cheetah, which transformed into a jeep as it moved.

   In this problem you are to implement morphing of line drawings. A single drawing P is defined by a sequence of points $p_1, \ldots, p_n$ and consists of line segments $p_i p_{i+1}$. Each p_i is a pair of numbers, i.e. the x coordinate and the y coordinate, of course. The input to the morphing routine are two drawings P and Q, each with n points each, and a frame count t. The morphing routine must draw P first, and then erase and redraw versions of P which are progressively more and more similar to Q, until eventually Q gets drawn. The ith drawing P_i consists of n points $p_{i1}, \ldots, p_{in}$, where p_{ij} lies on the line joining $p_0 j$ and p_{tj} and divides it in the ratio $i : t - i$.

   So for example, the following call will morph a "V" into a flatter "V" which has moved to the left (suggesting a flying bird!):

```
(morph '((-0.1 0.1) (0.0 0.0) (0.1 0.1))    ;; initial configuration
       '((0.0 0.14) (0.05 0.0) (0.1 0.14))  ;; final configuration
       10                                   ;; 10 frames
)
```

6. Implement morphing of one drawing having n points to another drawing of m points. Use some simple idea to split/merge points. For example, if $n > m$ then more than one of the points in the starting drawing will head towards the same point in the target drawing.

7. Suppose I have a complete scene with several drawings which morph. How can I extend the previous routine to handle this?

Chapter 10

Trees

In this lecture we study binary trees, how to draw them, and some of their uses.

A tree is a special kind of a mathematical object called a *graph*. A *graph* is defined by two sets, V and E. V is any set, but E is required to be a set of unordered pairs of elements from V. For example, $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{\{1, 4\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{5, 6\}\}$. The elements of V are said to be nodes or vertices, the elements of E are called edges. An edge $\{u, v\}$ is said to connect vertices u and v. While refering to edges, it is more customary to use the notation (u, v) rather than $\{u, v\}$, and we will use that in the rest of this chapter. Pictorially, a vertices may be represented by points in the plane and edges by lines connecting the corresponding points. The above graph is pictured below.

A *cycle* is a sequence of edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k), (u_k, u_1)$. A graph in which there are no cycles is said to be a tree.

Any vertex in a tree may be designated as the *root*; the vertices that are connected to the root are said to be the *children* of the root. The root is said to be the *parent* of the children. The vertices connected to a child u (except for its parent) are called its children, and so on. Figure 10.1 shows an example. 5 is the root; 4, 6 are the children of 5; 1,2,3 are the children of 4.

A rooted tree in which every vertex has 0, 1, or 2 children is a *binary tree*.

Height The height of a binary tree is the length (number of edges) of the longest root leaf path.

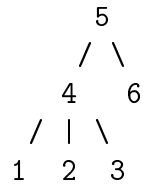

Figure 10.1: A rooted tree

10.1 Representing trees in Scheme

Rooted trees can be represented by noting that they have a recursive structure. You may define a rooted tree as

1. A single root node.

2. A single root node attached to rooted trees.

For example, the tree of Figure 10.1 would be thought as the node 5 connected to a single tree with root 4 and children 1, 2, 3 and also connected to a tree which just consists of the node 6.

Noting this recursive structure we can represent trees using a list. The general pattern is

```
(root-node first-subtree second-subtree ...)
```

So we would represent the above tree as `(5 (4 1 2 3) 6)`.

10.2 Operations on trees

Trees can be processed using their recursive structure. Suppose we wish to find the number of leaf nodes in a tree. This is simply the sum of the number of leaves in each subtree. The code might be something like:

```
(define (numleaves tree)
  (if (list? tree)
      (reduce + 0 (map numleaves (cdr tree)))
      1)
```

10.3 Exercise

1. An expression such as 5*4+3 can be written out as a tree, with operators in the nonleaf nodes, and values at the leaves. Given such a tree, write a function which returns its value.

2. Given an arithmetic expression tree, convert it into a list in which the expression appears as if it were written out. You may need to insert parentheses to enforce operator precedence.

3. Write a function that takes an expression tree and prints it out on the graphics screen. You should allow operators such as
   `f` which are intended to denote division, however the numerator and denominator must be written one above the other, as in $\frac{a}{b}$ rather than a/b. The
   `f` operator maybe nested – so do this properly. Try to make your output resemble expressions printed in a book etc.

4. A scheme object X is said to be a search tree if either (1) X is a number, or (2) X is a list '(A B C) where B is a number and A and C are search trees such that B is larger than the largest number in A (or A itself if A is a number) and smaller than the smallest number in C (or C itself if C is a number). Examples of search trees: 5, '(1 2 3), '((1 2 (3 4 5)) 7 (8 9 100)). The following are not search trees: '(1 2 3 4), '(1 5 (2 3 4)), '((1 2 3)).

   Write a function CHECK that determines whether a list is a search tree. For example (CHECK '(1 2 (3 7 9))) should return #t.

Chapter 11

Analysis of Algorithms

Most of this course is about programming style; our concern is readability, extensibility and user friendliness of programs. Our focus is not on the speed of programs, or on developing fast algorithms. However, the speed of programs is important. It is important that we know how to analyze the speed of our programs. Analysis of algorithms is a vast and deep topic. Our goal here is to get a very brief flavour of it.

Usually in analyzing any algorithm it is customary to decide that a certain operation is the most dominant one, and then attempt to estimate how many times the algorithm performs that operation. The number of times that operation is performed will vary depending upon the size of the input given to the program, and even the precise input itself. So it is customary to ask: for a given input size, how many operations does the program perform over all possible input instances of that size? For example, if we are considering sorting programs, it is customary to count the number of comparisons performed by the algorithm. So we could ask, for input size n, what is the maximum number of comparisons needed to sort any input with n keys? This is typically called the *worst case complexity*. In a similar manner it is possible to define the average case complexity, which is the number of operations needed by the algorithm averaged over all input instances of the given size n.

In this lecture we evaluate the worst case complexity of a selection sort algorithm.

11.1 Selection sort

The basic step of selection sort is to remove the smallest element in the input that is yet to be sorted and appending that element to the output. This may be expressed in scheme as follows:

```
(define (selsort l)
  (if (null? l)
      '()
      (let ((s (smallest l)))
        (cons s (selsort (remove s l))))))

(define (smallest l)
  (define (smallestwithc candidate l)
```

```
    (if (null? l)
        candidate
        (if (<= candidate (car l))
            (smallestwithc candidate (cdr l))
            (smallestwithc (car l) (cdr l)))))
  (smallestwithc (car l) (cdr l))
)


(define (remove e l)   ;; assuming e occurs in l
  (if (= e (car l))
      (cdr l)
      (cons (car l) (remove e (cdr l)))))
```

We will estimate the number of comparisons performed by selsort. To do this we need to estimate the number of comparisons performed by smallest and also by remove. This done by writing *recurrence equations*.

11.2 Analysis of smallest

Smallest called on a list of length n simply calls smallestwithc (i.e. short for "smallest with candidate") with a list of length n-1 and a candidate which is just the first element of the initial list. So we should simply analyze the number of comparisons performed by smallestwithc.

Suppose x_i denotes the number of comparisons performed by smallestwithc when it is called on a list of length i. Now, for the empty list (length = 0) the function simply returns the candidate without doing any comparisons. Thus $x_0 = 0$. If it is called on any list of length $i > 0$, then it does one comparison, and then calls itself on a list of length $i - 1$. The number of comparisons done in the recursive call is x_{i-1} by definition. Thus we have:

$$x_i = 1 + x_{i-1}$$

where the equation holds for all $i > 1$. In other words, we have the entire family of equations $x_1 = 1 + x_0$, $x_2 = 1 + x_1$, and so on. Such a collection of equations is called a *recurrence*. The simplest way to solve a recurrence is to substitute the equations into one another:

$$x_i = 1 + x_{i-1} = 1 + 1 + x_{i-2} = 1 + 1 + 1 + x_i - 3 = \ldots = 1 + \cdots + 1 + x_0$$

where the number of 1s in the last expression is clearly i. But since $x_1 = 0$, we get $x_i = i$.

Thus when smallest is called with a list of length n it must be the case that $x_{n-1} = n - 1$ comparisons are performed by smallestwithc which gets called.

11.3 Analysis of remove

We will leave it to the reader to verify similarly as above that the number of comparisons needed to remove an element from a list of length n in the worst case is n.

11.3.1 Analysis of selsort

Let y_i denote the number of comparisons needed to sort a list of length i in the worst case (i.e. for that input of length i for which the maximum number of comparisons are needed).

When called on an empty list, the function returns directly, so clearly $y_0 = 0$. If called on a list of length $i > 0$, it first calls smallest on the input list, then calls remove on the input list, and then finally recursively calls itself on a list of length one less, i.e. length $i-1$. Thus we have:

$$y_i = i - 1 + i + y_{i-1} = 2i - 1 + y_{i-1}$$

We can solve this recurrence by substitution as before. So we get:

$$y_i = 2i - 1 + y_{i-1} = 2i - 1 + 2(i-1) - 1 + y_{i-2} = 2i - 1 + 2(i-1) - 1 + \cdots + 3 + 1 + y_0$$

The last expression is simply an arithmetic series of odd numbers from 1 to $2i - 1$ to which is added $y_0 = 0$. But the arithmetic series sums up $2i(i/2) = i^2$.

Thus selsort takes n^2 comparisons to sort any list of length n in the worst case.

11.4 Exercises

1. Estimate the number of comparisons needed by other sorting algorithms.

2. Write a scheme program to compute x^n where n is an integer, using the following idea (to be used recursively):

   (a) If n is even, then $x^n = \left(x^{n/2}\right)^2$.

   (b) If n is odd, then $x^n = \left(x^{(n-1)/2}\right)^2 \times x$.

   How many multiplications does your program need for computing 2^{17}? You are expected to estimate this as follows. Let M_n denote the number of multiplications needed to compute x^n. Can you relate M_{17} to other M_i and so on?

Chapter 12

Matrices And Shortest Path Problems

We will see an unusual kinds of matrix multiplication in this lecture. This will enable us to compute shortest path in a map. It will also provide a good example of higher order functions.

Suppose we are given a matrix A such that a_{ij} gives the distance between cities i and j, if they are connected by road directly. If cities i and j are not connected directly, then we will have $a_{ij} = \infty$. Further, $a_{ii} = 0$ for all i, of course. For example consider the road map shown in Figure 12.1.

We could represent it as the following matrix (with 1=Mumbai, 2=Pune, 3=Nashik, 4=Kolhapur, 5=Nagpur):

$$\begin{bmatrix} 0 & 160 & 200 & 450 & \infty \\ 160 & 0 & 220 & 350 & \infty \\ 200 & 220 & 0 & \infty & 500 \\ 450 & 350 & \infty & 0 & \infty \\ \infty & \infty & 500 & \infty & 0 \end{bmatrix}$$

The usual matrix multiplication $C = AB$ of matrices A, B is defined as consisting of elements $c_{ij} = \sum_k a_{ik} * b_{kj}$. Suppose we define c_{ij} using addition in place of multiplication and taking the minimum instead of summing, i.e. let us define:

$$c_{ij} = \min\{a_{i1} + b_{1j}, a_{i2} + b_{2j}, \ldots, a_{in} + b_{nj}\}$$

assuming n is the number of rows in A. Now consider what happens we compute A^2 using this definition, i.e. If $D = A^2$ we have

$$d_{ij} = \min\{a_{i1} + a_{1j}, a_{i2} + a_{2j}, \ldots, a_{in} + a_{nj}\}$$

But now note that $a_{ik} + b_{kj}$ is simply the length of the path if we go from city i to city j, passing only through city k on the way. But we take the minimum value of $a_{ik} + a_{kj}$ over all all possible k (including $k = i, j$ as well). Thus d_{ij} as calculated above is nothing but the length of the shortest path from i to j passing through any single city along the way, however the single intermediate city could be i or j itself. Let us say that a path has k hops if it passes through $k - 1$ cities (excluding the origin and the destination). Thus it follows:

> The ijth entry of A^2 give the length of the shortest path having at most 2 hops from city i to city j.

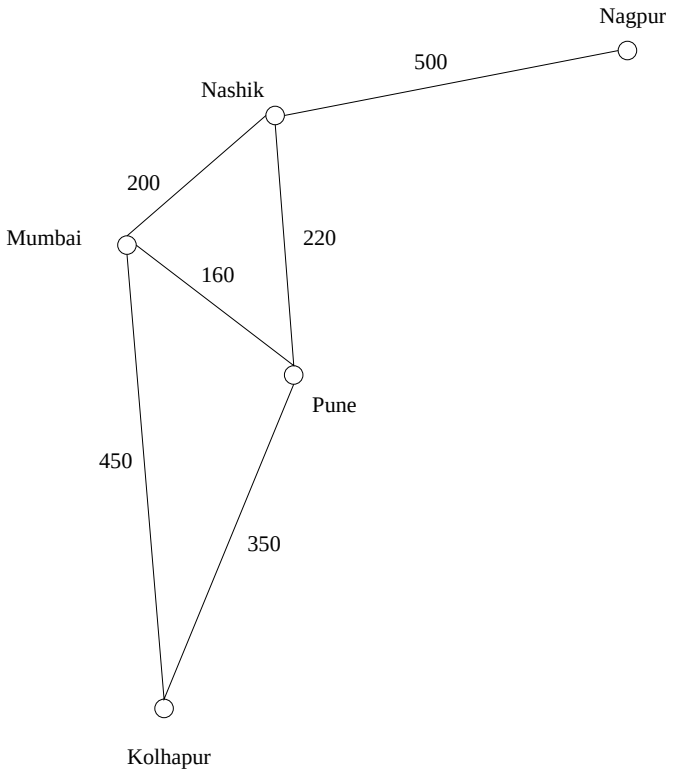

Figure 12.1: Schematic Map

We could continue this process! Thus by squaring the matrix A^2 we will obtain the matrix A^4 which can be seen as giving the lengths of all paths with at most 4 hops (i.e. which have at most 3 other cities on the way). So squaring $\lceil \log n - 1 \rceil$ times, we will obtain a matrix A^m where $m = 2^{\lceil \log n - 1 \rceil} \geq n - 1$. But the entries of this matrix will denote the length of shortest paths having at most $m \geq n - 1$ hops. But no shortest path need have more than $n - 1$ hops (else a city would be repeated, and we can remove these circuitous paths). Thus A^m will give the lengths of the shortest paths.

