Programming Cognitive Robots

Hector J. Levesque

A cognitive robot is one that uses explicitly represented knowledge to make decisions about what to do. Programming a cognitive robot is about giving it enough knowledge about its world, its actions, and what needs to be done for it to be able to calculate in a practical way the actions it should take. This book presents a new programming language for this purpose, and illustrates its use on a number of examples. It then follows with three case studies that can form the basis for term projects in cognitive robotics: one in a simulated world of squirrels, one for a LEGO robot, and one in a real-time video game environment. While the book is intended primarily for undergraduate computer science students, it also includes three chapters on more advanced topics in cognitive robotics. All the software described in the book can be freely downloaded. (A LEGO kit must be purchased separately for the LEGO project.)

In memory of my brother Paul

Contents

Projects

Advanced Topics

End Matter

Preface

This is a book about computer programming, but programming of a special sort. It is not about programming web servers, or weather simulations, or graphical user interfaces. It is about programming a *robot*. Furthermore, it is not about programming the low-level operations of a robot, like how it might interact with a video camera or control the motorized wheels on a mechanical platform. Nor is it about programming specific robotic tasks like grasping a fragile object or avoiding collisions in a crowded room. Instead, this is a book about getting a robot to figure out, at the highest level, what it should be doing. The premise here is that, before a robot even considers how to do things like picking up an object or navigating safely through a room, it needs to be able to decide whether it should be doing these things in the first place. Programming a robot to make high-level decisions like this in an informed way is what we are calling *cognitive robotics*.

Cognitive robotics

A *cognitive robot* (as the term will be used here) is a hardware or software agent that uses what it knows about its world to make decisions about what to do. So cognitive robots are to be contrasted with robots that do what they do because they have been explicitly programmed to do so, or with robots that do what they do because they have been trained to do so on massive amounts of data (the current fashion in much of AI).

The term cognitive robotics is due to Ray Reiter who first used it in 1993 to refer to the study of the knowledge representation and reasoning problems faced by an autonomous robot in a dynamic and incompletely known world. The emphasis, in other words, is on designing and implementing robot controllers that make use of *explicitly represented knowledge* in deciding what to do. It is not about robot controllers that merely solve a certain class of problems or happen to work in a class of application domains by learning or by some other means.

The sort of research pursued by Reiter and his colleagues was primarily theoretical in nature, and leaned heavily on the mathematics of logic and computation. Nonetheless, what came out of their work was a very different way of looking at programming. Instead of thinking of a program as a specification of what should happen inside a computer, a program could also be thought of as specifying a complex activity in the world that a cognitive robot might choose to engage in.

The thesis of this book is that, just as it is possible to write useful Python programs without knowing much about the mathematics of computation (Turing machines and the like, say), it is possible to program cognitive robots without having to master the mathematical foundations proposed by Reiter and colleagues. However, the focus on programming

means thinking hard, not just about what needs to be computed by a cognitive robot, but on how the required information should be represented and processed. Simple schemes that work well enough on small or toy versions of problems may not scale well on larger, more realistic cases. (Think of what works for calculating the first ten Fibonacci numbers, for instance, compared to what is needed for calculating the first hundred.) The assumption here is that a programmer will want to make deliberate choices about the data structures and algorithms a cognitive robot should use in keeping track of its world and deciding what to do, and not simply rely on a one-size-fits-all default choice.

Who is this book for?

Computer programming is not for the easily distracted. Parts of this book might make for casual reading, but other parts will require a more concentrated effort. It is expected that readers will want to sit at a computer and try out the examples in the book (and variations) for themselves as they are going through it. In the end, learning to program is a bit like learning to play the piano: hands-on practice is essential.

The ideal reader will already have an interest in robots or software agents at some level, as well as experience in several programming languages. A motivated third-year undergraduate computer science student should certainly fit the bill, but a hobbyist who has had to deal with a number of programming language should manage just as well. Readers who have never programmed before will likely find the book dry and unenlightening. Readers who have not seen a variety of programming languages may find the particular notation used here quirky and even exasperating (all those darn parentheses!).

So an ideal audience for this book will have two or more years of computer science education as background preparation. The expectation is that this book would not be used as a standalone textbook for an AI course, but as a resource for projects. For example, in a one-semester course, a third-year student should be able to get through the core part of the book (up to Chapter 6), and then take on a project in cognitive robotics. Three possible project areas are presented in Chapters 7, 8, and 9. A more advanced student might delve into some of the more demanding topics discussed in Chapters 10, 11, and 12.

The emphasis in this book is on getting robots to decide for themselves what actions to perform. There is much less emphasis on how the robots should actually perform the actions that have been selected. The robot itself is treated somewhat like an external server: it receives requests to do certain actions (or to return some sensing information), but how it goes about doing them is its own business and not the main focus here.

In a minimal project based on this book, a student might be content with a program that prints out the actions that the robot should perform. A more ambitious project would also involve building a robot or software agent that can carry out the selected actions. The robot itself might be built and programmed by another student in a completely different programming language on a separate computer, with the two parts communicating over TCP. Simple cases of how such a bipartite system would work over a TCP network are considered in detail in the project chapters 7, 8, and 9.

Overview of the book

The first two chapters of the book present preliminary material:

1. This chapter introduces the programming language used here, called *ERGO*, and the ideas behind its programs. It previews a small but complete *ERGO* program.

2. This chapter reviews the Scheme programming language. The *ERGO* system uses Scheme to do all the necessary work on numbers, strings, arrays, and so on.

The next four chapters show how to program a cognitive robot in *ERGO*:

3. This chapter presents the declarative part of *ERGO* programming, which involves specifying the properties of the world the robot lives in and the actions that affect it. This chapter also shows how a declarative specification can be used directly for basic (sequential) planning.

4. This is the first of two chapters on the procedural part of *ERGO* programming. This includes the usual deterministic facilities like sequence, iteration, and recursion, but also nondeterministic facilities involving search and backtracking.

5. This is the second of two chapters on the procedural part of *ERGO* programming. This one emphasizes the concurrent aspects along with the idea of performing some tasks while monitoring and reacting to other conditions in the world. Reacting to another agent is illustrated using minimax game playing.

6. This chapter looks at an *ERGO* system in its entirety and how it can control an actual robot or software agent over TCP or by other means. The idea is that an *ERGO* program will generate actions for the robot to perform, but the robot can also report exogenous (external) actions that have occurred outside the program.

The next three chapters present possible projects in cognitive robotics and are independent of each other:

7. In this chapter, *ERGO* is used to control a simulated squirrel in a world of acorns, walls, and other squirrels. The squirrel does not know where these are located, but can move around and use sensing to detect their presence. The goal is to be the first squirrel to collect a number of acorns.

8. In this chapter, *ERGO* is used to control a LEGO Mindstorm robot with motors and sensors. Here the goal is to wheel around on a floor or table top delivering and picking up ersatz packages at various locations.

9. This chapter explores the use of *ERGO* in a real-time video game. In this case, *ERGO* is used to control a car that is taking a joy-ride while attempting to elude a patrol car that is chasing it under the real-time control of the user.

The final three chapters discuss more advanced topics in cognitive robotics and again each can be read independently:

10. This chapter discusses the relationship between *ERGO* and the mathematical foundations of cognitive robotics in symbolic logic developed by Ray Reiter and his colleagues. The execution of a program is recast as a problem in logical reasoning.

11. This chapter and the next deal with an *ERGO* system that has only incomplete knowledge of its world. This chapter generalizes programming to the case where the system has to deal with the numerical uncertainty arising from the noise and inaccuracies in sensors and effectors.

12. In this chapter, the sequential planning seen in Chapter 3 is generalized to deal with incomplete knowledge. The generated plans will no longer be sequences of actions, but more complex graph-like structures that branch and loop.

The book concludes after Chapter 12 with a short epilogue and index material. Chapters 2 through 5 include short exercises. Chapters 10 through 12 have bibliographic notes. Chapters and sections marked with an asterisk (∗) are more advanced and can be skipped on first reading without loss of continuity. Technical terms are underlined when they are first used and indexed on Page 200.

Required software and hardware

There is considerable software used in this book, which will be freely available online. The starting point is the Racket system `https://racket-lang.org/` which runs on Windows, MacOS, and Linux, and must be installed and running to use everything else.

All the remaining system software was written by the author and can be downloaded from a special website associated with this book. Installation instructions are there as well. All the example programs appearing in this book and instructions for running them can be found at this site.

As for the three cognitive robotic projects considered in this book, the one discussed in Chapter 7 involves a software agent that runs in the Racket environment, and so no additional software or hardware is needed. The project discussed in Chapter 8 involves a hardware robot, the LEGO Mindstorms system `https://www.lego.com/en-us/mindstorms/`, which must be purchased separately (for about $350US). The software for managing the LEGO robot uses a freely available package from `http://www.ev3dev.org/`. Finally, the project discussed in Chapter 9 involves a software agent intended to run in the Unity video-game environment `https://unity3d.com/`, which is freely available to students.

One of the goals of the *ERGO* system discussed here is to ensure that programmers will not feel a need to switch to some other computational platform for reasons of efficiency. In other words, *ERGO* software is not intended just for toy problems or quick prototyping, with all the more "serious" development work carried out elsewhere. The execution speed of *ERGO* can be expected to lie somewhere between that of a highly-optimized compiled language like C and that of an interpreted language like Python.

What will a student get out of all this?

This book and its accompanying software are offered as an educational tool. There are three main things that a student can expect to learn here:

1. A student will learn yet another way to think about programming, different from issues in imperative programming, functional programming, logic programming, concurrent programming, object-oriented programming. Programming a cognitive robot has some aspects of all of these, but with a very different slant.

2. A student will see AI from a very different perspective than in traditional AI courses. The book is not based on probability and machine learning, nor on logic and theorem-proving. While it is possible to make those connections (and first steps can be found in the advanced chapters of this book), this book takes a more hands-on route to the idea of a cognitive agent.

3. A student will see robotics in a different light. Early computer programming in the 1950s stayed very close to the computer hardware. It took some time before the ideas of data and procedural abstraction led to the higher-level programming languages we now use. Similarly, most work in robotics today stays very close to the robotic hardware. This book attempts to take a more abstract view of what a robot is and what a robot can do.

As always, how much students actually get out of the book will depend on how much they put into it. Our hope is that the book will excite a student into exploring more deeply the many issues involved.

Acknowledgments

This book started a while back as a collaboration between Maurice Pagnucco and me. It would never have gotten off the ground without him, and the organization and direction of the book was something we arrived at together. He really should have been a co-author of the book, except for the fact that he did not get a chance to do any of the writing. Instead, he was co-opted into taking on a full-time load of administrative work at his university which, unfortunately, he appears to excel at. My loss was their gain. But I remain totally indebted to him for all he did contribute.

Next, I must thank my colleague and mentor, the late Ray Reiter, who was the first to think about cognitive robotics in a systematic way, and who inspired me on this topic as on so many others throughout my career. I was very fortunate to have been part of his Cognitive Robotics group at the University of Toronto, and I thank him and all the other members of that group, students, colleagues and visitors, for the ideas they planted in me. I want to especially single out Yves Lespérance, Fangzhen Lin, Richard Scherl, Giuseppe de Giacomo, and Sebastian Sardina for the development of GOLOG, CONGOLOG, and INDIGOLOG, the ancestors of the *ERGO* programming language discussed here.

I also want to thank my research friends in Cognitive Robotics, Vaishak Belle, Jim Delgrande, Sebastian Sardina, and Steven Shapiro, who went slogging through a draft of the book, hunting for infelicities, even after I warned them that it was intended for undergraduates and somewhat thin on research ideas. Are these kind folks then to blame for any errors or confusions that remain? I think not.

The chapters 10, 11, and 12 have bibliographic sections acknowledging the contributions of others to the ideas there. But many of the earlier examples were also lifted from the work of others. The fox, hen, grain problem and the jealous husbands problem from Chapter 3 are classics that date back to the ninth century. The basic elevator from Chapters 1 and 6 was presented in the first GOLOG paper. The reactive elevator from Chapter 5 was discussed in the first CONGOLOG paper. The grocery store example from Chapter 4 is due to Maurice Pagnucco. The squirrel world of Chapter 7 is a variant of the Monty Karel robot world written by Joseph Bergin and colleagues. The LEGO delivery robot of Chapter 8 was originally due to Maurice Pagnucco and me in a system called LEGOLOG.

I also want to thank the folks at MIT Press and Cambridge University Press who went through the book carefully, and tried hard to see how they could publish it, before eventually deciding that it did not work for them. *C'est la vie!*

Finally, and as always, I want to thank my family and friends for their love and unwavering support. "Aren't you supposed to be retired?" they would sometimes ask, but then were always willing to accept that, after a long career in academia, old habits die hard.

Toronto, July 2018

Preliminaries

Chapter 1

Introduction

As the title says, this is a book about programming cognitive robots. The book is not so different from other books on programming you might see online or at your favourite bookstore. Some of them are about programming in Python, or in Scala, or in Matlab. This one happens to be about programming in a language called *ERGO*. In this chapter, we give a general introduction to programming a cognitive robot and to *ERGO*. (The word "ergo" means "therefore" in Latin, emphasizing that a cognitive robot will need to be able to draw conclusions based on what it knows. But for reasons that will become clearer in Chapter 10, *ERGO* also stands for "ERGO Reimplements GOLOG.")

1.1 Declarative programming

What is special about *ERGO* programs is not how they look, but what they are about: they deal with the behaviour of a cognitive robot. This is a robot that expects to be given knowledge about its world, including what it can and cannot do, and about what needs to get done overall. The robot is then expected to figure out what to do.

So a major part of programming a cognitive robot in *ERGO* is telling it what it needs to know. This is sometimes called declarative programming. Traditional programming is mostly *procedural*: we tell a computer system to do something or other, like "increment variable X by 1" or "open a new text window on the screen." There will be some of that here too, but the emphasis in *ERGO* is on building what we call a *basic action theory* (or *BAT* for short): a declarative specification of the properties of the world that will matter to the robot (which we call the *fluents* of the world), and of the *primitive actions* available to the robot, including how those actions change the world or extract information from it.

A declarative specification like this does not deal with what the robot should do, however. Our goal is to develop a cognitive robot that can determine which of its primitive actions to perform and when. But how will we tell the system enough for it to know what to do? As we will see, there is considerable flexibility in *ERGO* in how we do this.

At one extreme, we will be able to tell the system quite explicitly the actions we have in mind. That is, there will be times where the simplest specification of what we want done will be to name the actions involved, like "push the button in front of you three times," say. (Of course, at this extreme, we wouldn't need a BAT at all; the system is not really deciding anything for itself.)

At the other extreme, we will be able to tell the system how we want the world to be and let it work out what to do, like "make sure everybody in my group has a copy of the memo," for instance. Here the system has to analyze the current state of the world, the desired final state of the world, and plan for itself a program of actions that will get it from here to there. Or we may not be interested in a single goal to achieve, but in a condition to be maintained, like "make sure everybody in my group is always made aware of my travel plans, but only ever send them messages on this during off-hours."

So at one extreme, the actions to perform are listed explicitly, and at the other extreme, they are left entirely implicit. As we will see, it is actually between these two extremes where the most interesting *ERGO* programming takes place. We will most often end up specifying what we want the robot to do not as an explicit sequence of actions, nor as a condition to achieve or to maintain, but in terms of what we call a *high-level program*.

1.2 Scripting at a high level

A high-level program in *ERGO* is somewhat like a script in a scripting language like Javascript or AppleScript or Python. There are all the usual programming constructs such as sequences, conditionals, iteration, recursion, concurrency. But there are three major differences:

1. The primitive statements of the program are not fixed in advance; they are the actions of the BAT, and vary from application to application.

2. The system has to reason with the information provided by the BAT to keep track of what those actions do, how they change the world or extract information from it.

3. Perhaps most significantly, an *ERGO* program can leave out many details, details which will then be filled in by the system during execution.

The idea here is that *ERGO* will keep track of what it is doing and why, as well as what it has found out about the world, and then use this information to sort out the details that have not been made explicit in the program.

For example, at some point, an *ERGO* program might say (in effect) "If condition Q is now true in the world, then do action A and otherwise do action B." It will be up to the system to know whether Q is true, and select A or B accordingly. This condition Q typically will not be something internal to the system, like "does variable X have value N?" or "is the text window currently visible on the screen?" but some external property of the world that the robot knows about, like "is the door in front of you locked?" So at one extreme, the *ERGO* program might have said "Perform actions C, D, then E," and it will be up to the system to determine how these actions change the world, including whether or not they make the condition Q true. At the other extreme, the *ERGO* program might say at some point "Now achieve condition P," and the system will have to figure out how to do this. (As a special case, the program might say "Now find out whether condition R is true" and the system might need to figure out how to do that.) Or the *ERGO* program might be nondeterministic and say something more like "Now do either action F or action G as required," where the system will need to make a reasoned (that is, non-random) choice based on what has been done so far and what still remains for it to do.