Exercises

1. Compute A^2, A^3, A^4 for the matrix given earlier. Verify that $A^4 = A^5 = A^6 = \ldots$.

2. Write a matrix multiplication function that takes as parameters the two matrices, a function add, a function mult and a value zero (which is the identity for the add function, i.e. (add zero x) =¿ x.). It should be possible to get ordinary matrix multiplication by specifying +, *, 0 for the parameters add, mult and 0.

3. Write binary functions imin and i+ that normally require the arguments to be numbers, but can also allow the arguments to be the symbol 'infinity. i+ should return 'infinity if either argument is 'infinity, otherwise the sum. imin should return the min if both arguments are numbers, 'infinity if both arguments are 'infinity, and the number if one of the argument is a number and the other is 'infinity.

4. Write a function which uses the matrix multiplication routine of the first problem and the functions defined in the second problem to find the length of the shortest paths in a map. Assume the input is a matrix giving distance between cities as discussed. Use the symbol 'infinity to represent ∞. The output of the function will be a matrix, in which the ij entry will give the length of the shortest path between i and j. In essence you will be computing the n-1th power of the input matrix.

5. Write binary functions pmin and p+, which take as input some representation of a path, e.g. (length-of-the-path list-of-cities-on-the-path). pmin simply returns the path having the smaller length, while p+ returns the path obtained by concatenating the given two paths. You are to decide on the precise representation of the path as per your convenience.

6. Write a function which uses the matrix multiply routine of problem 1 and the functions pmin and p+ to generate the shortest paths themselves (not just the length of the shortest path). In particular, the output should be a matrix, in which the ij entry gives the length of the shortest path as well as the path itself (specified as a list of cities along the path). The input should be the same as the input for problem 3, i.e. a matrix whose ij element gives the distance from city i to city j. From this you should generate a matrix whose ij element describes the path from city i to city j. Then you should simply compute the n-1th power of this matrix.

Chapter 13

Symbolic Computing

In addition to the numeric data, scheme allows processing of symbolic data. A symbol is almost the same as a character string – but while a character string may contain spaces, a symbol is any string that satisfies the rules of being an identifier in Scheme. We will think of symbols in this manner as being simple kinds of strings, we will not consider some of the more complicated aspects of symbols that the Scheme interpreter allows.

A symbol `s` is represented as `'s`. In Scheme, the quote is used to prevent the evaluation of what follows, i.e. whatever follows is to be taken literally rather than thought of as representing its value. Thus if you type `s` the Scheme interpreter will assume you mean the value of `s`, if you type `'s` the Scheme interpreter assume you mean `s` itself, i.e. the symbol `s`. The quote has been used with lists earlier for the same purpose, we similarly allow symbols to be inside lists as well.

```
(define r 'a)                  ; r has the value symbol a
(define r a)                   ; r has the same value as the value of
                               ; assuming a has been defined earlier.
(define p '(a b c))            ; list of symbols a b and c.
(define q (list a b c))        ; list of the values of a b c.
```

A list containing symbols is called a symbolic expression or an *s-expression*. The usual list operations work on s-expressions, of course. For example in the above example `(car p)` would return `'a`.

13.1 Uses of s-expressions

S-expressions are a general way of representing symbolic data are mathematical expressions, programs, rules. For example:

```
(define a '(* x (sin x)))

(define b '(define (square x) (* x x)))
```

Such symbolic data can be processed as if it is a list, using the usual list operations. We give some examples of such processing.

13.1.1 Symbolic Differentiation

You have probably seen how a function could be *numerically* differentiated, i.e. evaluating the formula $f'(x) \approx (f(x+h) - f(x))/h$ for some small h. However, if a function is given to us symbolically, we can find its derivative symbolically, using the usual rules of differentiation e.g.

$$\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$$

$$\frac{d}{dx}(uv) = v\frac{d}{dx}u + u\frac{d}{dx}v$$

$$\frac{d}{dx}\sin x = \cos x$$

$$\frac{d}{dx}g(u) = \frac{d}{dx}g(x)\frac{d}{dx}u$$

These rules are recursive! Thus they can easily be encoded in Scheme. Here is the code for the first 2 rules and the simple rules that the derivative of x itself is 1, and that of any other constant is 0.

```
(define (deriv s)
  (cond ((null? s) '())
        ((not (list? s)) (if (eq? s 'x)
                             1
                             0))
              (else (let ((op (car s))
                          (args (cdr s)))
                (cond ((eq? op '+) (cons '+ (map deriv args)))
                      ((eq? op '*) (list '+ (list '* (car args)
                                                  (deriv (cadr args)))
                                            (list '* (cadr args)
                                                  (deriv (car args)))))
                      )))))
```

If you now execute `(deriv '(* x x))` Scheme will return `(+ (* x 1) (* x 1))` which in fact can be simplified to $2x$, which is indeed the derivative of x^2 as required.

Exercises

1. Extend the deriv function given above to handle, trigonometric functions, exponentiation and natural log.

2. Write a function `(simplify s)` which will simplify an arbitrary algebraic expression s. The idea is to use this to simplify the output of deriv. So if you write `(simplify (deriv '(* x x)))` you should get `(* 2 x)`.

   Be warned that simplification is hard! The program will need to know about properties of 0 and 1 vis-a-vis addition and multiplication, as well as commutativity, distribution of addition over multiplication, and possibly other things. Write the program in stages, incorporating different rules progressively.

3. Write a program for doing symbolic integration. This is also hard.

13.1.2 Processing Programs

An s-expression can very well be representing a program, and so we could process it to augment the program, or find out its properties. Or we could translate the program from one language to another. These are processes that happen in compilers, for example.

We will consider a very simple example as an illustration: translating a program from Scheme to Lisp. These languages are very similar, though there still are many differences. We will pretend that there is only one difference– the way in which functions are defined in Lisp. The form is `(defun function-name (arg1 arg2 ...) function-body)`. So the `square` function would be written as

```
(defun square (x) (* x x))
```

Here is a program which will do this translation. It does not handle all the cases though.

```
(define (trans def)
  (let ((name (caadr def))
        (args (cdadr def))
        (body (cddr def)))
    (append (list 'defun name args) body)))
```

Exercises

1. Extend the translator to translate nested definitions as well.

2. Scheme has a construct `trace` which can be used to print the arguments of a function everytime it is called, and the result returned by the function as it exits. Write a function `(trans s)` which transforms a function definition to produce this effect. Given below is an example input:

   ```
   (trans '(define (f x y z) body))
   ```

   which should evaluate to

   ```
   (define (f x y z)
     (print x y z)
     (let ((result body))
        (print result)
        result))
   ```

   Here the function `print` simply prints its arguments on the screen (assume this function is given).

Chapter 14

More on Higher Order Functions

14.1 Functions returning functions

The ability to return functions as a result of a function call is a programming convenience; in addition it provides some unexpected power: the ability to simulate data structures.

We give a simple example first. Here is a function genquad which takes arguments a,b,c and returns a function which may be used to evaluate the quadratic $ax^2 + bx + c$. Genquad may be invoked several times with different parameters to generate different quadratics.

```
(define (genquad a b c)
 (define (f x)
  (+ (* a x x) (* b x) c))      ;; A
 f)

(define g (genquad 1 2 3))      ;; B
(define h (genquad 4 5 6))      ;; C
(g 7)                           ;; D
(h 8)                           ;; E
```

Lines B and C store functions in variables g and h. Note however, that the function f returned by genquad makes a reference to variables a,b,c which are not the formal parameters of f. What values do these variables take, when the functions g and h are evaluated on lines D and E?

The rule used in Scheme is that the variables inside the body of a returned function will be the same variables as would have been used if the function had been invoked in the same environment in which it was created. Thus, g got defined using the call (genquad 1 2 3) – the function f returned as a result was defined in an environment in which the variables a,b,c had the values 1,2,3 respectively. Thus when (g 7) is executed, we will have x=7, and a,b,c will be 1,2,3 respectively. Similarly, (h 8) will execute f with values of a,b,c,x being 4,5,6,8 respectively.

14.1.1 Simulation of Data Structures

We can simulate lists using high order functions.

```
(define (make-pair a b)
   (define (f command)
         (cond ((eq? command 'car) a)
               ((eq? command 'cdr) b)
               ))
   f)
```

Now consider the two statements:

```
(define x (make-pair 6
            (make-pair 7 (make-pair 8 (make-pair 9 'empty-list)))))
(define y (cons 6 (cons 7 (cons 8 (cons 9 '())))))
```

Both the statements could be thought of as creating a list of the numbers 6,7,8,9. The means of accessing the lists are not the same, but we can surely access the elements from x just as well as we can from them. So to access the third element we would write

```
(car (cdr (cdr y)))
(((x 'cdr) 'cdr) 'car)
```

As you can see, both will yield the value 8 of the third element of the list. The above code represents the empty list by the symbol 'empty-list, but this is not necessary; '() could have been used as well. In any case, the conclusion is:

Given Higher Order Functions, the language need not separately provide data structure; data structures can be constructed out of higher order functions!!

14.1.2 Simulating let command using functions

Suppose we have a let statement:

```
(let ((a 1)
      (b (* c d)))
   body
   )
```

It can be verified that its effect is the same as the following statement which instead uses a lambda.

```
((lambda (a b)
   body)
   1 (* c d))
```

Since the lambda form is called with arguments 1 and (* c d), the formal parameters a and b acquire these values, and the body gets executed. The value of the form is the value of the last line of the body. But this is in fact exactly what would happen if we used a let as above!

14.1.3 Simulating do commands using functions

Implementation of DO LOOPS using lambda is done as a lab exercise.

14.2 Unnamed recursive functions

The question considered here is: can we use `lambda` expressions to define unnamed recursive functions? Note that *named* recursive functions are easily defined using `lambda`. For example, it is perfectly acceptable to write

```
(define fact (lambda (n)
               (if (= n 0)
                   1
                   (* n (fact (- n 1))))))
```

This definition is equivalent to the standard one. However, it does define the name `fact`. Can this be avoided? At first glance, it would seem impossible, because in the body of the `lambda` when we make the recursive call, we would need to name the function. So we would need to define the name explicitly.

It turns out however, that `lambda` expressions can be used to define unnamed recursive functions as well; how to do it is slightly complex, but very elegant. It also exposes some hidden power of the `lambda` form. Hence we will discuss it. It is good to understand this section if you wish to make certain that you *really* understand the `lambda` command. However be warned that some mental gymnastics are involved.

Here is how the `fact` function must be expressed:

```
(lambda (n f)
    (if (= n 0)
        1
        (* n (f (- n 1) f))))
```

The first important idea is already in the expression above: the function contains an extra argument `f` in addition to `n`. Thus this `lambda` expression is a higher order function. Now comes the second idea. When we wish to apply this function to a number, e.g. 5 (to compute 5!) we need to supply the second argument as well. So in this case we supply itself! This completes the construction. More precisely, 5! could be computed by writing:

```
((lambda (n f)
   (if (= n 0)
       1
       (* n (f (- n 1) f))))
 5
 (lambda (n f)
   (if (= n 0)
       1
       (* n (f (- n 1) f)))))
```

This is only slightly different from how we call an ordinary unnamed function. For example if we wish to evaluate the polynomial $x^2 + 3x + 10$ and $x = 5$ we would write

```
((lambda (x)
   (+ (* x x) (* 3 x) 10))
 5
)
```

This story has two morals. First, if you allow functions to take other functions as parameters, then you dont need to explicitly allow functions to be recursive. The second moral is that with the above construction it is possible to entirely eliminate `define` statements from any program!

14.2.1 Exercise

1. What is the result of executing the following?

```
((lambda (n f)
   (f n f))
 5
 (lambda (n f)
   (if (= n 0)
       1
       (* n (f (- n 1) f)))))
```

2. Show how an unnamed function could be constructed for constructing gcd. Here is the standard `gcd` for reference.

```
(define (gcd x y)
   (if (= (remainder x y) 0)
       y
       (gcd y (remainder x y))))
```

   Show how you would use your unnamed function to compute the gcd of 119 and 34.

Chapter 15

Relations

A relation is a set of k-tuples, for some fixed k. Each k-tuple in the relation is often called a row of the relation. The set of all the ith elements of all the k-tuples is called a column of the relation. These names are based on the pictorial interpretation of writing down a relation so that each tuple is written out horizontally, one tuple in each line. Remember that the tuples are unordered within the relation, whereas the elements of each tuple are ordered within each tuple.

Relations are commonly used in building databases. For example, information about student roll numbers and hostels could be stored in a relation in which there is a column each for storing roll number, name, and hostel number, and there is a row for each student.

Relations can be manipulated by performing predefined operations on them such as *select, project, count, cross-product* and *join*.

15.1 Selection

The select operation has the format

```
(select relation predicate)
```

The predicate specified must take as argument any row of the specified relation. The value of a select operation is a new relation consisting only of the rows in the specified relation that match the specified predicate.

You should realize that select is simply a special case of list-transform-positive.

15.2 Projection

The format is:

```
(project relation list-of-columns)
```

The value of this operation is a new relation in which only the specified columns are present, in the order they are named in list-of-columns. Note that we cannot implement this relation simply by dropping columns – this could cause two rows to become identical in which case we need to remove duplicates.

15.3 Count

This is simply the number of rows in the relation, i.e. can be implemented in scheme by using the length built-in function for lists.