So in the end, the essence of programming cognitive robots is this:

> *Giving the system enough knowledge about the world, its actions, and what needs to be done for it to be able to calculate the appropriate primitive actions to perform.*

Furthermore, this knowledge needs to be represented in such a way that the system can not only complete the required reasoning, but do so efficiently enough to be practical.

1.3 Finding the right level of primitive actions

One thing this book is *not* about is building robots. An *ERGO* program presumes that there is already a robot of some sort (or perhaps a simulation of one) that is able to carry out the primitive actions specified by the BAT. The *ERGO* system will keep track of what the robot knows at any given point and what it is trying to achieve overall. The robot itself is responsible for performing the actions that have been selected by the *ERGO* system (and perhaps returning sensing information).

So what should those primitive actions be? Take the case of an ordinary robot made up of motors and sensors of various sorts. The actions for this robot need not be in one-to-one correspondence with the commands available for the motors and sensors. As a simple example, it may be useful to think of running a motor for N milliseconds as one indivisible, primitive action, even though this might actually require applying a certain voltage to the motor, waiting N milliseconds, and then reducing the voltage to 0. It is assumed, in other words, that there is some sort of *robot manager* between the *ERGO* program and the sensors and effectors of a robot. (See Figure 6.1 on page 93.) What *ERGO* takes to be a single primitive action may involve a number of smaller operations with the sensors and effectors, the details of which are of no concern to the rest of the *ERGO* program.

So imagine a robot operating in a household. Should its primitive actions be at the level of "apply voltage X to motor Y," or more like "grasp the object on the table with the left gripper", or more like "remove a large pot from the cupboard," or perhaps even things like "make two servings of spaghetti bolognese"? None of these possibilities are ruled out.

There is a bit of a balancing act involved in building a cognitive robotic system. If the primitive actions are too close to the lowest level commands to the sensors and effectors, the BAT may get mired in details best dealt with otherwise. For example, the goals the system is working on are likely not relevant in deciding how tightly to grasp an object. The object should be grasped just tightly enough so that it won't be dropped. So the information about the torques to apply might not be part of the BAT.

On the other hand, if the actions of the BAT are too far removed from the commands to the sensors and effectors, the whole point of having a cognitive robot might be lost. In the limit, we can imagine a robotic system with just one primitive action, something like "do it!" where the entire operation of the robot is now the responsibility of the robot manager! In that case, all the decision-making would be hidden in the performance of that one primitive action. The robot would not be able to use what it knows about the state of the world and the goals it is working on, should something go wrong, for example. Since the actions of a BAT are assumed by *ERGO* to be primitive and indivisible, the system is never able to interrupt a primitive action halfway through to consider achieving its goals in some other way.

In general, there are no hard and fast rules about what the primitive actions of an *ERGO* program should be. Part of the skill of programming cognitive robots (that comes with practice) is finding the right level of actions to use. These can be thought of as "basic behaviours" that a robot can carry out quickly and reliably, without concern for the higher-level goals being worked on. If a behaviour cannot be carried out sufficiently quickly, it might be best to break it down and use something more like a "start the behaviour" action and one or more "terminate the behaviour" actions (as discussed in Chapter 6). If a behaviour cannot be carried out sufficiently reliably, it might be best to use a primitive action more like "attempt to do the behaviour," and then use sensing information to assess the outcome (as discussed in Chapters 11 and 12).

Traditional robot programming might come to an end once all the basic behaviours of the robot are realized in code. In *cognitive* robot programming, we take these primitive actions as just the starting point for all the higher-level decision making.

1.4 What do *ERGO* programs look like?

Once somebody has seen a few programming languages, learning a new one like *ERGO* is often a matter of looking at some example programs to get a sense of what they are like in general, and then going to the manual for the minutiae, including the details of syntax.

Figure 1.1 shows an entire *ERGO* program for a simple elevator-like robot discussed in Chapter 6. The most noticeable thing about this program, perhaps, is all the parentheses! To seasoned programmers, this might suggest that *ERGO* is a Lisp-like programming language, and indeed the language is embedded in the Racket dialect of Scheme, a Lisp derivative. Apart from comments (which begin with a semi-colon), expressions in *ERGO* are all of the form $(e_1 \ldots e_n)$, where the e_i are either numbers, atomic symbols (that is, identifiers like `define-action`, `turnoff`, and `#:mode`), or further parenthesized expressions. In this book, we expect programmers to be able and willing to adapt to any syntactic regime and, in this case, to a fully parenthesized notation with prefix operators. So, for example, *ERGO* programs will say things like

```
(and (< 3 floor) (< floor 8))
```

instead of

```
3 < floor && floor < 8.
```

For those who have never actually programmed in Scheme, but have experience with languages like Python or ML, the language is not that hard to use (once you get beyond the parenthesized syntax), and a quick tutorial is presented in the next chapter.

Looking more closely, what does this elevator program in Figure 1.1 actually say? First, it says that there are only two fluents, that is, two properties of the world that this elevator needs to worry about: what floor the elevator is on (called `floor`) and which elevator call buttons have been pushed (called `on-buttons`). In addition, the program specifies that at the outset, the elevator is on floor 7 and buttons 3 and 5 have been pushed. This is what the elevator knows about its world initially. (This is the simple case where the values of these two fluents are known at the outset. In a more complex setting, a cognitive robot may have incomplete knowledge about its world.)

Figure 1.1: Program file `Examples/basic-elevator.scm`

```
;;; This is a version of the elevator domain, formalized as in Ray Reiter's
;;; original in Golog, where the actions take numeric arguments.

;; The basic action theory: two fluents and three actions
(define-fluents
   floor 7                    ; where the elevator is located
   on-buttons '(3 5))         ; the list of call buttons that are on

(define-action (up n)         ; go up to floor n
   #:prereq  (< floor n)
   floor n)

(define-action (down n)       ; go down to floor n
   #:prereq  (> floor n)
   floor n)

(define-action (turnoff n)    ; turn off the call button for floor n
   on-buttons (remove n on-buttons))

;; Get to floor n using an up action, a down action, or no action
(define (go-floor n)
   (:choose (:act (up n)) (:test (= floor n)) (:act (down n))))

;; Serve all the floors and then park
(define (serve-floors)
   (:begin
      (:until (null? on-buttons)
         (:for-some n on-buttons (go-floor n) (:act (turnoff n))))
      (go-floor 1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Main program: run the elevator using the above procedure

(define (main) (display (ergo-do #:mode 'first (serve-floors))))
```

The next three `define-action` expressions specify the primitive actions that the elevator is capable of performing: `up` (going up to a floor), `down` (going down to a floor), and `turnoff` (turning off a call button). Obviously different elevators will have different capabilities and limitations (and a much more intricate one is presented in Chapter 5), but this one is assumed to be able to carry out these three actions. Furthermore, in the case of going up to floor n, the action has a *prerequisite*: the action can only be performed successfully when the floor the elevator is currently on is below n: `(< floor n)`. Finally, the action has an *effect*: after it has been performed successfully, the elevator will be on floor n, that is, the `floor` fluent will have value n. The `down` action has a similar prerequisite and effect, and the `turnoff` action has an effect on which buttons are on, but no prerequisite.

The next two `define` expressions specify *ERGO* procedures of interest. The first one `go-floor` says that getting to a floor means either going up, going down, or just being on the floor, as appropriate. The second one `serve-floors` says that (for our purposes) the elevator service amounts to doing two things in sequence: first, repeatedly selecting any floor whose call button is on, getting to that floor, and turning off its call button until no call buttons are on, and second, parking the elevator on the first floor. (Note that this program

does not try to minimize elevator time by choosing the floors to serve more carefully, or deal with other minor complications like doors or passengers.)

The final line of the file says that the main thing to do overall is to find the first sequence of actions that would constitute an execution of the `serve-floor` procedure and print it out as a list. If this file is given to *ERGO* to execute, the output it produces might be something like this:

```
((down 3) (turnoff 3) (up 5) (turnoff 5) (down 1))
```

So *ERGO* is done when it has determined the primitive actions to execute. This list of actions could then be passed to the robot manager of an elevator (or a simulated one) for actual execution in the world.

In this example, the *ERGO* system calculates everything that the elevator needs to do in advance, a complete sequence of actions to perform. This is called *offline execution*. (In some cases of offline execution, a more complex specification than a sequence is needed.) In *online execution*, by contrast, *ERGO* calculates just the next action the robot needs to do, and then continues only after the robot has performed that action. This has two big advantages: first, it allows *ERGO* to work well in cases where it would be impractical to calculate in advance *everything* that needs to be done by a large program; second, it allows sensing information obtained by the execution of one action to help in determining what the remaining actions should be. (As we will see, online execution requires defining programmed interfaces between *ERGO* and the outside world.) These two modes of execution and others are discussed in Chapter 6.

Chapter 2

Scheme Review

This chapter offers a review of Scheme, the language in which *ERGO* is embedded, and which provides it with all the usual programming datatypes. The presentation is aimed at readers already somewhat familiar with Scheme or one of its cousins. The chapter has five sections. The first presents a quick overview of the language; the second reviews the programming basics; the third considers more advanced programming; the fourth covers the Scheme primitives used in the example programs in this book; and the fifth summarizes how to run Scheme (and *ERGO*) from the command line, and makes some brief comments about the running time to be expected.

2.1 Language overview

Central to the Scheme language is the idea of a *form*, which is an expression like `(+ 3 5)` that can be evaluated by the language processor. The value of a form will be a data element from one of the supported datatypes: numbers (integers, exact rational, and floating point), strings (enclosed in double-quotes), Booleans (written `#t` or `#f`), symbols (called atoms in Lisp), lists (using round or square parentheses), ports and channels (for reading/writing), functions (*aka* procedures), vectors and arrays (numerically indexed structures), and hash-tables (instead of Lisp property lists). Scheme symbols are written as sequences of characters excluding space, quote characters, hash marks, and parentheses. (The characters of a symbol are usually either alphanumeric or one of the following: `?`, `!`, `:`, `+`, `*`, `/`, `-`, and `_`.)

Scheme forms come in four distinct varieties: constants, variables, function applications, and special forms. These forms are evaluated in quite different ways, as follows:

1. A Scheme *constant* is a number, string, or Boolean, and it evaluates to itself.

2. A Scheme *variable* is any symbol that has been given a binding by a `lambda` or `define` (described below), and it evaluates to that binding. There are many predefined global variables in Scheme. Most of them are bound to functions, like the variables + and `append`, but some are bound to other values, like the variable `pi`. (The predefined global variables that will be used in this book are covered in Section 2.4).

3. A *function application* is a non-empty list whose elements are themselves Scheme forms. The first element must evaluate to a function, and the remaining elements

evaluate to the arguments for the function. The entire form then evaluates to the result of applying the function to those arguments. For example, `(+ (* 2 4) 5)` is a function application, where + is a variable that evaluates to the addition function, and the forms `(* 2 4)` and `5` evaluate to `8` and `5` respectively, arguments that are indeed appropriate for addition. The entire form then evaluates to `13`.

4. A *special form* is a non-empty list whose first element is a symbol, and which is evaluated in a special way according to that first element. The most common special forms used in Scheme are the following:

   - `(quote` v`)`, where v is any symbol or list, is a special form that evaluates to itself. This special form is almost always written as a single-quote character followed by the v. So `'hello` means the same as `(quote hello)`.
   - `(if` e_1 e_2 e_3`)`, where the e_i are forms, is a special form that is evaluated as follows: first e_1 is evaluated; if the value is `#f`, then e_3 is evaluated and its value is the value of the entire form; if the value is anything else (including `#t`, of course), then e_2 is evaluated and its value is the value of the entire form.
   - `(lambda (`$x_1 \ldots x_n$`)` e`)`, where each x_i is a distinct symbol and e is a form, is a special form that evaluates to the function of n arguments which, when given arguments $v_1, \ldots v_n$, returns as its value the result of evaluating e in a context where the variable x_i is bound to v_i. (See examples in the next section.)
   - `(let ((`x_1 e_1`)` $\ldots$ `(`x_n e_n`))` e`)` is an abbreviation for the function application `((lambda (`$x_1 \ldots x_n$`)` e`)` $e_1 \ldots e_n$`)`, and is the usual way variables are given temporary local bindings in Scheme programs.
   - `(define` x e`)` where x is a symbol and e is a form, is a special form that is evaluated for its effect, which is to bind the variable x to the value of the form e. This is the usual way variables are given persistent global bindings. The form `(define (`f $x_1 \ldots x_n$`)` e`)` abbreviates `(define` f `(lambda (`$x_1 \ldots x_n$`)` e`))`, and is the normal way of defining functions in Scheme programs.

A typical Scheme program in a file consists of a number of `define` forms, culminating in something like `(define (main)` e`)`. The Scheme program is then run by evaluating these forms in sequence to set up all the bindings, and then evaluating the form `(main)`.

From now on, for ease of wording, we follow the common practice of referring to a function using a variable that is bound to it. So we say things like "the function `main`" rather than the more accurate "the function that is the value of the global variable `main`."

2.2 Basic programming

While Scheme offers a variety of datatypes, to illustrate the programming ideas, the examples in this section and the next deal with numbers and functions over numbers only.

As noted in the previous section, the main idea of Scheme is that of a form passed to the language processor for evaluation. In the simplest case, the result is another form, but one that has been reduced (or simplified or normalized). So `(+ 3 5)` evaluates to `8`, as does `(* 2 4)`, `(* (+ 3 1) 2)`, and `8` itself. This evaluation is what takes places when Scheme is used interactively:

```
> (+ 3 5)
8
```

(The bold here indicates what a user would type in an interactive Scheme sesssion.) So at its most basic, interactive Scheme behaves like a prefix-notation calculator:

```
> (expt 2 (+ 3 5))
256
> 256
256
> (> (* pi pi) (max 3 10 7))
#f
```

Note that the evaluation of forms is *recursive*: to get the value of `(expt 2 (+ 3 5))`, the Scheme processor must (among other things) get the value of `(+ 3 5)`, and to do this, it must get the value of the variable +. As a numeric calculator, Scheme places no limits on the size of integers; the form `(expt 2 (expt 2 20))` will gladly fill a screen with digits.

Many Scheme forms are already in reduced form and therefore evaluate to themselves: numbers, strings, Booleans, and quoted expressions. All the remaining Scheme forms are either symbols or lists. Those that are symbols are variables (like the `pi` and the > above), and those that are lists are either function applications or special forms.

Let us return to the `(if` e_1 e_2 e_3`)` special form. This looks like a function application but it is not because the evaluation is done in a special way: only one of the e_2 and e_3 forms will be evaluated. Note also that the Boolean constant `#f` plays the role of falsity; any other Scheme datum plays the role of truth:

```
> (if 'hello 2 (/ 10 0))
2
> ((if (> 3 2) + *) 4 5)
9
```

The first form shows that no error is produced despite the `(/ 10 0)`. The second form illustrates how the first form in a function application need not be a symbol like +; it can be any form that evaluates to a function.

A related special form is `(case` e_0 `[`l_1 e_1`] [`l_2 e_2`] ... [else` e_n`])` where the e_i are forms and the l_i are lists. This special form is evaluated by evaluating the e_0, finding the first list l_i that contains that value as an element, and returning the value of the corresponding e_i as the value of the entire form:

```
> (case (+ 3 4) [(4 5 8) 7] [(3 7 9) (* 3 4)] [else 5])
12
> (case (+ 3 4) [(4 5 8) 7] [(3 6 9) (* 3 4)] [else 5])
5
```

A third example of a special form is `(lambda` p e`)` which evaluates to a function. In its simplest version, the p is a list of distinct symbols and the e is a form that uses these symbols as variables. For example, the special form `(lambda (x) (+ x 5))` evaluates to the function of one argument that adds five to its argument:

```
> ((lambda (x) (+ x 5)) 6)
11
> ((lambda (x) (+ x 5)) (min 12 6 (abs -18) 24))
11
> ((lambda (x y) (* (+ x 5) y)) 4 2)
18
> ((lambda (x fn) (fn 3 x 4)) 6 +)
13
```

As a notational convenience, the special form

(let ((*x*1 *e*1) ... (*xn* *en*)) *e*)

behaves just like

((lambda (*x*1 ... *xn*) *e*) *e*1 ... *en*)

and is the normal way of introducing temporary local variables:

```
> (let ((mult *) (x (expt 2 5)))
    (mult x (+ x 1) (+ x 2)) )
35904
```

Note that the `(expt 2 5)` form is evaluated only once because of this use of a local variable.