15.4 Cross Product

We define the cross product of two relations A = (a1,a2,a3..,an) and B = (b1,b2,b3...,bm) as the set of all possible rows obtained by concatenating any row of A with any row of B. The cross product has as many rows as the product of the number of rows in A and B, and as many columns as the sum of the columns in A and B. Here is an example:

```
A = |1 | 10|          B = |5 | 55|
    |2 | 20|              |4 | 30|
    |3 | 33|

A X B  =  |1 10 5 55|
          |1 10 4 30|
          |2 20 5 55|
          |2 20 4 30|
          |3 33 5 55|
          |3 33 4 30|
```

15.4.1 Applications of the cross-product

The concept of cross product is employed in many applications -

Consider a relation representing the son-father relationship - the first field name specifies the son and the second field name the father.

```
Let  R = |A B|
         |B C|
         |D B|
```

Taking the cross product of R with itself -

```
R X R =  |A B A B|
         |A B B C|
         |A B D B|
         |B C A B|
         |B C B C|
         |B C D B|
         |D B A B|
         |D B B C|
         |D B D B|
```

Selecting those elements which have column2 = column 3 -

```
|A B B C|
|D B B C|
```

This represents those combinations in which the entry specifies the grandfather-father-son of a family.

15.5 The JOIN function

SYNTAX : join(r1,field1,r2,field2)

The function returns a relation which has the con-catenation of all elements of r1 and r2 in which the field1 and field 2 respectively match.

eg. join(r1,2,r2,1)

```
when r1 = |e f|     and    r2 = |f g|
          |d f|                 |b h|

returns    |e f f g|
           |d f f g|
```

15.6 Examples

Consider a relation R1 consisting of two columns, the first one naming the father, and the second a son. For example:

R1 = {(Dashrath,Ram), (Dashrath,Laxman), (Ram,Lava), (Ram,Kusha)}

Consider a relation R2 of two columns in which the first column names the husband and second the wife.

R2 = {(Dashrath,Kousalya), (Dashrath,Kaikeyi), (Dashrath,Sumitra), (Ram,Sita), (Laxman,Urmila)}

Now if we want to find the number of sons of Dashrath, we could use

Count(Select(R1, column 1="Dashrath"))

Or to construct a relation between father-in-law and daughter-in-law we could write:

Project(Join(R1,2,R2,1),'(1,4))

Exercises

1. Write code for all the primitives. Make sure that you remove duplicates while implementing the project operation.

2. How will you construct a relation in which each row names (two) brothers?

3. Express the join relation in terms of the other relations.

Chapter 16

Memoization

Memoization is a very powerful idea for making many recursive algorithms fast. Here we will describe it for the fibonacci program.

PROBLEM: Write a function that calculates the n'th Fibonacci number.

The simple minded solution to this is:

```
(define (fib i)
  (if (< i 2)
      1
      (+ (fib (- i 1)) (fib (- i 2)))))
```

The above solution is wasteful. For example, (fib 5) will call (fib 3) and (fib 4); but fibonacci(4) will itself call (fib 3) again. Clearly, we dont need to repeat this call again – all we need is to save a result once we calculate it.

Here is how:

```
(define (memofib n)
  (define memo (make-vector (+ n 1)))    ;; vector of n+1 nulls.
  (define (init-memo)
    (vector-set! memo 0 1)               ;; vector indexing starts at 0
    (vector-set! memo 1 1))
  (define (fib n )
    (if (number? (vector-ref memo n))    ;; nth number already calculated?
        (vector-ref memo n)
        (begin
          (vector-set! memo n (+ (fib (- n 1)) (fib (- n 2))))
          (vector-ref memo n))))
  (init-memo)
  (fib n))
```

This program is very similar to the simple fibonacci program – it just checks to see if the work has already been done, otherwise it does the work. After doing the work, the result is saved in a vector so that it can be used later if necessary.

Exercises

1. Run the original program as well as the optimized program for a large value of n. The difference should be immediately apparent.

2. How many recursive calls does a call to the original program result in? You may express your answer in terms of the Fibonacci numbers themselves.

3. COIN CHANGING PROBLEM. Given an unlimited supply of coins of certain denomination, find the minimum number of coins needed to give change for any specified amount. The denominations should be taken as an input to the program in a list, i.e. the function should look like

   (change amount list-of-coins)

   For example (change 40 '(1 2 5 10 20 25 50)) should return 2.

   Hint: Suppose Change(n) denote the smallest number of coins needed to give change of n paise. If P is one of the allowed denomination, then clearly $Change(n) \leq 1 + Change(n - P)$. This is because I can add a P paise coin to the optimal way to give change for n-P paise and get change for n paise which may not necessarily the optimal way. But the above inequality must be true for some denomination and hence we get

   $$Change(n) = 1 + \min_P Change(n - P)$$

   For example, if the denominations are 1,2,5,10, then this condition means that

   $$Change(n) = 1 + \min\{Change(n-1), Change(n-2), Change(n-5), Change(n-10)\}$$

   Use this inequality to write a recursive program. Use memoization to make the program efficient.

4. We implemented memoization by using the `set!` primitive. This is not strictly necessary, as it turns out. See if you can discover why. Invent a scheme to do this. Your scheme should work in general; but to explain it you should give the code for computing Fibonacci numbers in this manner. Your code should be in the functional style. However be warned that it will probably be more complex to understand as compared to the code that uses `set!` given in the text.

   (Hint: Pass a list of the previously computed results to each recursive call, and have each call return not only the solution but also the list of all results computed till that point. Utilize the new list for subsequent computation. You need not write the function to lookup from the list, but you should carefully state what such a function would have to do.)

Chapter 17

Embedded Languages

The most interesting programs that you see around you are *programming environments*. These do not take an input and produce an output in the conventional sense, they allow you to develop programs which in turn take inputs and produce outputs. The scheme interpreter is one such program; but there are others. For example there are simulation environments in which you can write programs that describe complicated machines (say circuits or chemical reactors) and then you need to describe the inputs that are to be given to these machines and the simulation environments will tell you the output produced by your circuits or reactors. Another example of a programming environment is MATLAB, which is a language for manipulating matrices. So the MATLAB program allows you to write a program in the MATLAB language and helps you to execute that program.

The Scheme interpreter is interesting in that it can be adapted to behave like a specialized programming environment. This is typically done by two mechanisms. The first mechanism is of course the development of libraries of procedures which do the required collection of specialized tasks (e.g. matrix manipulation, simulation of circuits etc.). But in addition to procedures, the Scheme language also has *special forms* – these are expressions which evaluate the arguments passed to them in some special manner. Scheme provides facilities viz. *macros* for effectively defining new special forms. In some sense it is the special forms that give a language its character (the procedure mechanism is the same in most languages) – thus the facility to define special forms is akin to the ability to define your own language.

In fact, it is this facility that makes Scheme an ideal language in which to design special purpose programming systems. For example, by suitably designing libraries of procedures and designing a few special forms, Scheme can be made to work like an Object Oriented, or a Logic Programming Language. This is often referred to as *embedding* new language primitives (e.g. object-oriented or logic programming primitives) into an existing language (e.g. Scheme).

In the next chapter, we will see a very simple example of this embedding. Although the example is very simple, it is actually very useful, and it illustrates the basic issues in embedding primitives from one language into another.

Here we discuss the macro facility in scheme.

17.1 Macros

The syntax for defining a macro is as follows:

```
(define-global-macro  (macro-name macro-parameters)
        body
        )
```

For example:

```
(define-global-macro (increment a)      ;; macro definition
        (list 'set! a (list '+ a 1))
        )
(define b 100)
(increment b)                           ;; macro invocation
```

A macro invocation is processed in 2 phases, an *expansion* phase and an *evaluation phase*. In the expansion phase, the macro invocation is textually replaced by the *expansion result*. In the evaluation phase, the expansion result is treated as ordinary Scheme code and evaluated.

The expansion phase is as follows: (1) The arguments of the macro invocation are not evaluated, but are bound as such to the formal parameters of the macro. In case of the invocation `(increment b)`, the formal parameter `a` is bound to the symbol `'b`. (2) The macro definition is then evaluated. This is basically the same as the evaluation of any Scheme procedure. For our example, the result is `(list 'set!  'b (list '+ 'b 1))`, i.e. `'(set!  b (+ b 1))`.

This result is evaluated in the expansion phase. Thus the invocation `(increment b)` is expanded to `(set!  b (+ b 1))`, which is then evaluated. The overall effect is to set `b` to the value 101.

Macro expansions may contain further invocations of macros. Macros may in fact be recursive; however this must be done carefully. This is described further in Section 17.1.2.

Note that other languages such as C also provide macros (using the so called preprocessor facility), and these are in spirit similar to Scheme macros.

17.1.1 Convenient notation for writing macros

The following notation is often used while writing macros. There are 3 command:

1. The backquote command denoted by the character ‘ having the same name
2. The unquote command denoted by the character comma `,`
3. The splice-unquote command denoted by the characters `,@`

The backquote command is a genaralization of an ordinary quote `'`. Like an ordinary quote, it may be followed by a symbol, in which case the effect is identical. Like an ordinary quote it may be followed by a list which does not contain the unquote or splice-unquote commands, in which case again it behaves like an ordinary quote.

If an unquote command appears inside a backquoted list, then the value of whatever is unquoted is used. For example, we might have:

```
(define a 100)
'(a)
`(,a)
```

The commands `'(a)` and `` `(a) `` will both evaluate to a list consisting of the symbol `'a`, as usual. The command `` `(,a) `` however will evaluate to a list containing the value of `a`, i.e. the list '(100).

The unquote-splice command causes the value to be used, but the value is *spliced* into the list. So suppose we have:

```
(define a '(10 20 30))
```

Then `'(1 a 2)` will denote a list consisting of the number 1, the symbol `'a` and the number 2; the command `` `(1 ,a 2) `` will denote the list '(1 (10 20 30) 2), and the command `` `(1 ,@a 2) `` will denote the list '(1 10 20 30 2).

Using the above notation, the increment macro could be written as:

```
(define-global-macro (increment a)
        `(set! ,a (+ ,a 1))
        )
```

Notice that this is much more readable and compact.

As another example, consider the following macro which will mimic the trace facility of Scheme.

```
(define-global-macro (tdefine template body)
  (pp template)            ;; only for debugging,
  (pp body)                ;; maybe omitted.
  `(define ,template
     (pp (list ,@(cdr template)))
     (let ((result ,body))
       (pp result)
       result)))
```

So if you now type

```
(tdefine (sum x y z) (+ x y z))
```

the macro will translate this to:

```
(define (sum x y z)
   (pp (list x y z))
   (let ((result (+ x y z)))
      (pp result)
      result))
```

Relationship between quote and backquote: At first, it might seem that every form involving a backquote and the unquote/splice-unquote operators can be naturally converted to one involving only the usual quote. For example, `` `(a ,b ,@c d) `` can also be written as `(append (list 'a b) c (list 'd))`. However, a backquoted expression can contain a quote, for example consider the code:

```
(define c 'd)
`(set! a ',c)
```

The second line will now evaluate to `'(set! a 'd)`. There is no easy way to produce this result without using backquotes.

17.1.2 Nesting and recursion

We may use all usual Scheme features such as nesting and recursion. Here is an example.

```
(define-global-macro (incrementmany L)  ;; increment many variables
  (if (null? L)
      '()
      `(begin
         (set! ,(car L) (+ ,(car L) 1))
         (incrementmany ,(cdr L)))))

(define x 1)
(define y 2)
(incrementmany (x y))                   ;; invocation.
```

The invocation will cause `L` to be bound to the list `(x y)`. Since this list is not null, the first expansion will return the result

```
(begin
  (set! x (+ x 1))
  (incrementmany (y)))
```

But this result itself contains a (recursive) invocation. For this invocation the parameter `L` will be bound to the list `(y)`. The result of this will be

```
(begin
  (set! y (+ y 1))
  (incrementmany ()))
```

This also contains an invocation of `incrementmany`, this time with parameter `L` bound to an empty list. Thus this will return the result `'()`. Thus, the overall expansion will be:

```
(begin
  (set! x (+ x 1))
  (begin
     (set! y (+ y 1))
     '()))
```

When this code is executed, the effect will be to increment both `x` and `y`.

It is important to be careful while using recursion in macros, however. Here is an (erroneous) recursive implementation of the foreach macro of Exercise 5.

```
(define-global-macro (foreach x L command)   ;; Does not work!
  `(if (null? ,L)
       '()
       (begin
         (let ((,x (car ,L)))
           ,command)
         (foreach ,x (cdr ,L) ,command))))

(foreach i '(1 2 3) (pp i))                  ;; invocation
```

The result of expanding the above invocation is:

```
(if (null? '(1 2 3))
    '()
    (begin
       (let ((i (car '(1 2 3))))
         ,(pp i)
       (foreach i (cdr '(1 2 3)) (pp i)))))
```

It is now easy to see why this wont work. The expansion will contain the recursive call `(foreach i (cdr '(1 2 3)) (pp i))`. This is not the same as `(foreach i '(2 3) (pp i))`! This is because the arguments to a macro are *not* evaluated before expansion. So the recursive macro expansion will be called with parameter `L` set to `'(cdr '(1 2 3))`. In other words, the recursion will not terminate! Constrast this with the proper use of recursion given above.

Exercises

1. Write the following without using the backquote `` `(a ,b (c ,@d) e) ``.
2. Write the following using the backquote `(list 'a 'b c 'd)`.
3. Write a macro such that `(dotimes i n body)` will expand to

   ```
   (do ((i 0 (+ i 1)))
       ((= i n))
    body)
   ```

   i.e. the idea is that the body should be executed n-1 times with i taking values from 0 to n-1.
4. Modify the tdefine macro given above so that it prints the arguments with an explanatory message, i.e. if we invoke `(sum 1 2 3)` currently the message printed would just be `(1 2 3)`. Instead get it to print `(Calling (sum 1 2 3))`.
5. Write a macro `(foreach varname List command)` that will execute the specified command for every element of the given list. The variable specified as `varname` can be used in the `command`, and it will take as values successive elements of `List`. For example:

```
(define L '(1 3 5 7))
(foreach x L (pp x))
```

This should cause `pp` to be invoked on every element of `L`, i.e. it should print out 1, 3, 5, 7. `(foreach ...)` is not expected to have a value. While writing the macro, you may need to use temporary variables– you may use variable names of the form `foreach00`, `foreach01`, ..., and you may assume that these names will never be used in the user program. (Using the temporary variables is not necessary, the problem can also be solved without any temporary variables; but you are free to use them if you wish.)