Of course it is also useful to have global variables whose values persist. These are introduced using the special form `(define` *x* *e*`)`, where *x* is a symbol and *e* is a form:

```
> (define three (- 5 2))
> (* three 5)
15
> (define add5 (lambda (x) (+ x 5)))
> (add5 2)
7
> (add5 (* three 2))
11
```

Note that the `define` special form does not produce any value; it is evaluated only for its effect. (It is also possible to *reassign* a variable to a new value as in traditional imperative programming languages using a `set!` special form, but we will not be using this in the programs here.) As a convenience, there is a shorthand for a `define` of a `lambda`:

(define (*f* *x*1 ... *xn*) *e*)

behaves just like

(define *f* (lambda (*x*1 ... *xn*) *e*))

and is the usual way new functions are defined. For example:

```
> (define (cube x) (* x x x))
> (cube 3)
27
> (define (sumsq x y)
     (+ (* x x) (* y y)))
> (sumsq 3 5)
34
```

When used as above, `define` produces a global variable; however, it can also be used within the body of a `let` or another `define` to produce local variables. For example,

```
(define (sumsq x y)
   (define xsq (* x x))
   (define ysq (* y y))
   (+ xsq ysq))
```

behaves just like

```
(define (sumsq x y)
   (let ((xsq (* x x)))
     (let ((ysq (* y y)))
       (+ xsq ysq))))
```

Note how the second `let` is nested within the first in this case.

The Scheme language uses *lexically scoping*, meaning that a form that is nested within another can refer to the variables of the enclosing one. Consider a form like this:

```
(define (sumsq x y)
   (define xsq e1)
   (define ysq e2)
   e)
```

The form e can use the variables `x`, `y` as well as `xsq` and `ysq`. However, if there are local variables introduced within e_1 or within e_2, they will not be visible to e. Of course both e_1 and e_2 will be able to use `x` and `y` (and, in addition, e_2 will be able to use `xsq` because of the nesting of `let` forms noted above). Furthermore, to allow for recursion, the variable `sumsq` can be used just like `x` and `y` within e_1, e_2, and e.

This lexical scoping rule allows the recursive factorial function to be written in the expected way: (Note: a `;` indicates a Scheme *comment*; the rest of the line is ignored.)

```
> (define (fact n)                ; the factorial function
     (if (< n 2) 1
         (* n (fact (- n 1))) ))
> (fact 8)
40320
```

The `let` special form has provisions for recursion as well. The form

```
(let f ((x1 e1) ... (xn en)) e)
```

is just like the previous `let` except that the symbol f can also be used within e as a local variable whose value is the value of `(lambda (`$x_1 \ldots x_n$`) e)`. So, for example, here is a form whose value is the sum of the numbers from 3 to 10:

```
> (let loop ((i 3))
     (if (> i 10) 0
         (+ i (loop (+ i 1))) ))
52
```

Temporary functions (often with names like `loop`) are useful even within named functions. For example, to sum the values of an arbitrary function from a lower bound to an upper bound, the following can be used:

```
(define (sumup incr low hi fn)
  (if (> low hi) 0
      (+ (fn low) (sumup incr (+ low incr) hi fn)) ))
```

Notice how the iteration is really only over a single variable and that the other variables stay fixed. It might be clearer therefore to write something like this:

```
(define (sumup incr low hi fn)
  (let loop ((i low))
     (if (> i hi) 0
         (+ (fn i) (loop (+ i incr))) )))
```

This is better. However, it still has a drawback: on the last line, the function must call the `loop` function, but also remember its current position and the value of `(fn i)` to be able to add it to the recursive result. This means that it must use a stack, and when the difference between `hi` and `low` is large, Scheme may run out of memory. Here is a better version:

```
(define (sumup incr low hi fn)     ; sum of fn from low to hi by incr
  (let loop ((i low) (acc 0))
     (if (> i hi) acc
         (loop (+ i incr) (+ acc (fn i))) )))
```

This one produces the same values, but in this case, the function is *tail-recursive*, meaning that when `loop` is called recursively, that function application is the last thing that needs to be done. Evaluation in this case does not require a growing stack. (The Scheme processor treats tail-recursion exactly like iteration in other programming languages: assign the new values to the variables and then branch to the top of the loop.)

Of course the difference between tail-recursion and non-tail-recursion is even more pronounced with functions that use recursion more than once. Here is the naive version of the Fibonacci function:

```
(define (fibo n)                   ; the naive version of Fibonacci
  (if (< n 2) 1
      (+ (fibo (- n 1))
         (fibo (- n 2)))))
```

Here is a tail-recursive version of the same function:

```
(define (fibo n)                                ; a better version
  (let loop ((n n) (cur 1) (prev 1))
     (if (< n 2) cur
         (loop (- n 1) (+ cur prev) cur) )))
```

The first version will not be able to calculate even `(fibo 100)` (since this would require on the order of 2^{100} arithmetic operations), while the second version will easily fill the screen with `(fibo 100000)`. Another way to deal with the problem of calculating the same value repeatedly (as in the naive version) is to remember (or "memoize") calculated values in a data structure so that they are computed only once. This can be done easily for Fibonacci using a *list* whose elements will be the Fibonacci numbers in reverse order:

```
(define (fibo n)                    ; a memoized Fibonacci using a list
  (define (fibo-list n)
    (if (< n 2) '(1 1)
        (let ((y (fibo-list (- n 1))))
          (cons (+ (car y) (cadr y)) y))))
  (car (fibo-list n)))
```

In more complex cases (of what has been called dynamic programming), a more elaborate data structure than a list might be required.

* 2.3 Advanced programming

The idea of a function returning another function for later use is perhaps one of the trickiest for Scheme novices to master (and maybe the least like what is done in more traditional programming). To see this in action, consider the `sumup` and `cube` functions described earlier. We have the following behaviour:

```
> (sumup 1 3 10 cube)
3016
```

Of course, `lambda` expressions can also be used as the final argument to `sumup`:

```
> (sumup 1 3 10 (lambda (x) x))
52
```

The final argument to `sumup` must be a function of one argument. So we might consider defining functions that *return* such functions as their values. Here is an example:

```
> (define (square-times y) (lambda (x) (* x x y)))
> ((square-times 2) 3)
18
> (define st5 (square-times 5))
> (st5 3)
45
> (sumup 1 3 10 st5)
1900
```

So `square-times` is defined to return a function of one argument. For example, the form `(square-times 5)` evaluates to a function that squares its single argument and multiplies it by five. The variable `st5` evaluates to the same function. Note how the `5` passed as an argument to the `square-times` function ends up being "frozen" in the `st5` function. (A function that includes values for some extra variables is sometimes called a *closure*.)

This idea is used extensively in the implementation of *ERGO*. It also allows Scheme to support an easy form of object-oriented programming. For example, suppose we want to define a class of "number pair" objects that support a variety of methods. Here is how a generic class might be defined:

```
(define (mypair x y)
  (lambda (msg)
    (case msg
       ((sum) (+ x y))
       ((diff) (abs (- x y)))
       ((dist) (sqrt (+ (* x x) (* y y))))
       ((print) (displayln x) (displayln y)))))
```

With this definition, we then get the following behaviour:

```
> (define pair1 (mypair 3 4))
> (pair1 'sum)
7
> (pair1 'diff)
1
> (pair1 'print)
3
4
> (pair1 'dist)
5
```

Note that the body of a `lambda` is evaluated only when the function is actually called with arguments. Among other things, this allows us to create objects that are potentially infinite streams of data. For example, consider this:

```
(define all-evens (let loop ((i 0)) (lambda (a) (if a i (loop (+ i 2))))))
```

Without the `lambda`, this definition would generate an error attempting to call `loop` with ever larger values of `i`. With the `lambda` however, we freeze the current value of `loop` and `i`, and delay the evaluation until the function is called:

```
> (define (first s) (s #t))
> (define (rest s) (s #f))
> (first all-evens)               ; the first element
0
> (first (rest all-evens))        ; the second element
2
> (first (rest (rest all-evens)))  ; the third element
4
```

So in effect, `all-evens` behaves like an infinite stream of even numbers.

A final Scheme idea used extensively in the implementation of *ERGO* is that of a function to be used as a *continuation*, that is, as the final destination of a computed value (or values). For example, consider a function like this:

```
> (define (myarith x) (+ x 7))
> (myarith 9)
16
```

We might define `myarith` to take an extra argument to be called on the computed value:

```
> (define (myarith x fn) (fn (+ x 7)))
> (myarith 9 (lambda (v) v))
16
> (myarith 9 sqrt)
4
> (myarith 9 (lambda (v) 'ok))
'ok
```

In a sense, this version of the function `myarith` no longer returns a value; it does its local computation and then asks the `fn` argument to handle the result.

This idea allows us to program things like the backtracking needed in *ERGO* in a very convenient way. In the implementation, each *ERGO* primitive takes two extra functions as arguments, a failure continuation and a success continuation. The primitive never returns a value. In some cases, it calls the failure or success continuation directly with some computed value. In other cases, the primitive passes the buck to a second primitive, but with a modified failure or success continuation that will do additional work if it is ever called. For example, in nondeterministic choice, the first primitive will compute a first choice and a new failure continuation; another choice will be computed only if the second primitive later fails and calls this new continuation. (This is discussed in more detail in Section 4.6.)