Chapter 18

Embedding Advanced Graphics into Scheme

Scheme provides fairly low level primitives for graphics. The main drawing procedures are `graphics-draw-point` and `graphics-draw-line`. These procedures take as arguments coordinates of points (and line endpoints) with respect to the screen. The primitives that we have used in this course nicer in that they allow alternate frames of reference to be defined (implicitly). For example, in the code

```
(line 0 0 0.1 0.1)                    ;; line 1
(trans 0.5 0.5 (line 0 0 0.1 0.1))    ;; line 2
```

although line 1 and line 2 appear to have identical endpoints, in reality they will be drawn differently, because line 2 is drawn after translating the origin to (0.5,0.5). Such a facility makes it very convenient to abstract pictures as procedures; suppose `(window)` causes a window to be drawn, then invoking `(trans dx dy (window))` will cause a window to be drawn shifted by `(dx,dy)`.

In this chapter we will study how these primitives `trans, scale` etc. have been provided. There are two aspects:

1. Algorithm and data structures. The main question is how can we keep track of the current frame of reference. Specifically, we should always be in a position to determine where to draw a line on the screen when a user requests a line to be drawn after a series of translation, scaling and other commands. This is discussed in Section 18.1.

2. User Interface Issues. What are the different ways in which the basic functionality relating to translation/scaling etc. be offered to the user? Our definition of `trans, scale, rotate, line, point` is only one possible interface. There could be several other possible ways. This is discussed in Section 18.2.

18.1 Algorithmic Issues

The surface on which the drawing appears will be called the *screen*. In the case of Scheme, it is a window that appears on the monitor in addition to the shell window. The screen is

associated with a frame of reference; in case of the Scheme graphics screen, the origin is at the center of the screen and the four corners of the screen have coordinates ± 1. The commands `graphics-draw-point` and `graphics-draw-line` take as arguments the coordinates in the screen frame of reference.

The frame of reference in which the user operates is initially the same as the screen frame. However, by using translate, scale and rotate the user can define new frames of reference and draw with respect to these frames. If the user draws a point (x, y) in a certain frame of reference, then where it will be drawn on the screen is determined by the frame changes the user has executed before. In the rest of this section, we will only consider the question of plotting points. Drawing of lines and more complex figures can be done in an analogous manner.

The basic idea is to use a matrix F to represent the current frame of reference. F is a 3×3 matrix with a special structure ($f_{31} = f_{32} = 0$, $f_3 3 = 1$), and given a point having coordinates (x, y) in the current frame, its coordinates (x',y') in the screen frame are obtained by performing a matrix multiplication with F:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Initially, F is simply the identity matrix. Clearly, in this case the matrix multiplication above gives $x' = x$ and $y' = y$ which is exactly what we want.

Suppose we now want have a frame which is centered at (dx, dy) in the screen frame, i.e. obtained by doing a translation of (dx, dy). A point (x, y) in such a frame must be plotted at point $(x + dx, y + dy)$ on the screen. The effect of this frame is modelled if we set

$$F = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix}$$

Now $F(x, y, 1)^T = (x + dx, y + dy, 1)^T$ as we want. Scaling can be represented by setting:

$$F = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Composition of frame changes The important feature is that frame changes can be composed very easily. Suppose, F_1 is a matrix that models a certain translation, and F_2 models some scaling. Then the matrix modelling the translation followed by the scale change is simply the matrix product $F_1 F_2$. In fact this is true for any number of transformations of any kind: the new matrix is obtained from the old simply by performing a matrix multiplication corresponding to the new transformation.

18.2 User Interface

There are a number of different ways in which graphics facilities can be provided to the user. The set of primitives we have provided is only one such way.

Here, for example, is a completely different way in which graphics facilities could be given. Instead of having the notion of "the current frame of reference" in which points and lines are drawn, one could consider the notion of a "current pen position", with respect to which all drawing is done. For example, the command `(point)` would be invoked without giving any arguments and it would mean that the point would have to be drawn at the current pen position. `(line x y)` would mean draw a line from the current position to the point (x,y). We would have to provide a command `(move x y)` which would cause the pen to move to (x,y) without drawing.

In the rest of this chapter, we will discuss 2 user interfaces which are similar to the one we have used in this book. These are simpler to implement. After discussing these we will discuss how the our actual interface is implemented.

In what follows we will only discuss translation and plotting points; other transformations (scaling, rotation) and drawing other figures (lines) are similar.

We will also assume that we are given functions `matrixmultiply` and `matrixvectormultiply` that do these operations. Finally we will use the following definition for constructing a matrix representing transformations:

```
(define (trans-matrix dx dy)
  `((1 0 ,dx)
    (0 1 ,dy)
    (0 0   1))))
```

18.2.1 User Interface 1

We provide the following interface to users:

1. `(trans!  dx dy)`: This causes the reference frame to be translated by (dx,dy).
2. `(point x y)` This asks a point to be plotted at (x,y) of the current frame of reference. The command for drawing lines is similar.

Implementation: We maintain a global variable F which represents the transformation matrix corresponding to the current frame of reference. The `trans!` command is implemented simply by multiplying F by a suitable matrix as defined above. Similarly we define `scale!, rotate!`. The implementation of `trans!` is very simple:

```
(define (trans! dx dy)
  (let ((ftrans (trans-matrix dx dy)))
    (set! F (matrixmultiply F ftrans))))
```

Given that the current matrix is always available, the point command only needs to perform the required matrix multiplication. This is done as follows:

```
(define (point x y)
   (let ((screencoord (matrixvectormultiply F (list x y 1))))
         ;; screencoord is the result vector represented as a list
     (graphics-draw-point (car screencoord) (cadr screencoord))))
```

Criticism: This interface is not very convenient for users. Typically a user will want to write "Draw picture P_1 with origin at (x_1, y_1), then picture P_2 with origin at (x_2, y_2), ...", where the points (x_i, y_i) are all with respect to the current frame. These actions would be represented using above primitives as:

```
(trans! x1 y1) (Picture1) (trans! (- x1) (- y1))
(trans! x2 y2) (Picture1) (trans! (- x2) (- y2))
...
```

The reverse translations `(trans! (- xi) (- yi))` are necessary to bring the frame back to original to get ready to draw the next pictures. This code is more verbose than what our actual interface needs:

```
(trans x1 y1 (Picture1))
(trans x2 y2 (Picture2))
...
```

Not requiring the users to write the code for reversing the translations improves compactness, and further reduces the chances of errors (say the user forgets to write the reversals).

18.2.2 User Interface 2

The problem we need to solve is: how to suppress the evaluation of some of the arguments being passed to a procedure. Lambda expressions can easily accomplish this, i.e. lambda expressions can be used to delay evaluation.

We do require a different interface, but it is almost the same as our ideal interface:

1. `(lambdatrans dx dy g)`: Here `g` is a function that contains the drawing commands, dx and dy are numbers. This primitive causes the drawing commands to be executed after translating the origin by (dx,dy). As an example, to plot a point at (0.1,0.1) after translating the origin to (0.5,0.5) we would write `(lambdatrans 0.1 0.1 (lambda () (point 0.5 0.5)))`.

2. `(point x y)`: This is the same as before.

The crucial difference between `trans` and `lambdatrans` is that the arguments to former are the drawing commands themselves, while to the latter, they must be contained in a function which is passed as an argument. Here is how lambdatrans is implemented:

```
(define (lambdatrans dx dy g)
   (set! F (matmul F (trans-matrix dx dy)))
   (g)
   (set! F (matmul F (trans-matrix (- dx) (- dy))))
   )
```

Notice that this implementation correctly does what we want. When we call the function using say `(lambdatrans 0.1 0.1 (lambda () (point 0.5 0.5)))`, the arguments are indeed evaluated – but the result of evaluating the lambda expression is a function – which is what get passed. The passed function is actually called only inside lambdatrans itself – and this happens after the matrix F has been changed to reflect the translation.

18.2.3 User Interface 3

The `trans` command is actually implemented as a macro. The implementation of the other commands is similar.

```
(define-global-macro (trans dx dy . body)
   (let ((forwardtrans (list 'trans! dx dy))
         (one-line-body (cons 'begin body))
         (backwardtrans (list 'trans! (list '- dx) (list '- dy))))
     (list 'begin forwardtrans one-line-body backwardtrans)
     ))
```

Now suppose we invoke the above with `(trans 0.5 0.5 (line 0 0 0.1 0.1) (line 0.1 0.1 0.1 0.2))`, the code that will result from the above will be:

```
'(begin (trans! 0.5 0.5)
        (begin (line 0 0 0.1 0.1) (line 0.1 0.1 0.1 0.2))
        (trans! (- 0.5) (- 0.5)))
```

which is exactly what we want.

We give another definition using the notation discussed in the previous chapter.

```
(define-global-macro (trans dx dy . body)
   `(begin (trans! ,dx ,dy)
           ,@body
           (trans! (- ,dx) (- ,dy))))
```

Now suppose we invoke the above with `(trans 0.5 0.5 (line 0 0 0.1 0.1) (line 0.1 0.1 0.1 0.2))`, the code that will result from the above will be:

```
'(begin (trans! 0.5 0.5)
        (line 0 0 0.1 0.1) (line 0.1 0.1 0.1 0.2)
        (trans! (- 0.5) (- 0.5)))
```

which is also what we want.

Exercises

1. The implementation of `trans` given above uses two matrix multiplications – the first for the forward translation and the second for the backward. To save arithmetic and also roundoff errors, it will be better to save the original value (say by generating a `let` statement). Write a macro that does this.

2. Currently none of the graphics commands return any interesting value. Suppose you would like to return a description of the bounding box generated for the drawing done inside each command.[1] Adapt all the drawing commands so that the user can do this easily. Show your modifications for `trans, line, point`, and explain what the user will have to do in addition, if anything.

[1] The bounding box is the smallest axis parallel rectangle containing the drawing, and it can be represented by its bottom left and top right corners, i.e. 4 numbers.

Chapter 19

Object Oriented Programming

A computer program is typically developed for modelling some real life phenomenon or a mathematical process. The real life phenomenon or the mathematical process concerns some objects which may be concrete or abstract, e.g. circuit components, salary records, matrices. The goal is to trace how objects evolve as certain operations (e.g. changing the value of the signal fed to the input of a component, or calculating the effect of income tax, or inverting a matrix) are performed upon them. These objects are associated with many kinds of data (e.g. value of the voltage at the input of a gate, or the x-component of the position of an electron, the house rent allowance), or other objects themselves. How to represent and manage the objects constituting a program is a central question in programming. It may be said that this question forms the starting point for the development of the Objected Oriented Programming Paradigm.

Note that *representation of an object* does include a description of its state, e.g. the x-component of the position of an electron, but needs to include a representation of its *behaviour* as well, e.g. what happens to the output if the voltage at the input of a gate changes. For computation, we need the state as well as the behaviour.

As with other paradigms, we will only discuss some of the important features of OOP. We start with the simplest features and move to the more complex ones.

19.1 Aggregation

The most elementary facility needed in managing data is that of *aggregation*. For example, in day-to-day conversation, we refer to a complex number as a single entity, even though we know that it consists of two parts, say a real part and an imaginary part. Likewise, it is most convenient if in our programs we can refer to a complex number as a single entity. This obvious convenience is supported in most modern programming languages, e.g. using the derived types in Fortran 90, or the structures in C. Scheme also provides a mechanism to group together data into an entity also called a `structure`. It is worth noting, however, that older languages, eg. Fortran IV, did not have this facility. In Fortran IV, you would be forced to talk about a complex number using two variables.[1]

[1]This problem is, of course, much worse. Suppose your program concerns particles which might have a mass, a charge, a position, a velocity. In Fortran 90/Scheme/C you can group all these things together into a single object, which has *fields* for the mass, charge, x, y, z positions and velocities. In Fortran IV you

In Scheme, here is how you might define a structure `complex` having two fields `real` and `imaginary`:

```
(define-structure complex real imaginary)
```

Given the definition, the following code would set `a` and `b` to the complex numbers $3 + 2i$ and $5 - 4i$.

```
(define a (make-complex 3 2))
(define b (make-complex 5 -4))
```

In general, given the name of a structure, by prefixing `make-` to it you get a procedure called the *constructor*. The constructor when called with the values for the fields returns the structure object. Given a structure, we can examine its fields. For example, we may write

```
(define c (complex-real a))
(define d (complex-imaginary b))
```

After this `c` would be set to 3, and `d` to -4. Once we define a structure say `s`, then Scheme itself defines procedures `s-f` for accessing its field `f`, for every `f`. Functions of the type `s-f` are called accessor functions.

Once we define a structure for representing complex numbers, it seems qnatural to define addition etc. for complex numbers. For example:

```
(define (complex-add p q)
  (make-complex (+ (complex-real p) (complex-real q))
                (+ (complex-imaginary p) (complex-imaginary q))))
```

So if we now write

```
(define f (complex-add a b))
```

we would have `f` set to a structure with `real` field 8 and `imaginary` field -2, i.e. representing the number $8 - 2i$ which is what we want.

19.2 Data Abstraction

Once we define the structure for representing complex numbers and write the functions such as `complex-add` which can manipulate complex numbers, we can pretend that complex numbers are a part of our programming language itself. Indeed, we could take our definition and associated functions, put them in a file, and give them to another user who could also pretend that his Scheme language directly supports complex numbers. While writing programs effectively we could then think of complex numbers as being just another data type that Scheme supports, along with the more standard data types like integers (on which you can perform addition etc.) or lists (on which you can perform operations such as `car` or `cdr` etc.) In fact, this strategy of defining a structure and functions that model a certain

would need to refer to the 8 variables separately. For this reason, in Fortran IV code, you would see very long argument lists to subroutines.

entity (say complex numbers) is very useful. The structure + functions together are said to constitute an *Abstract Data Type*.

Building abstract data types is very useful. Suppose we are to write an electric circuit simulator. The entities in this simulator might be gates and wires. While writing the simulator it might be most convenient to first build and abstract data type for representing a wire and another for representing a gate. For each distinct entity, say a gate, we could define a structure to store the relevant data, and then write code for the operations/processes in which a gate is involved. Once we have done that, we can then pretend that Scheme supports gates!

Dangers with using structures: programmers may accidentally modify field values. It could be said that we could make it a convention not to do so – however, this is not the same thing.

19.3 Function overloading

Represent complex numbers in a polar form. We still want to do arithmetic. Can a user do so without using new names?

Solution: Scheme allows you to check the type of a structure, so make changes to the add function : it now checks the input type and does a conversion if necessary.

Works but is ugly. Modularity is lost.

19.4 Code evolution

These all deal with extending existing software.

19.4.1 Example 1: Library information system

Suppose a library information system currently supports the following operations.

```
(make-patron name address)
                ; Enroll a new patorn.
(issue-book patron-record bookno)
                ; Issues a book to the patron with the given record.
(accept-book-return patron-record book)
                ; Removes the book from the record.
(print-ith-issued patron-record i)
                ; returns the booknumber of the ith book issued to this patron.
```

Suppose the library starts a category of donor-patrons, who might get more facilities. How should the information system change?

- Need to store patron category along with each patron.
- Issue-book may change – donors may be able to borrow more books.

Challenge: Can new features be added without changing old code at all.

19.5 Example 2: Extensions to Deriv

Add the operator **, or ln or tan.

Can this be done without modifying the existing code at all?

19.6 Basics of OOP

In functional programming, a program may be thought of as a map (i.e. a function) from the set of all possible inputs to the set of desired outputs. Likewise, it is useful to view programming in a certain manner to understand the object oriented paradigm.

In this view, a program is developed for modelling some real life phenomenon or a mathematical process. The real life phenomenon or the mathematical process concerns some objects which may be concrete or abstract, e.g. circuit components, salary records, matrices. The goal is to trace how objects evolve as certain operations (e.g. changing the value of the signal fed to the input of a component, or calculating the effect of income tax, or inverting a matrix) are performed upon them. How to represent these objects on a computer and how to describe the operations that can be performed upon them, are the central (interrelated) questions in OOP.

The term *object* as used in Object Oriented programming, refers to the computer representation of mathematical or real life objects. Each object (in the OOP sense, from now on) can be thought of as containing data as well as code. For example, the object representing a circuit component will contain voltage values associated with the component. But in addition, the object will also be associated with code that describes how the object behaves under different circumstances. For example, each object representing a circuit component would have to be associated with code which defines how the component behaves when the input voltage is changed. As another example, consider an object which represents a matrix – such an object could be associated with code that inverts matrices. The code associated with objects is organized as *methods*; for example a circuit component object could have a `compute-next-state` method, and a matrix object could have an `invert` method associated, and a salary record object could have an `income-tax` method. Methods are like procedures, i.e. they can be called with arguments and they return values, and further more one method of one object can call another method of another or the same object.

19.6.1 Classes

First of all, all the different objects constituting a program need not be individually described. For example, a circuit could be made up of several AND gates, or several distinct matrices might be used in a program. Each object representing a gate would have to have different data stored locally, however the code associated with the the different objects representing different AND gates could be identical. Such a collection of objects is said to belong to the same *class*. Alternatively, we say that the objects are all *instances* of the same class.

OOP provides the ability to describe classes. The description of a class includes the names of the local variables that will appear inside each object that is an instance of the class, and also the code that will be associated with the instances.

19.6.2 Inheritance

Several classes might themselves share features. In the example stated earlier, a donor and a patron are objects for whom book issue is different; however other properties (e.g. both need to have a name and an address) are the same. OOP provides the notion of *inheritance:* which is simply the ability to define one class as a special version or a *subclass* of a given class. If class A is a subclass of class B, then class B is said to be the superclass of class A. For example, using OOP we can say that donor is a subclass of patron, and then go on to describe how donor differs from patron. Except for the differences described, the donor class behaves identically as patron. The OOP terminology for this is that a subclass *inherits* its properties, i.e. names of local variables and the methods from its superclass. The subclass can be made to behave differently from its superclass by defining new properties for it, or by *redefining* the inherited properties.

19.6.3 Defining Classes

We describe OOP primitives embedded into Scheme. This is based on the work of Bryan O Sullivan. More commonly, OOP is supported by languages such as C++, Java, Smalltalk and others.

A class definition has the following form:

```
(define-class class-name
   (superclass)
   list-of-additional-fields)
```

In this book, class names will have enclosing angled brackets as a convention; this is only for readability. Here are two examples of class definitions:

```
(define-class <patron>
   (<root>)
   (name address books))

(define-class <donor>
   (<patron>)
   (locker-number))
```

A class is always defined as a subclass of some class. To enable this process to get started, Scheme OOP contains the definition of a predefined class `<root>` from which other classes can be derived. The `<root>` class does not have any local variables (also called *fields* or *slots*) associated with it; nor does it have any methods.

As mentioned earlier, when a class is defined as a subclass of another class, all the properites (i.e. fields and methods) are inherited. The class definition mentions the additions field that objects of the class are to have. Thus class `<patron>` is defined to have fields `name, address, books` in addition to whatever fields that `<root>` has. But there is nothing in `<root>`, so the definition causes `<patron>` to have just the fields `name, address, books`. The class `<donor>` is defined to be a subclass of `<patron>` and is deemed to have an additional field `locker-number`. Thus the fields in `<donor>` will be `name, address, books, locker-number`.

19.6.4 Creating Instances

Given the above definitions we can create instances using the `make-object` form having the following syntax.

```
(make-object class-name initial-values-of-fields)
```

This creates an instance of the specified class, with field values as specified. The initial values should be specified in the same order that appears in the definition of the class and its superclasses. Here are some examples:

```
(define x (make-object <patron> "Ranade" "IITB" '()))
(define y (make-object <donor>  "Dhamdhere" "IITB" '() 569))
```

This will cause `x` to be an object of class `<patron>` with its fields `name, address, books` set to values ``Ranade'', ``IITB'' and the empty list respectively. `y` will be set to be an object of class `<donor>` with its fields `name, address, books, locker-number` set to values ``Ranade'', ``IITB'', the empty list and the number 569 respectively.

Simplified Constructors

The `make-object` command is often somewhat cumbersome to use. For example, when making a `<patron>`, it is cumbersome to specify the value `'()` for the `books` field. To avoid this it is customary to define *simplified constructors* as follows.

```
(define (make-patron name address)
   (make-object <patron> name address '())
)
```

Now it suffices to say

```
(define x (make-patron "Ranade" "IITB"))
```

Such a simplified constructor will also be useful for the `<donor>` class; and in fact with most classes.

19.6.5 Defining Methods

Methods must be declared, and then defined for different classes, then they may be invoked on objects of those classes. The general syntax is:

```
(define-generic name-of-the-method)  ;; declaration
(define-method method-name class-on-which-to-be-defined arguments
   code)                             ;; how to define
(method-name object-name arguments)  ;; how to invoke
```

We will start with an example for the `<patron>` class defined above.

```
(define-generic issue-book)

(define-method issue-book <patron> (bookno)
  (cond ((< (length (self:  books)) 4)
         (self:set! books
             (cons bookno (self: books)))
         #t)
        (else '())))

(issue-book x 123)    ;; invocation
```

In the above code, assume `x` is as defined earlier, i.e. instances of `<patron>`.

In any method invocation of any method, the first argument must be the object on which the method is to be invoked. The subsequent arguments are bound to the arguments in the arguments listed in the method-definition. In the above example, 123 is thus bound to `bookno`. The code in the method definition can be any scheme code, and it must return a value which is the value of the method invocation. However, in addition to ordinary scheme code, two OOP commands can also be used. These are for accessing and modifying the local variables of the object, respectively:

```
(self: local-variable-name)
(self:set! local-variable-name new-value)
:self:
```

Thus `(self:  books)` in the code above would refer to the `books` field of `x` for the invocation `(issue-book x 123)`; while `(self:set!  books ...)` would set the same field as specified. Within a method, the object on which is the method is being executed can itself be referred to as `:self:`.

Encapsulation: Local variables, or the field in an object cannot be accessed/modified directly. They can only be accessed/modified inside methods defined for the class. The rational for this is *encapsulation*, i.e. the local variables are hidden from the external world, and can be accessed by users only in a controlled manner, through the methods.

Inheritance: Although `issue-book` is explicitly defined only for class `<patron>` it will apply to class `<donor>` as well due to inheritance. So we could write `(issue-book y 456)` – this would be legal; but the above code would be used and it would check for the lending limit of 4 for both `x` and `y`.

However, we can redefine the `issue-book` method for the subclass `<donor>` as follows:

```
(define-method issue-book <donor> (bookno)
  (self:set! books
             (cons bookno (self: books)))
  #t)

(issue-book x 123)    ; at most 4 books
(issue-book y 345)    ; unconditional issue
```

The general logic that is used to decide which code is to be used when a method is invoked on a certain object `z` of class `Z` is as follows. If the method has been defined explicitly for `Z`,

then that definition is used. If the method is not defined for `Z`, then the OOP system checks if the method is defined in the immediate superclass of `Z`. If the superclass has the method, then that definition is used; if not, the superclass of the superclass is checked, and so on. If neither `Z` nor any of the superclasses has a definition for the method, then the system signals an error.

19.6.6 Remark: Incremental specification of functions

In conventional programming languages, a function definition is a single define form:

```
(define (issue-book patron book) ...)
```

In OOP, the function definition may be scattered.

```
(define-method issue-book <patron> (bookno)
 ...issue only if limit not exceeded...)

(define-method issue-book <donor> (bookno)
 ...issue unconditionally...)

(issue-book person1 book5)
```

The define-method forms can be spread out over several files. They dont have to be specified together. It is this flexibility that makes OOP programs easier to extend. In conventional programming, including the special case for donors would require *modification* to the function `issue-book`. While making such modifications, there is always the danger that a bug gets introduced in a part of the code that has already been extensively tested. In OOP, the new code gets added as a separate method, `without the need to alter the existing code in any way`.

This is applicable to the other examples as well. For example, the conventional definition of deriv is:

```
(define (deriv exp) ...)
```

In OOP, we could have different classes to represent different types of symbolic expressions. The deriv method for each such class would be defined separately and independently, as and when the class is added to the code.

```
(define-method deriv <sine-expression> ()
   (cos ...))

(define-method deriv <cos-expression> ()
   (sin ...))
```

19.7 Utility Functions

(class-of object) : returns the class of the object

```
(class-of x) ==> <patron>
```

(object? x) : is x an object?

```
(object? x) ==> #t
```

(is-a? object class) : is object of an instance of class (direct instance), or a subclass of class, or a subclass of subclass of class ... (general instance)?

```
(is-a? y <donor>) ==> #t

(is-a? y <root>)   ==> #t

(is-a? x <donor>) ==> '()
```

19.8 Exercises

1. Add a method to handle the operation `(issue-many patron-object list-of-books)`, which should cause all the specified books to be issued if possible. (Hint: Use `:self:` to repeatedly call the `issue-book` method of the same object. You can even use the Scheme function `map`.)

2. Develop classes for representing symmetric and asymmetric (square) matrices. Write the methods `matrix-ref` and `matrix-set!` for both classes. Also write simplified constructors `make-matrix` and `make-smatrix` which respectively return an ordinary representation and a representation for symmetric matrices.

3. You are to design a *dictionary* program that provides information about nouns and verbs of English. It must support the following queries: (1) (plural w) : If w is a noun return its plural; else return the message "Not a noun". (2) (past w) : If w is a verb then return its past tense, else return the message "Not a verb".

   Design a set of classes which will make it easy to store words. You may assume that English has 3 kinds of nouns: those whose plural is obtained by appending s (e.g. boy, girl), appending es (e.g. bus), or whose plural is irregular (e.g. ox, life). You may similarly assume that there are only two kinds of verbs, those whose past tense is obtained by adding ed (e.g. play, jump) or irregular (e.g. study, run, go). Give the code that builds a dictionary storing boy, ox (plural oxen), jump, go. It should be easy to add new words.

   You may want to use the symbol-append function which concatenates symbols, e.g. `(symbol-append 'boy 's) => 'boys`

Chapter 20

Object Oriented Design

To write an OO program to solve a certain problem, we need to decide what classes and methods to use. We also need to determine a proper inheritance heirarchy i.e. which class should inherit from which class. Often there is no unique answer to these questions; on the other hand it is possible to identify some general principles.

This chapter has 2 case studies.

20.1 A Biology Tutor

We start with an example problem. Suppose we need to develop a *biology-tutor program* for answering questions about animals. The program is expected to take questions such as (no-of-limbs cat) and return the appropriate response, i.e. 4 in this case. The following questions are to be supported: (1) no-of-legs (2) colour-of-blood (3) blood-temperature, (4) life-span. The following animals should be in the database: cat, spider, housefly, cockroach, chimpanzee. The lifespans of these animals may be assumed to be 5, .1, .05, .1, 30 years respectively. The answers to the other questions should be given as per the following passage.

> The animal kingdom may be divided into vertebrates and non-vertebrates. Vertebrates have red blood while non vertebrates have colourless blood. Within vertebrates, mammal (e.g. gorilla, cat) and bird families are warm blooded, while non-vertebrates and other vertebrate families such as reptiles, amphibians and fish are cold-blooded. Within non-vertebrates are the families insecta (e.g. housefly, cockroach) and arachnida (e.g. spiders). Members of these families have respectively 6 and 8 limbs. Mammals have 4 limbs.

How do we write the program? How do we ensure that the program is convenient to extend?