2.4 The language primitives

This section reviews the language primitives of Scheme, that is, the predefined global variables of Scheme whose values happen to be functions. (All the global variables that actually get used in example programs in this book are listed on Page 197.)

2.4.1 Numbers

Many of the functions involving numbers have already been used: `+`, `*`, `-`, `/`, `expt`, `max`, `min`, `abs`, `sqrt`. There are many others. The main predicates over numbers are `>`, `<`, `>=`, `<=`, and `=`. The function `random` returns a pseudo-random number between 0 and 1.

2.4.2 Strings

There are many string-related functions in Scheme, but in this book, strings are used mainly for displaying messages. The `display` function can be used to print any datum including strings and returns no value. If the string contains a `\n`, a newline is printed:

```
> (let ((x 34)) (display "The value is ") (display x) (display ".\n"))
The value is 34.
```

The `displayln` function is just like `display` except that it always terminates with a `\n`. More convenient is the `printf` function, modeled after the one in the language `C`:

```
> (let ((x 34) (y 13)) (printf "The values are ~a and ~a.\n" x y))
The values are 34 and 13.
```

The `eprintf` function is the same but sends the output to the standard error port.

2.4.3 Symbols

The function `symbol?` tests whether its argument is a symbol. The only other function worth noting is the predicate `eq?` which tests if two symbols are identical. The predicate `equal?` is used to test if two arbitrary data elements print the same. The predicate `eq?` (and its derivatives, `remq`, `memq`, `assq`, and `hasheq` described below) can also be used on Booleans and small integers (less than 2^n on an n-bit machine).

2.4.4 Lists

As can be expected from its ancestry in Lisp, Scheme has a number of list primitives. The following come directly from Lisp: `cons`, `car`, `cdr`, `list`, `append`, `reverse`. One minor difference with Lisp is that the result of evaluation is displayed as a quoted expression:

```
> (append '(a b c d) '(e f g))
'(a b c d e f g)
> (reverse (cons 1 '(2 3 4)))
'(4 3 2 1)
> (car (cdr '(joe john jim)))
'john
```

As in Lisp, there are also functions cαr, where α is a string of up to four `a` or `d` characters. For example, the function `caddr` returns the third element of a list. The function `list-ref` returns the i-th element of a list for a given i (indexed starting at 0). The function `null?` tests if a list is empty. (Note that the symbol `nil` has no special status in Scheme; the form `'()` is used to refer to the empty list.) The function `memq` tests if a symbol is an element of a list, while `member` does the same for any datum. (When these functions do not return `#f`, they return the tail of the list containing the element.) The function `remq` deletes the first occurrence of a symbol from a list, while `remove` deletes the first occurrence of any datum. The functions `remq*` and `remove*` delete every element of one list from another. Regarding lists and numbers, the function `take` returns the first n elements of a list, `drop` returns all but the first n elements, and `length` returns the length of a list.

It is often useful to pair up symbols with other data. An association list (or alist) is a list of the form `((`s_1 d_1`) (` s_2 d_2`) ... (`s_2 d_2`))`, where the s_i are symbols and the d_i are any data. The function `assq` takes a symbol s and an alist as arguments, and returns the first `(`s_i d_i`)` whose s_i is s. (If the s cannot be found in the alist, `assq` returns `#f`.)

The function `map` applies a function to each element of a list (or lists of the same length) and returns a list of the results:

```
> (map (lambda (x) (+ x 3)) '(4 6 8))
'(7 9 11)
> (map - '(10 7) '(2 3))
'(8 4)
```

The functions append-map, sum-map and product-map are similar to map except that instead of making a list of the results, they are appended, added, or multiplied together. The functions and-map and or-map are similar but perform Boolean operations on the results, as explained later. The function for-each is similar but applies the given function for effect only and returns no value. The function filter is used to produce those elements for which the given function does not return #f.

Just as let provides a convenient alternative to the use of an inline lambda, there are similar alternatives for mapping functions. The expression

(for/list ((x_1 e_1) ... (x_n e_n)) e)

is equivalent to

(map (lambda ($x_1 \ldots x_n$) e) e_1 ... e_n).

The functions for/append, for/sum, for/product, for/and, for/or, for/only and for do the analogous job for the mapping functions append-map, sum-map, product-map, and-map, or-map, filter and for-each respectively.

A quoted expression is any symbol or list preceded by a single quote, as in Lisp. A backquoted expression is similar except that the backquote character ‘ is used: this evaluates like a quoted expression except that a comma can be used to refer to the value of a form. A comma followed by an @ symbol says that the form, which must evaluate to a list, should be spliced into the list. Here are some examples:

```
> (let ((x 3)) ‘(a b x d))
'(a b x d)
> (let ((x 3)) ‘(a b ,x d))
'(a b 3 d)
> (let ((x '(1 2 3 4))) ‘(a b ,(cdr x) d))
'(a b (2 3 4) d)
> (let ((x '(1 2 3 4))) ‘(a b ,@(cdr x) d))
'(a b 2 3 4 d)
```

2.4.5 Boolean values

As already noted, in terms of Boolean values, there is really just #f and everything else. Nonetheless, there are a few Boolean operations worth mentioning. The forms #t and #f are constants and evaluate to themselves. The function not returns #t when its argument evaluates to #f and #f otherwise. Conjunction and disjunction are performed using the and and or special forms. They evaluate their arguments only as far as needed, returning either #f or the value of one of their arguments.

```
> (and (> 2 3) (/ 10 0))
#f
> (and #t #t #t 1 2 3)
3
> (or #f #f #f 1 2 (/ 10 0))
1
```

The functions `and-map` and `or-map` provide a form of universal and existential quantification respectively over lists. The function `and-map` applies a given function to each element of a list (or to lists of the same length) and conjoins the results, stopping if any of the applications return `#f`; `or-map` is analogous but disjoins the results, stopping if any of the function applications return something other than `#f`:

```
> (or-map (lambda (x) (< x 5)) '(6 7 8 9))
#f
> (and-map > '(7 9) '(5 8))
#t
```

As noted earlier, the expressions `for/and` and `for/or` provide convenient alternatives:

```
> (for/or ((x '(6 7 8 9))) (< x 5))
#f
```

* 2.4.6 Ports and channels

The `display` function sends its datum to the standard output. The function `read` (of no arguments) is the opposite of `display` and returns the next datum from the standard input, blocking if there is none. These input-output functions take an optional extra argument which is a port indicating where the input or the output should take place.

Ports must be opened and closed explicitly. Here is an example with files:

```
(let ((iport (open-input-file "my-source-file.txt"))
      (oport (open-output-file "my-target-file.txt")))
  (displayln "--- Here is the processed value" oport)
  (let ((data (read iport))) (displayln (process data) oport))
  (displayln "--- That was the processed value" oport)
  (close-output-port oport)
  (close-input-port iport) )
```

This example does a single `read` from the file. Obviously, multiple reads are possible while the file is open, with data satisfying `eof-object?` returned at the end of a file.

Ports over TCP connections are slightly more complex. Conceptually, a program will either be a *TCP client*, wanting to send requests to another source, or a *TCP server*, wanting to receive requests. However, in both cases, the communication may need to be two-way: for example, a client may send a question to the server and expect to receive an answer. Therefore both clients and servers will use input and output ports: both `open-tcp-client` and `open-tcp-server` return a *list* whose first element is an input port and whose second element is an output port.

For example, assume that there is already a server running somewhere that is willing to receive requests on TCP port number 8123. The following shows a simple client:

```
(let ((ports (open-tcp-client 8123 "localhost"))) ; make the TCP connection
  (my-requester (car ports) (cadr ports))         ; make some requests
  (close-input-port (car ports))                  ; close the TCP ports
  (close-output-port (cadr ports)))
```

The `open-tcp-client` produces an error if there is no server waiting on port 8123. (The second argument is the hostname and defaults to `"localhost"`.) Data can be sent to the server on the output port using `display`, and whatever answers received on the input port.

A server can be defined in a similar way:

```
(let ((ports (open-tcp-server 8123 "localhost"))) ; make the TCP connection
  (my-server (car ports) (cadr ports))            ; serve some requests
  (close-input-port (car ports))                  ; close the TCP ports
  (close-output-port (cadr ports)))
```

In this case, the `open-tcp-server` blocks until some client makes a connection. (Again, the hostname argument can be omitted.) Data can then be received from the client on the input port using `read`, and whatever answers sent back on the output port.