20.1.1 Design 1

We define an `<animal>` class in which we store the data for each animal.

```
(define-class <animal> (<root>) (legs blood-color blood-temp life))
(define-generic no-of-legs)
(define-generic color-of-blood)
```

```
(define-generic blood-temperature)
(define-generic life-span)
(define-method no-of-legs <animal> () (self: legs))
(define-method color-of-blood <animal> () (self: blood-color))
(define-method blood-temperature <animal> () (self: blood-temperature))
(define-method life-span <animal> () (self: life))
(define (make-animal legs bc bt ls)
  (make-object <animal> legs bc bt ls))
```

To insert the data for an animal, we could type for example:

```
(define cat (make-animal 4 'red 'warm 5))
```

Given this definition, we could type `(blood-temperature cat)` and get the answer `'warm`.

20.1.2 Design 2

While the previous design is functionally correct, it does not make use of the known biological facts. As a result, the task of the person entering the data is cumbersome. For example, while entering the data for `cat`, it should be enough to state that it is a mammal and that it has a life span of 5 years. *All other attributes such as the blood color, blood temperature, and the number of legs are determined once it is specified that a* `cat` *is a mammal.* Thus to insert information regarding a cat it should be enough to type `(define cat (make-object <mammal> 5))` or something similar.

We can in fact accomplish this. Our code however becomes a bit longer.

```
(define-class <animal> (<root>) (life-span))
(define-class <vertebrate> (<animal>) ())
(define-class <invertebrate> (<animal>) ())
(define-class <mammal> (<vertebrate>) ())
(define-class <bird> (<vertebrate>) ())
(define-class <rest> (<vertebrate>) ())
(define-class <insect> (<invertebrate>) ())
(define-class <arachnida> (<invertebrate>) ())

(define-generic no-of-legs)
(define-generic color-of-blood)
(define-generic blood-temperature)
(define-generic life-span)

(define-method life-span <animal> () (self: life-span))

(define-method color-of-blood <vertebrate> () 'red)
(define-method color-blood <invertebrate> () 'colorless)

(define-method blood-temperature <animal> () 'cold)
(define-method blood-temperature <mammal> () 'warm)
```

```
(define-method blood-temperature <bird> () 'warm)

(define-method no-of-limbs <vertebrate> () 4)
(define-method no-of-limbs <insect> () 6)
(define-method no-of-limb <arachnida> () 8)
```

Now in fact it is enough to type `(define cat (make-object <mammal> 5))` to insert a cat into our database. Suppose we now invoke `(color-of-blood cat)`. Let us trace through how Scheme executes this call. As we remarked in the previous chapter, since `color-of-blood` is a generic method, the precise code to execute depends upon the class of the argument to it, i.e. `cat`. The class of `cat` was defined as a `<mammal>`. So Scheme checks if a `color-of-blood` method is defined for this class. As you can see, our code does not define such a method. So Scheme looks in the superclass (i.e. `<vertebrate>`) of `<mammal>`. The method has indeed been defined in the class `<vertebrate>`, so Scheme executes that code, which simply returns `'red`.

20.1.3 General Principles

The second design is not only more convenient from the point of entering data, but is also remarkable in many other ways.

First, notice that the object for each animal contains only one field: `life-span`. This doesnt mean that each animal cannot be associated with other data attributes; only that the data attributes need not be stored separately for each object. In fact, all attributes except for `life-span` are not stored in our objects at all but are part of the methods. This is another important idea: data that is common to all objects in a class is more compactly represented by defining a method.

Second, methods can be defined at different levels of the heirarchy. For example, instead of defining `<color-of-blood>` at the level of `<vertebrate>`, we could have defined it separately for all the subclasses of `<vertebrate>`, i.e. by writing

```
(define-method color-of-blood <mammal> 'red)
(define-method color-of-blood <bird> 'red)
```

instead of writing

```
(define-method color-of-blood <vertebrate> 'red)
```

Clearly, defining `color-of-blood` once for the entire class `<vertebrate>` is better.

Third, do note that the OOP framework allows methods to be *over-ridden*. We have used this for `blood-temperature`. Since most animals are cold-blooded, it is more compact to specify this as a general rule `(define-method <animal> blood-temperature () 'cold)` and then define the exceptions for `<mammal>` and `<bird>`.

Summary

If all instances share the same constant value for a local variable then use a method that simply returns that fixed value, rather than keeping a variable in every object.

If a field or a method is useful for all the subclasses of a class A then it should be defined in A. In fact a method should be defined in A even if it is useful for most of the subclasses; if subclasses B, C need to define the method differently then it can always be redefined in B and C.

20.2 Simulation

These days, almost no commercial product is manufactured without first making a mathematical model of it, feeding the model to a computer, and using the computer to determine whether the product will indeed behave satisfactorily in practice. This entire exercise is called *computer simulation*. Indeed, computer simulation is used while designing cars, computers, buildings, bridges, and even nuclear weapons.

To develop a simulator the following must be done:

1. Determine the objects that constitute the system to be simulated. For example, an electrical circuit consists of transistors, resistors, voltage sources, etc. These are the objects that need to be represented while developing a simulator for electrical circuits.

2. Define a class to model each type of component. The class will hold data associated with the component. For example, a transistor object will be associated with different voltages (e.g. the voltage at its gate).

   : Different basic components can be defined as classes , with appropriate local variables which signify the component; and appropriate functions characterising its behaviour

3. Some mechanism for connecting components together : A way of representing the connections or implementing them. Generic functions provide a key to this.

4. Building new componenets (more complex) : New classes con be created inheriting the features of basic ones. Additional features can be characterised using additional variables and functions.

SIMULATION DRIVER

Having determined a way of representing a circuit and configuring it with multiple level of components(sub-components etc.); one wants to test-run circuit " Run this system for one time step" Hence one can probe contents of objects.

SIMULATING LOGIC CIRCUITS

Component class library

Basic components required for implementing a circuit : Various gates like AND, OR, NOT ,XOR gates,etc.

LOGIC VALUES

1. Representing values in binary manner - going digital from analog
2. Only two values - True or False
3. A possible representation - 5 volts = T , 0 volts = F

TRUTH TABLES

For example - An XOR gate with inputs X and Y. Output is true only when exactly one of them is true.

X	Y	OUT
T	T	F
T	F	T
F	T	T
F	F	F

GATE DELAY

Time after which output will take the promised value Saying Delay = 1 would meanthe output comes from the gate after one unit of time .

NOTE

NEVER Short two outputs.

PIN

To be simplest, it will denote an inert wire. It causes no changes, but is ameans of providing input or getting output.

GATE

It will have pins as its members - several input pins may be there but only one output pin is permissible.

COMBINATIONAL CIRCUITS

A circuit with no feedback (an output becomes input for next cycle of operation) no memory (an output can be accessed even after another operation) is called "Combinational Circuit".

Operations on Pins

1. Read - Obtaining the output
2. Write - Providing the input

Operations on Gates

1. Tick - Read values on input pins and compute the value to be written after the next time step i.e gate delay.
2. Write - Take value computed in the previous step and write it to the output pin.

Basic Step of Simulation

1. Tick all gates compute the state for next time instance
2. Write all gates update pin values

20.3 Exercises

1. The goal of this exercise is to develop a simulation of an Economic System so as to explore the effects of *globalization*. Real economists will of course cringe at our model of an economic system – it is doubtless extremely simplicstic – but then we wont show this to real economists. Or we can show it to them and ask them to help us improve it.

   Basically our system will consist of certain number of buyers and sellers. Each buyer or seller will be located in a *village*. Several villages will comprise a *taluka*, and several talukas a *district*. Our model will be just for one district.

   For concreteness, let us assume that our district consists of 8 talukas, each of which consists of 8 villages. Each village has 4 buyers and one seller. Each seller S will start off with a working capital of Rs. 10, and will have to pay a rent of Re. 1 every day. There is a single commodity being traded, say chapatis. Each seller will produce as many chapatis as there is demand. The cost of each chapati to each seller S will be a fixed number C(S). For each seller S, we will fix C(S) as a random number[1] uniformly distributed in the interval (rupees) 0.9 to 1.1. C(S) will be fixed for each S for the entire course of the simulation. Who sells what to whom gets decided every day separately as follows.

[1] `(random x)` will return a random number in the interval 0 to x (non-inclusive). The number returned is real if x is real, and integer if x is integer.

Each day starts off with the each seller announcing the price at which he will sell chapatis. Each buyer needs one chapati every day, however he can buy this from any seller. If the chapati is bought from the seller in the same village, then there is no overhead. If it is bought from a seller in the same taluka, then there is an overhead of 10 paise. If it is bought from another taluka, there is an overhead of 20 paise. Each buyer buys his chapati such that price+overhead is smallest. The sellers collect the money (the overhead goes for transport) from the buyers and pay the rent. The daily profit/loss gets added/subtracted from the working capital. If the working capital falls below zero, the seller goes out of business.

Here is how the seller fixes his price. Define the BE4 price for a seller S as C(S)+0.25, i.e. the price at which he will break even (i.e. no profit no loss including the rent) if he sells 4 chapatis. On day 1, every seller fixes his price at his BE4 price. On subsequent days his rule is very simple: he raises his price by 10 % if he made did not make a loss the previous day, else he lowers the price by 10 %, but never below his BE4 price.

(a) Evolve the system for 30 days and report the total receipts of the sellers and the total money paid out by the buyers.

(b) Incorporate the effect of increased globalization as follows. Set to 0 the overhead for buying from outside your village or your taluka. Now evolve the system for 30 days.

2. The goal of this exercise is to design a program that will be able to draw faces on the screen. Faces can be round, tall, wide. Noses can have different shapes, let us call them ordinary, and parrot-beak-shaped. Eyes can have different colours, and can either be set close or wide (i.e. distance between the eyes can be small or large) and can have different colours. The mouth can be large or small. This list can go on, but let us stop here.

   How will you design a program that can take user requirements and draw a face? In particular, you should provide facilities for the user to say something like "Draw a face which is round, with blue eyes that are set close, and a parrot-beak nose". Say what classes you will use, including the heirarchy. What will the methods and the local variables be? How will a user specify the face that is to be drawn?

   A high level answer is expected with some details. You could spend the entire hour writing out an answer to this problem – but remember it only has 20 marks; so spend only 20 minutes on it. More only if you have finished solving the other problems.

3. Write a set of classes that will help in doing animations. This is an open ended problem; however here we will consider a simple version. Create a system which allows to (i) define shapes, (ii) Assign velocities and initial positions to the shapes (iii) Show the evolution of the system as time goes on.

   Specifically you should build a "screensaver" showing 2-3 kinds of aeroplanes (i.e. polygonal shapes) objects moving across the screen. The aeroplanes should disappear and reappear so as to create a pleasing, natural (to the extent possible!) effect.

Chapter 21

Constraint Logic Programming (CLP)

It is useful to think of programming as consisting of two activities

1. Writing down the specifications precisely and exhaustively. Identifying what is the input and what is the output and specifying what are the *constraints* that the output must satisfy for correctness.

2. Writing a program that constructs the required output.

In the Constraint Logic Programming paradigm, we only do the first step! A programmer writes down all the constraints that the output must satisfy, and then a *constraint solver* which is at the heart of the CLP system processes the constraints and attempts to find the output values that satisfy the given constraints.

Consider the problem of constructing a 3×3 magic square using the digits 1 through 9 exactly once. A magic square is simply a square array of numbers such that all row sums and column sums are identical. Suppose $a, b, c, d, e, f, g, h, i$ are the numbers in the array as shown in Figure 21.1. Here is how we could then specify this problem.

Find $a, b, c, d, e, f, g, h, i$ such that

1. $1 \leq a, b, c, d, e, f, g, h, i \leq 9$
2. $a, b, c, d, e, f, g, h, i$ are distinct integers.
3. $a + b + c = d + e + f = g + h + i = a + d + g = b + e + h = c + f + i$.

It is clear that any assignment of values satisfying the above constraints will give a magic square. In the CLP paradigm, we would simply assert the above constraints (using suitable syntax) and ask the system to find us a solution.

Will a solution always be found if it exists? Will a message be printed if no solution exists? Suffice to say for now, that a whole range of solvers are available, and they work by

a	b	c
d	e	f
g	h	i

Figure 21.1: Magic Square

using very sophisticated ideas as well as very obvious ones (e.g. try all possible assignments until the right one is found). How quickly a solver completes its work (or even whether it completes in finite time at all) depends upon the problem being solved as well as the sophistication of the solver. In fact it depends also upon how the problem is specified[1]

It is safe to say that the time taken by the CLP solver (which is a general purpose program) will in general be more than a program written precisely to implement the given specifications. However, writing the specifications alone is always much easier, and as a result CLP systems are becoming very popular for solving certain kinds of problems. In fact many CLP systems are available commercially and in the public domain. These include CHIP, ILOG, ECLIPSE

21.1 Finite Domain CLP

In this book we will consider a relatively simple variant of CLP called finite domain CLP. This basically means that our problem must be specified using a finite number of variables, and that each variable can take values from a finite domain. We will call such variables *domain variables*. The magic square problem belongs to this category with 9 domain variables, each variable taking values from the finite domain 1,2,3,4,5,6,7,8,9.

Our CLP system is embedded in Scheme. It contains commands for constructing domain variables, specifying constraints, and a very simple solver. Note that a CLP program in our system may freely use scheme commands and additional Scheme variables as needed. Scheme variables behave in the usual manner, i.e. their value is explicitly set by the program. Domain variables, which must be explicitly constructed will have their values set by the Constraint Solver when it is called.

A CLP program is in 4 parts:

1. Initialization of the CLP solver.
2. Definitions of domain variables.
3. Specification of constraints.
4. Invocation of the CLP solver.

These 4 parts are discussed below. We will use two examples to illustrate the ideas. The first is a trivial problem described only because it is exceedingly simple:

> **Quadratic Equation Problem:** Find positive integers x, y such that $x + y = 7$ and $xy = 12$.

The second problem is the magic square problem described earlier.