Channels are FIFO queues that are similar to ports, but are typically used for synchronization purposes. The form `(make-channel)` creates a new channel, after which the form `(channel-get` *chan*`)` behaves like a `read`, blocking if the queue is empty, and otherwise returning the entry at the front of the queue (and removing it). Similarly, the form `(channel-put` *chan datum*`)` behaves like a `display`, adding the entry *datum* to the back of the queue.

2.4.7 Hash-tables, vectors, and arrays

Hash-tables provide mappings with near constant-time access from symbols to arbitrary data. A hash-table is created using the `hasheq` function and the mapping is accessed using the `hash-ref` function.

```
> (define age (hasheq 'john 10 'george 12 'jimmy 14 'joe 9))
> (hash-ref age 'george)
12
> (hash-ref age 'joe)
9
```

The `hash-ref` function takes an optional third argument which is the value to return if no mapping can be found.

```
> (hash-ref age 'george 23)
12
> (hash-ref age 'jess 23)
23
```

A new hash-table can be created from an old one: the expression `(hash-set` *h s x*`)` has as value a hash-table just like *h* except that symbol *s* is now mapped to value *x*.

Vectors provide a similar mapping but from numbers (that is, from non-negative integers) to arbitrary data. A vector is created using the `vector` function. The mapping is accessed using the `vector-ref` function, where the indexing starts at 0.

```
> (define b (vector 'john 'george 'paul 'ringo 'zeppo))
> (vector-ref b 2)
'paul
```

This is like the `list-ref` function, but with constant access time. A vector can be created from an old one with (`vector-set` *v i x*), just like with `hash-set`.

Arrays provide a similar mapping but from lists of numbers to arbitrary data. An array is created using `build-array` whose arguments are the dimensions (as a list) and a function that returns the initial mapping, where indexing again starts at 0. (A one dimensional array is like a vector.) The mapping is accessed with the `array-ref` function.

```
> (define coords (build-array '(3 5) +))
> (array-ref coords '(2 4))
6
```

An array can be created from an old one with (`array-set` *a l x*) just like with `hash-set`.

The functions `hash-set`, `vector-set`, and `array-set` make modified copies of their first arguments. The functions `hash-set*`, `vector-set*` and `array-set*` behave just like their unstarred versions, but allow multiple changes with a single copy. For example, (`hash-set*` *h* s_1 v_1 ... s_n v_n) has as value a hash-table that is like *h* but where each symbol s_i is mapped to v_i, evaluated in sequence. (There are Scheme functions `hash-set!`, `vector-set!` and `array-set!` that actually modify existing data structures, but we will not be using them in the examples in this book.)

∗ 2.4.8 Functions

The usual way to produce a function is with the `lambda` special form. However, there are some additional features of `lambda` worth noting, all of which then carry over to the `define` special form.

The general form is actually (`lambda` *p* e_1 ... e_n) where the e_i are forms. The idea is that the function will evaluate all in the e_i in sequence (for effect), returning as value the value of the final e_n. (The `let` and `case` special forms are similar.)

In addition, the *p* need not be a list of symbols. If a single symbol is used instead, it indicates a function that takes an arbitrary number of arguments; the symbol is then a variable whose value is a *list* of the actual arguments:

```
> ((lambda x (append (cdr x) x)) 'a 'b 'c 'd 'e)
'(b c d e a b c d e)
```

The *p* can also be a list of symbols that ends with a `.` and then a symbol. In this case the final symbol is a variable whose value is a list of all the remaining arguments (if any):

```
> ((lambda (x y . rest) (cons (* x y) rest)) 3 4 5 6 7 8)
'(12 5 6 7 8)
```

When arguments have been collected in a list, the function `apply` can be used to pass them to functions that require multiple arguments:

```
> (apply + '(2 3 4))
9
> ((lambda x (apply * (cddr x))) 2 3 4 5)
20
```

Finally, a `lambda` special form can define a function that takes optional arguments. These are indicated by putting within the *p* a keyword, that is, a special symbol beginning with `#:`, followed by a list [*v e*], where *v* is a symbol and *e* evaluates to its the default value:

```
> (define (test x #:my-optional-arg [y 3]) (+ x y))
> (test 5)
8
> (test 5 #:my-optional-arg 4)
9
```

There are no provisions for defining special forms as such in Scheme, but there is an elaborate macro facility which will not be used in the programs in this book. (It is used extensively in the implementation.)

2.5 Running Scheme and *ERGO* programs

Once Racket Scheme and *ERGO* have been properly installed, they can be run interactively from a command line using the following:

```
racket -l ergo -i
```

To load a file, something like

```
racket -l ergo -f myfile.scm -i
```

would be used. This file would normally contain a number of `define` expressions to be used in the interactive session, and perhaps some definitions that use the *ERGO* primitives discussed in the chapters to follow. The expression `(include "my-other-file.scm")` within this file can be used to load additional user files.

To load that same file and then evaluate `(my-main-function 3 5)` in non-interactive mode (with standard input and output for `read` and `display`), the following is used:

```
racket -l ergo -f myfile.scm -e '(my-main-function 3 5)'
```

The flag `-m` can be used as an abbreviation for `-e '(main)'`. The function `main` of no arguments would then need to be defined somewhere in the file `myfile.scm`. (This function can also be defined to take arguments, in which case it will be given the command-line arguments after `-m` as strings.) Typically, it would call one of the *ERGO* top-level functions like `ergo-do` or `ergo-simplan` as defined in later chapters.

In developing Scheme and *ERGO* programs, a programmer needs to keep in mind what to expect in terms of running time, to get a sense of when extra care will be required in the programming. To this end, we present a very rough guideline for those *ERGO* programs that perform some sort of search (as many of them will end up doing):

> **The Million-Billion Rule for *ERGO* search:**
> *A search through a million nodes or less can typically be done quickly enough for most purposes; a search through a billion nodes or more will need special attention.*

So, for example, searching a binary tree of depth twenty in *ERGO* will not be a problem, whereas searching one of depth thirty will require extra work.

As a concrete example, consider the "fox, hen, grain" puzzle presented in the next chapter. As a planning problem, the solution requires seven actions, and since there are

only four possible actions to consider at each step, the total search space is well under a million. On the other hand, for the "jealous husband" puzzle in that same chapter, because a solution requires eleven actions and there can be as many as twenty possible actions to consider at each step, some attention is required. (Sections 3.6 and 4.5 discuss efficiency considerations in more detail.)

2.6 Exercises

1. Write a tail-recursive version of the factorial function. Compare it to the non-tail-recursive one given in the text. Is there any reason to prefer one over the other?

2. Write a recursive definition of the `map` function. Do the same for the `sum-map` function. Noting the similarities, write a recursive function that generalizes both. (It is called `foldl` in Scheme.) Define `append-map` in terms of this function, and explain why `and-map` cannot be defined in this way.

3. One way to deal with floating-point numbers of arbitrarily large precision in Scheme is to use arbitrarily large integers: multiply the floating-point inputs by some large power of 10, do all the calculations over integers, and then reinterpret the final output as a floating-point number. Use this idea to calculate the first 1000 digits of the square root of 2 using Newton's method.

4. Languages like Scheme are often thought of as unsuitable for floating-point calculations, for various reasons. But some of these number problems lend themselves nicely to Scheme solutions. Write a function that takes four arguments, an error tolerance, a function f of n arguments, a list of of n lower bounds, and a list of n upper bounds, and computes using the Monte Carlo method the definite integral of f in the area defined by the given bounds to within the given tolerance.

Fluents, Actions, Programs

Chapter 3

Basic Action Theories and Planning

What makes cognitive robotics different from robotics more generally is the *cognition*. A cognitive robot does not simply act and react to the world around it; rather, it thinks about that world and uses what it knows to help it decide what actions to perform.

In general, of course, the knowledge of a robot is not expected to be fixed in advance. A robot might not know what is inside a box, for example, but may be able to find out by looking inside. For now, however, it will be convenient to make a very strong assumption: *anything that the robot needs to know to do its job will be given to it at the outset.* There will be no need to deal with sensing or perception since there is no additional information for the robot to acquire as it tracks its changing world. (This assumption of complete knowledge will be relaxed in later chapters.)

3.1 A world of rooms with boxes

Imagine a world as depicted in Figure 3.1. This shows the floor plan of a building with four rooms connected by three doors, two open, one closed. There are two boxes and a robot located in the building. The layout of the rooms and doors are considered to be *static* features of this world. But whether or not a door is open and what room the robot and the boxes are located in are considered to be *dynamic* features. In particular, it is assumed that the robot can open or close a nearby door; it can change its location by going through a door when the door is open; it can change the location of a box by pushing it through a door, provided the door is open and the robot and the box are in the same room.