[1]For the magic square, we could note that each of the rows/columns must sum to 15 since the numbers 1 through 9 which must all appear sum to 45. Thus, we could have specified the third condition as $15 = a + b + c = \ldots$. This will almost certainly ease the work for the solver, leading to a solution faster.

21.2 Initializing the CLP Solver

This is done by the command:

```
(clp-init)
```

This performs some of the book-keeping functions needed by the solver. If the command is issued in the middle of a session, it causes the solver to start afresh, discarding the constraints that might have been added earlier. This plays a similar role as `gr:reset` played in graphics (i.e. clearing the screen, discarding the transformation commands issued till then).

Every CLP program must begin with this command.

21.3 Definition of domain variables

Here is how a domain variable can be constructed

```
(dvar domain-specified-as-a-list)
```

This returns a domain variable. This is simply a scheme object which can be stored in an ordinary scheme variable so that it can be referred to later. For example:

```
(define v1 (dvar '(1 2 3 4 5 6)))

(define v2 (dvar '(red blue green)))

(define v3 (dvar '(10 20 30)))

(define v4 (list (dvar '(1 2 3)) (dvar '(4 5 6))))
```

The above declares that v1 must take an integer value between 1 and 6, v2 must take as value a symbol 'red, 'blue, or 'green, and v3 one of the values 10, 20, 30. v4 is a list of two domain variables; these must respectively take values in [1,3] and [4,6].

21.3.1 Quadratic Equation Problem:

We are given that x, y are positive integers whose sum is 7. So we may conclude that both of them are in the range [1,6]. So we would write:

```
(define x (dvar '(1 2 3 4 5 6)))
(define y (dvar '(1 2 3 4 5 6)))
```

21.3.2 Magic Square Problem:

Each of our variables are known to be positive integers in the range [1,9]. Thus we may write:

```
(define 1to9 '(1 2 3 4 5 6 7 8 9))
(define a (dvar 1to9))
(define b (dvar 1to9))
(define c (dvar 1to9))
(define d (dvar 1to9))
(define e (dvar 1to9))
(define f (dvar 1to9))
(define g (dvar 1to9))
(define h (dvar 1to9))
(define i (dvar 1to9))
```

We could of course have written '(1 2 3 4 5 6 7 8 9) several times instead of defining the variable `1to9`. Note that `1to9` is just an ordinary Scheme variable and not a domain variable.

21.4 Specification of constraints

Any number of constraints may be specified, using the following syntax for each constraint:

```
(add-rule predicate-of-n-arguments dvar1 dvar2 .... dvarn)
```

The predicate must be specified as a Scheme procedure. Remember that a Predicate is a function which returns #t or (). Here are some examples:

```
(add-rule (lambda (x y) (= 15 (+ x y))) v1 v3)
             ;; assuming v1 and v3 are as declared above.
(add-rule (lambda (x y) (not (= x y))) (car v4) (cadr v4))
             ;; assuming v4 is as declared above.
```

The first command says that the values assigned to v1 and v3 must sum to 15. The second says that the domain variables in the list v4 should not take the same value.

21.4.1 Quadratic Equation Problem:

In this case we need to describe the rules for the sum and the product. This may be done as:

```
(add-rule (lambda (p q) (= (* p q) 12)) x y)
(add-rule (lambda (p q) (= (+ p q) 7))  x y)
```

21.4.2 Magic Square Problem:

There are two kinds of rules, the first kind constraining the sums of rows, columns and diagonals, and the second kind asserting that the variables must take distinct values. The first kind are easy to write down:

```
;; row sums
(add-rule (lambda (x y z) (= 15 (+ x y z))) a b c)
(add-rule (lambda (x y z) (= 15 (+ x y z))) d e f)
(add-rule (lambda (x y z) (= 15 (+ x y z))) g h i)

;; column sums
(add-rule (lambda (x y z) (= 15 (+ x y z))) a d g)
(add-rule (lambda (x y z) (= 15 (+ x y z))) b e h)
(add-rule (lambda (x y z) (= 15 (+ x y z))) c f i)

;; diagonal sums
(add-rule (lambda (x y z) (= 15 (+ x y z))) a e i)
(add-rule (lambda (x y z) (= 15 (+ x y z))) c e g)
```

Next we need to enforce the distinctness constraints. Since there are 9 variables that must be distinct, there will be $\binom{9}{2} = 36$ constraints. All these need not be written out separately; we can use a set of nested loops containing one `add-rule` command that is called 36 times.

```
(do ((i 0 (+ i 1))
     (alldvars (list a b c d e f g h i))
     )
    ((= i 8))
  (do ((j (+ i 1) (+ j 1)))
      ((= j 9))
    (add-rule (lambda (x y) (not (= x y)))
              (list-ref alldvars i)
              (list-ref alldvars j))))
```

As will be seen `add-rule` will be called 36 times to assert the distinctness of the ith and jth elements in the list of domain variables `alldvars`, for each of the 36 unordered pairs i, j.

21.5 Invoking the solver

The Solver may be called as follows

```
(clp-solve solution-handler)
```

where `solution-handler` is a procedure with no arguments that gets called whenever the Solver finds a solution. This procedure may print the solutions on the screen (as described below), just count the solutions, or do anything else. The solver will in generall attempt to find all the solutions, and call `solution-handler` everytime this happens. However, `solution-handler` can contain the command `(restart 1)` which if executed stops execution and returns control to the Scheme interpreter.

```
"New solution"
((3) (4))
"New solution"
((4) (3)) -- done
;Value: ()
```

Figure 21.2: Output for the Quadratic Equation Problem

21.5.1 Printing the solutions:

The solutions can be obtained by applying the function `domain` to each domain variable of interest. The function returns a list of values which the domain variable can take in the particular solution found by the solver. Usually, when the solver finds a solution, the list returned will have length 1. We will call such solutions *simple solutions*. Occasionally the solver might group solutions together. Suppose for example that for a certain problem involving domain variables v1 and v3 there are 2 solutions in the first of which v1 has the value 1 and v3 the value 10 and in the second of which v1 has the value 2 and v3 the value 10. In this case the solver might present the two solutions together as a *group solution*. This is done by the solver returning `'(1 2)` in response to `(domain v1)` and `'(10)` in response to `(domain v3)`. In general, a group solution may be thought of as a compact representation of many simple solutions[2].

Note that it is possible that the problem has no solution; in this case the procedure specified alongwith `clp-solve` will not be called at all. So if the solver returns without calling the specified procedure, then it means there are no solutions.

21.5.2 Quadratic Equation Problem:

In this case we could write:

```
(clp-solve (lambda ()
             (pp "New solution")
             (pp (list (domain x) (domain y)))))
```

The above invocation will result in printing out the two solutions as shown in Figure 21.2.

21.5.3 Magic Square Problem:

We could write:

```
(clp-solve (lambda ()
             (pp "New solution")
             (pp (append (domain a) (domain b) (domain c)))
             (pp (append (domain d) (domain e) (domain f)))
             (pp (append (domain g) (domain h) (domain i)))))
```

[2]Given a group solution, a simple solution can be constructed by picking one element from each of the lists returned. Thus the number of simple solutions is the product of the list lengths.

```
"New solution"
(2 7 6)
(9 5 1)
(4 3 8)
"New solution"
(2 9 4)
(7 5 3)
(6 1 8)
"New solution"
(4 3 8)
(9 5 1)
(2 7 6)

...
```

Figure 21.3: Fragment of the output for Magic Squares

The above invocation will print out all the solutions. A fragment of the output from the above invocation is shown in Figure 21.3.

21.6 Finding a solution vs. finding an optimal solution

In many problems the need is to find an optimal solution, e.g. maximize the value that can be carried in a given bag (exercise 6 below). Our CLP system does not handle such problems directly, but it can be made to do so indirectly.

Our CLP system basically answers the question "Is there an assignment of variables satisfying the given constraints?". If $f(x, y, z)$ is some function that we want maximized (e.g. the value of the items accomodated in the given bag), the we can certainly add a constraint which reads "f(x,y,z)¿100". If this has a solution, then you increase the bound 100 to 110 for example and try again. If no solution is reported by the system for 110, then you know that the maximum weight that can be carried must be between 100 and 110. Thus you can try 105 and so on.

21.7 Exercises

The following problems should be solved using CLP.

1. Find all 3 digit numbers which equal the sum of the cubes of their digits. One example is $370 = 3^3 + 7^3 + 0^3$.

2. Suppose I am building a house. This involves activities (i) Foundation-laying, (ii) Building-ground-floor (iii) building-first-floor (iv) ground-floor-plumbing (v) ground-floor-wiring (vi) first-floor-plumbing (vii) first-floor-wiring (viii) clearing-the-plot (ix) buying-construction-material. Each activity takes 1 day, and several activities may go on simultaneously, subject to the following rules:

(a) Foundation-laying can only be done after clearing-the-plot.

(b) Any of the activities involving building, plumbing and wiring can only be done after foundation-laying and buying-construction-material.

(c) Building-ground-floor must happen before building-first-floor.

(d) Wiring and plumbing of any floor can happen only after the floor is built.

(e) Both the floors must be plumbed on the same day.

The question to be answered is whether all the activities can together finish in 5 days. Write a CLP program to answer the question. Your program should print which day each activity happens.

(Although this problem is trivial, you should organize your program so that it is clear that the ideas you use will be able to handle other similar, possibly more intricate problems.)

3. Write a program to solve the n queens problem. The problem is to place n queens on an $n \times n$ chessboard so that no two queens are in the same row, column or diagonal. You should print all the solutions, along with a serial number for each solution. (Hint: Use n domain variables, the ith representing the position of a queen in the ith column.)

Chapter 22

Implementation of Our CLP Solver

Building a CLP system has 2 parts: developing a language using which the user can specify the domain variables and the constraints, and developing a Solver. In this chapter we will present a brief high level description of th Solver used in our CLP system. Our solver is very simple as compared to solvers in commercial packages such as CHIP and ILOG. Yet, by studying our solver, we will be able to gain some insight on how solvers work in general. We will also describe some of the ideas that are present in commercial solvers but not in ours. All this will also help us in writing better CLP programs.

The simplest strategy using which a solver could be built is as follows. Since CLP programs only have a finite number of domain variables, each with a finite domain, then there are only a finite number of candidates one of which must be the solution. In particular, if the domain variables are $x_1, \ldots, x_n$ and each variable x_i has a domain of size d_i, then we only need to examine if any of the $\prod d_i$ ways of setting the domain variables satisfies all the specified rules. If Πd_i is large, then this brute force strategy will require high execution time.

We could also use more sophisticated strategies that attempt to *understand* the specified constraints. For example, if all constraints in a program happened to be linear simultaneous equations, then we can use GAUSS ELIMINATION or CRAMER'S RULE[1] This will certainly be faster than considering all possible ways of assigning the variables and seeing which ones satisfied the equations. If instead of equations we have inequalities, then the constraints are said to constitute a *Linear Program*, and there are a number of fast algorithms which can be used to solve such constraints. Unfortunately, even the simplest sets of constraints arising in practice are more complex than linear simultaneous equations or linear programs. However, it is possible to develop strategies better than the brute force strategy described above. We will see some of these below.

We start by discussing the brute force strategy. Then we present ways of refining the brute force strategy. Often these refinements reduce execution time considerably. Then we will describe our solver. Finally we conclude by listing out the implications of the material of this chapter for writing CLP programs.

To facilitate our discussion, we will consider a very simple CLP problem.

Example Problem: Suppose we have 3 domain variables x, y, z, with domains {1,2,3,4,5}, {4,6,8}, {5,6,7} respectively, and two rules: (i) $x^2 + 2 = y$, and (ii) $z + y = 11$.

We will sketch how the problem is solved by the basic strategy and also what happens

[1]Gaussian Elimination would be strongly preferred, if the number of equations and unknowns was large.

when the strategy is refined. We do this by giving the required Scheme code. Do note however, that the task of the CLP solver is harder: it must first accept the above problem in a suitable format (e.g. using the primitives described in the last chapter), and then effectively construct the code which when executed will find the solution.

22.1 A Brute Force Strategy

In this we simply need to systematically generate all possible combinations for the values allowed for x, y, z and then check which values satisfy the given constraints. So a Scheme solution could be:

```
(define (solve)
  (let ((domainx '(1 2 3 4 5))
        (domainy '(4 6 8))
        (domainz '(5 6 7)))
    (do* ((i 0 (+ i 1))
          (x (list-ref i domainx)))
         ((= i (length domainx)))
      (do* ((j 0 (+ j 1))
            (y (list-ref j domainy)))
           ((= j (length domainy)))
        (if (= (+ 2 (* x x)) y)
            (do* ((k 0 (+ k 1))
                  (z (list-ref domainz k)))
                 ((= k  (length domainz)))
              (if (and (= (+ 2 (* x x)) y)          ;; constraint 1
                       (= (+ z y) 11))              ;; constraint 2
                  (pp (list x y z)))))))))
```

Note that the variables `x,y,z` are all possible values, starting with 1,4,5 through 5,8,7 – for all the 45 combinations. For each such combination we check the constraints. Specifically, `constraint 1` shown in above code is checked 45 times, while `constraint 2` is checked only if `constraint 1` is satisfied.

22.1.1 Refinement 1

We note that we need not wait to generate values for `z` in order to enforce the rule that $x^2 + 2 = y$; once the two outer loops have associated values with `x` and `y`, we can right away check `constraint 1`.

This allows us to improve the previous program as follows.

```
(define (solve)
  (let ((domainx '(1 2 3 4 5))
        (domainy '(4 6 8))
        (domainz '(5 6 7)))
```

```
  (do* ((i 0 (+ i 1))
        (x (list-ref i domainx)))
       ((= i (length domainx)))
    (do* ((j 0 (+ j 1))
          (y (list-ref j domainy)))
         ((= j (length domainy)))
      (if (= (+ 2 (* x x)) y)              ;; constraint 1
          (do* ((k 0 (+ k 1))
                (z (list-ref domainz k)))
               ((= k  (length domainz)))
            (if (= (+ z y) 11))            ;; constraint 2
                (pp (list x y z))))
```

Now note that constraint 1 will be checked only 15 times. Further, the entire `do` loop for constructing z values will be executed only when `constraint 1` is satisfied. Thus this program will be faster than the one given earlier.

22.1.2 Refinement 2

In the brute force program, the order in which we wrote the loops did not matter. With refinement 1, however, it turns out that choosing the right order can lead to different execution times.

Consider for example, what happens if we have the loop for `y` and `z` before the loop for `x`. Now, we can move up `constraint 2` and leave `constraint 1` in the innermost loop. The resulting code is:

```
(define (solve)
  (let ((domainx '(1 2 3 4 5))
        (domainy '(4 6 8))
        (domainz '(5 6 7)))
    (do* ((j 0 (+ j 1))
          (y (list-ref j domainy)))
         ((= j (length domainy)))
      (do* ((k 0 (+ k 1))
            (z (list-ref domainz k)))
           ((= k  (length domainz)))
        (if (= (+ z y) 11)                      ;; constraint 2
            (do* ((i 0 (+ i 1))
                  (x (list-ref i domainx)))
                 ((= i (length domainx)))
              (if (= (+ 2 (* x x)) y)           ;; constraint 1
                  (pp (list x y z)))))))))
```

Now note that constraint 2 will be checked only 9 times. Further, the entire `do` loop for constructing `x` values will be executed only when `constraint 2` is satisfied. Very likely, this program will be faster than both the programs given earlier.

22.1.3 Other Refinements

The ideas described above are only perhaps the easiest ones. There are many more possibilities. We will present some of them briefly. Do note that these are only some examples; how to construct good solvers is an active research area.

One class of ideas is based on the notion of enforcing *consistency*. For example, suppose x has domain {1,2,3,4,5} and y has domain {3,4,5,6,7}. Suppose further that we have a constraint $x > y$. Now it is clear that x can never equal 1,2,3 and y can never equal 5,6,7. Thus, even though nominally the domain allowed for x as specified by the user might be {1,2,3,4,5}, we can effectively *shrink* it to be {4,5}, and the domain for y can similarly be shrunk to become {3,4}. Shrinking domains is very useful; in our solve procedure we would need to loop for only 2 iterations in each of the loops corresponding to x and y. This can result in considerable time saving. Of course, enforcing consistency will not always necessarily be useful. Had our constraint been `x+y=8`, we would not be able to rule out any of the values in either of the domains.

We could also consider *eliminating* certain variables. For example, if our constraint is $x+y = 8$, then we could substitute $8-x$ instead of every occurrence of y. If all our constraints were of this type, our solver would then be effectively doing Gaussian elimination!

22.1.4 Recursive formulation

The code described above using do loops can also be expressed recursively. This is done in a standard manner as indicated earlier: any loop can be reformulated as a recursive call to a suitable defined procedure. For example, the code of Section 22.1.2 could be written as:

```
(define (solve)
  (let ((domainx '(1 2 3 4 5))
        (domainy '(4 6 8))
        (domainz '(5 6 7)))
    (instantiate-yzx)
)

(define (instantiate-yzx)
    (do* ((j 0 (+ j 1))
          (y (list-ref j domainy)))
         ((= j (length domainy)))
      (instantiate-zx y)))

(define (instantiate-zx y)
    (do* ((k 0 (+ k 1))
          (z (list-ref domainz k)))
         ((= k  (length domainz)))
        (if (= (+ z y) 11)              ;; constraint 2
            (instantiate-x y z))))

(define (instantiate-x y z)
```

```
(do* ((i 0 (+ i 1))
      (x (list-ref i domainx)))
     ((= i (length domainx)))
  (if (= (+ 2 (* x x)) y)              ;; constraint 1
      (pp (list x y z)))))
```

As you can see this code will produce the same output as the code of Section 22.1.2. However, as seen in Exercise 4, this style of coding can be adapted for the case in which the number of domain variables is unknown at the time of writing the code.

22.2 Design of solvers

Most solvers are based on some kind of a brute force strategy augmented with refinements like the ones given above. Refinement 1 discussed is extermely commonly used. Refinement 2 is also used very often. The notion of consistency described above is also used in some solvers.

There are some difficulties in incorporating ideas such as *elimination* in general purpose solvers. If the user specifies all constraints using an `add-rule` format, then the constraint $x + y = 8$ will be presented to the solver as:

```
(add-rule (lambda (x y) (= 8 (+ x y))) x y)
```

To perform elimination, the solver has to infer that this rule really is a linear equation. Unfortunately, the solver is simply given the `lambda` expression above; the solver has no way to examine the `lambda` expression and determine that it is a linear equation. To overcome this difficulty, many solvers provide special rule forms. For example, we could have a command:

```
(linear-equation '(2 3 8) x y)
```

which is used to specify a linear equation $2x + 3y = 8$. Now, the solver knows that a linear equation is specified with the first argument being the coefficients and the last arguments the variables. With this knowledge the solver can perform the required elimination.

Several such special commands are provided in many solvers, in addition to the general facility to describe arbitrary constraints.

22.2.1 Our solver

Our solver also uses a brute force approach augmented with refinement 1 specified above.

Refinement 2 is not used. Instead, the order in which the variables are assigned values is determined by the order in which the constraints are specified. Thus, if the constraints are specified as given above, i.e. (i) $x^2 + 2 = y$, and (ii) $z + y = 11$, then the outermost loop would be for x, the next would be y and the innermost would be z. If however, the order of specifying the constraints was reversed, i.e. (i) $z + y = 11$, and (ii) $x^2 + 2 = y$, then the outermost loop would correspond to the variable first named in the first constraint, i.e. z. The next loop would be for y, and the innermost loop would be for x. Notice, that this requires the user to be somewhat careful in ordering the constraints. All orders will produce the same answers, however, some orders will produce the results faster.

Our solver does have a special command `distinct`. Why this command is useful is the subject of one of the exercises.

22.3 On writing CLP programs

From the discussion above, we can derive some guidelines as to how to write CLP programs. The basic idea on which our guidelines are based is:

> Attempt to have constraints be checked as early as possible.

Whenever we check a constraint check fails, we get an effect as in refinement 1: we may avoid a lot of additional work.

The first and most important guideline is: whenever possible, express constraints as separate rules rather than as a single rule which asserts the conjunction of the individual constraints. The constraints of our example problem could either be expressed as a two rules using:

```
(add-rule (lambda (x y) (= y (+ (* x x) 2))) x y)
(add-rule (lambda (y z) (= 11 (+ y z))) y z)
```

or as a single rule as:

```
(add-rule (lambda (x y z) (and (= y (+ (* x x) 2))
                               (= 11 (+ y z)))) x y z)
```

Our guideline says that the first form is preferred. The reason for this should be obvious: if we use the first form, we get the effect of refinement 1. If we use a single rule which constraint all the three variables `x,y,z` simultaneously, the solver will not be able to do any constraint checking before a candidate value is assigned to each variable. Thus by specifying a single complex rule, the execution will effectively proceed as with the brute force strategy without the effect of any of the refinements.

This basic rule applies will help with all solvers.

22.3.1 Guidelines for our solver

Our solver enforces the rules in the order they are given, hence it is useful if the programmer determines this order with some care.

It is possible to state some guidelines. For example, if there are two rules, one which constrains variables `x,y` and another that constrains variables `x,y,z`. Then it is appropriate to specify the rules in the same order as above, i.e. in increasing order of the number of variables constrained.

Again the justification is easy to guess. To reduce execution time, it is better if we can force constrains to be enforced as early as possible. Hence the rule which can be enforced after instantiating two variables should come before the rule which can be enforced only after three variables have been instantiated.

Consider another example. Suppose we have two rules $y \neq x$ and $u + v = 10$, where the domain of all the variables is $\{1,2,\ldots,9\}$. Which of the rules should you specify first? Here

we note that the rule $u + v = 9$ is much stricter: it allows only 9 possible combinations for u, v, viz. (1,8), (2,7), (3,6) and so on till (8,1). Whereas the rule $y > x$ more tolerant: it allows the ones above as well as many others such as (1,2), (1,3) and so on. In this case it is usually helpful to specify the stricter constraint (i.e. $u + v = 9$) first. This is because the stricter constraint will enable you to rule out many combinations early on, so that subsequent work could be more productive. We will examine these ideas in the exercises.

A natural corollary of the above observation is that among rules of the same kind, those that involve smaller domains should be considered first.

22.4 Guidelines vs. rigid dictums

Note finally, that what we have specified above are only guidelines – they are not foolproof. It is possible in some examples, that it is actually faster to specify an apparently less strict rule before a more strict rule. This is because the execution time is determined not just by the relative ordering of two rules, but the ordering of all the rules together.

In other words, writing CLP programs is an art, not an exact science.

22.5 Exercises

1. Insert code into the above which counts how many times each constraint is checked. You could do this, for example, by having a global variable `check-count` which `check` increments by two (since currently `check` checks two constraints in it). At the end of the program, print the value of the variable. This should give an indication of how long execution took.

2. Modify the above code so that the first constraint is checked as early as possible. Suitably change the code that modifies `check-count`. Remark on whether checking constraints earlier reduces execution time.

3. The code given above has the loop for `x` as the outermost loop, then the loop for `y`, then the loop for `z`. Will the code be correct if you exchange the loop order? Answer on the basis of the count reported in `check-count`.

4. Write a Scheme program (without using CLP) to solve the N-queens problem.

   (Hint: Since you dont know N, you cannot write N nested loops. However, you can use the recursive approach as follows. Clearly, if N were known, you could have written N separate procedures, with the ith procedure determining the position of the ith queen. However, note that all the N procedures would really be identical, except for the number i. Hence you should be able to write a single procedure which does the work of all of them. If it helps your thinking, write the 3 procedures required to solve the 3-queens problem, and then you should be able to see how to write a single procedure.)

5. Suppose you are given data about which countries in a map are adjacent to one another. This could be given as a list, for example, `'((india pakistan) (iran afghanistan)`

`(india nepal) (india china) (china nepal)(afghanistan pakistan))` The goal is to find a colour for each country such that adjacent countries get different colours. The famous four colour map theorem asserts that any map (drawn on a sphere) can be coloured in 4 colours. Can you write a CLP program that does this? Examine the different orders in which you might specify the constraints and see the effect it has on the size of the tables constructed during the execution.

6. Suppose you are given the value v_i and weight w_i for each of n items, $i = 1, \ldots, n$. Your can carry a total weight W (given). Can you pick a set that has the value at least What subset should you pick so that it is as valuable as possible? Suppose the items have values 50,20,35 and weights 10,5,8 respectively, and W=14. Then the last two items should be picked.[2]

   Can you formulate the problem as a set of rules rather than as a single rule?

7. Substitute a unique digit for each of the letters in the following so that it becomes a valid arithmetic expression.

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \\ \hline \end{array}$$

[2]This example should pursuade you that the problem is difficult; simple strategies such as always pick the most valuable item or the item with highest ratio value/weight do not work.

Chapter 23

Advanced topics in CLP

The goal is to show see how objective functions can be included into our solver, as well as strategies such as branch and bound. We begin by considering how the current solver can be used, and then motivate the need for additional primitives and hint how they will be implemented.

This is very incomplete.

23.1 Knapsack Problem

Alibaba has just entered his famous cave and is dazzled by all the wealth lying there. He needs to decide how to carry back the most valuable objects such that his knapsack does not break. Suppose his knapsack can carry at most W kg weight. He sees n objects lying in the cave and can instantly judge that the ith object has weight W_i and value V_i. Which objects should he carry?

Alibaba basically needs to make a decision about each object: whether to take it or not. These decisions are what we need to help him with. It is customary to model the ith decision by an *indicator variable* x_i, with $x_i = 1$ indicating that the ith item is selected, and $x_i = 0$ indicating that the ith item is not selected. These variables must satisfy the following constraint:

$$\sum_{i=1}^{n} x_i W_i \leq W$$

Note that this is simply saying that the weights of the selected objects must be at most W kg, which is indeed the condition for the knapsack not breaking. The value that is picked up by Alibaba is given as:

$$\sum_{i=1}^{n} x_i V_i$$

Our goal is to maximize this value. So following the discussion in the last chapter, we will formulate our problem as:

Find $x_i, i = 1, \ldots, n$ and the largest V such that

1. Each x_i is either 0 or 1.
2. $\sum_{i=1}^{n} x_i W_i \leq W$

3. $\sum_{i=1}^{n} x_i V_i = V$

To find the maximum we will simply adopt trial and error (or binary search). In other words, we will solve the above problem for different values of V, and see which is the largest value of V for which a solution is obtained.

23.1.1 Solving for a fixed V

For any fixed V, we can indeed formulate the conditions given earlier in the language of our CLP solver. However this may not be the best formulation. The reason is that we only have two conditions, each of which refer to all the n variables. We had remarked earlier that instead of having one condition that refers to many variables, it is better to have several conditions each of which refer to a few variables. This can be done using the following observations:

$$\begin{aligned} & \sum_{i=1}^{n} x_i W_i \leq W \\ \Rightarrow \quad & \sum_{i=1}^{j} x_i W_i \leq W - \sum_{i=j+1}^{n} x_i W_i \\ \Rightarrow \quad & \sum_{i=1}^{j} x_i W_i \leq W \end{aligned} \tag{23.1}$$

We can also argue similarly for the condition on V

$$\begin{aligned} & \sum_{i=1}^{n} x_i V_i \geq V \\ \Rightarrow \quad & \sum_{i=1}^{j} x_i V_i \geq V - \sum_{i=j+1}^{n} x_i V_i \\ \Rightarrow \quad & \sum_{i=1}^{j} x_i V_i \geq V - \sum_{i=j+1}^{n} V_i \end{aligned} \tag{23.2}$$

The last inequality follows since $\sum_{i=j+1}^{n} x_i V_i \leq \sum_{i=j+1}^{n} V_i$.

Hence we can use rules of the form (23.1) and (23.2).