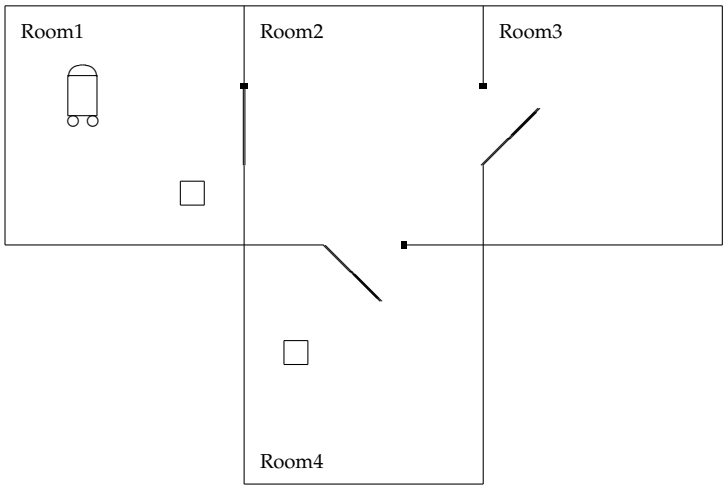
In representing knowledge about a world like this, three main concepts are relevant:

- a *fluent* is a property of the world that might be changed by an action;
- a *primitive action* (or action, for short) is an event that changes some fluents; and
- a *state* is a setting of all the fluents in the world.

The idea is that the world starts in some initial state, like the one depicted in the figure, where each fluent has a certain value. Then, as actions occur, the state of the world changes, and the fluents come to have different values.

A specification of all the fluents, actions, and states of a dynamic world is called a *basic action theory*, or *BAT* for short. In the simplest case, this specification is formulated

Figure 3.1: A simple robot world



in *ERGO* using two functions: `define-fluents` and `define-action`. The best way to get a sense of what is possible is to look at some examples. (As a programming language, *ERGO* can be thought of as Scheme with a few extra predefined functions and special forms. These are indexed on Page 198. A one-page summary of *ERGO* as a whole is included with the software distribution and reprinted on Page 199.)

3.2 Fluents and their states

Here is a first example of defining the fluents of a basic action theory:

```
(define-fluents
   door1-state 'closed
   door2-state 'open
   door3-state 'open
   box1-location 'room1
   box2-location 'room4
   robot-location 'room1)
```

This declaration says that a state of the world can be described using six fluents of the given names and whose values in the initial state are as given. In general, an *ERGO* BAT will contain one or more declarations of the form

`(define-fluents` $fluent_1$ ini_1 ... $fluent_n$ ini_n`)`

where each $fluent_i$ should be a symbol, and each ini_i should be a Scheme form whose value is the initial value of the fluent. (A single fluent can be defined with `define-fluent`.)

It is often convenient to use Boolean values or numbers (instead of symbols like `open` or `closed`) to represent a state using something more like this:

```
(define-fluents
   door1-open? #f   door2-open? #t   door3-open? #t
   box1-room-num 1  box2-room-num 4  robot-room-num 1)
```

(It is customary, though not necessary, to place a `?` at the end of Boolean-valued items.) But any Scheme value can be used to represent the value of a fluent. So, for example, the state of the doors might also be represented by a list of the doors that are open, and the locations of the objects might be represented by a hash-table:

```
(define-fluents
   open-doors '(door2 door3)
   location   (hasheq 'box1 'room1 'box2 'room4 'rob 'room1))
```

In this case, a single fluent is used to represent what is known about the location of all the objects: the value of the fluent `location` is a hash-table that maps any object (a box or the robot `rob`) to the room where it is located. (The value of a fluent can even be a Scheme function, as discussed below in Section 3.3.2.)

3.2.1 Fluent expressions

Once fluents have been declared using the `define-fluents` function, *ERGO* programs will be able to refer to the value of a fluent in a state. To do so, a program uses what is called a *fluent expression* or *fexpr*, which is nothing more than a Scheme form where the fluent names may appear as global variables.

So, for example, if `box2-room-num` is a fluent whose value in a state is `1`, `2`, `3` or `4`, then

```
(* box2-room-num 5)
```

is an fexpr whose value in that state is `5`, `10`, `15`, or `20`. If `open-doors` is a fluent whose value in a state is the list `(door2 door3)`, then

```
(memq 'door1 open-doors)
```

is an fexpr whose value in that state is `#f`. If `location` is the fluent defined as a hash-table above, then

```
(hash-ref location 'box1)
```

evaluates to `'room1`, while

```
(eq? (hash-ref location 'box2) (hash-ref location 'rob))
```

has value `#f`.

3.3 Actions

In *ERGO*, a state of the world is considered to change only as the result of an action. An action is declared by indicating its *prerequisite* (or precondition), that is, a condition that must be true before the action can be performed, and its *effects*, that is, the fluents changed by the actions and what their new values will be.

As a very simple example, suppose that we have the fluents door1-state, box1-loc, and rob-loc, described above. Imagine that there is an action push12! that can push box1 from room1 to room2 under the following conditions: door1 must be open, and both box1 and the robot must be located in room1. This action can be defined using the function define-action:

```
(define-action push12!
   #:prereq   (and (eq? door1-state 'open)
                   (eq? box1-loc 'room1)
                   (eq? rob-loc 'room1))
   box1-loc   'room2
   rob-loc    'room2)
```

(It is customary, though not necessary, to place a ! at the end of the name of an action.) The general form of a define-action expression is this:

(define-action *action-name* #:prereq *fexpr* $fluent_1$ $fexpr_1$ ... $fluent_n$ $fexpr_n$)

The idea is that the defined action can only be performed in states where the #:prereq fexpr is true, and it changes the state of the world in such a way that $fluent_i$ takes on the value of $fexpr_i$. The fluents named must be among those defined by the define-fluents declarations in effect, but the action need not mention all of them. Those that are not named are considered to be unaffected by the action. For the push12! action above, the state of the doors and the location of box2 is unchanged by the action.

An *ERGO* program will typically define many actions. Here is one that opens door1:

```
(define-action open1!
   #:prereq      (or (eq? rob-loc 'room1) (eq? rob-loc 'room2))
   door1-state 'open)
```

In this case, the action is considered to be possible only when the robot is located in room1 or room2. (However, note that this version does not require the door to be closed initially.) Here is a variant that toggles the state of door1:

```
(define-action toggle1!
   #:prereq      (or (eq? rob-loc 'room1) (eq? rob-loc 'room2))
   door1-state   (if (eq? door1-state 'closed) 'open 'closed))
```

If door states are represented using Boolean values (where true means open), the toggle action can be represented more concisely like this:

```
(define-action toggle1!
   #:prereq      (or (eq? rob-loc 'room1) (eq? rob-loc 'room2))
   door1-open?   (not door1-open?))
```

Note that within the fexpr (not door1-open?), the door1-open? refers to the state of the door prior to the action. So the effect of the toggle1 action is similar to an assignment:

```
X := not(X)
```

If the state of the doors is represented by a list of all the open doors, the action to open door1 can be defined like this:

```
(define-action open1!
   #:prereq     (or (eq? rob-loc 'room1) (eq? rob-loc 'room2))
   open-doors   (if (memq 'door1 open-doors)
                    open-doors
                    (cons 'door1 open-doors)))
```

Again the `open-doors` within the fexpr refers to the state just before the action. This version checks to see if the door is already open before adding it to the list. The alternative is to imagine an open action that can only be performed when the door is closed initially:

```
(define-action open1!
   #:prereq   (and (not (memq 'door1 open-doors))
                   (or (eq? rob-loc 'room1) (eq? rob-loc 'room2)))
   open-doors (cons 'door1 open-doors))
```

In this case, the action would fail if attempted in a state where the door was already open.

3.3.1 Parameterized actions

If the world under consideration has a number of doors, rooms, and boxes, it would be tedious and error-prone to define a series of actions each of which changes only a single door or a single object location. What is needed, of course, is a parameterized action. The more general form of an action definition is this:

`(define-action` (*name* $v_1 \ldots v_k$)
`#:prereq` *fexpr*
fluent$_1$ *fexpr*$_1$
...
fluent$_n$ *fexpr*$_n$)

where the v_i are variables that may appear in the fexprs for the action. For example, an action can be defined to open an arbitrary door d. Suppose the state of the doors is represented by a list of open doors and that the robot can open any closed door (by remote control, say) regardless of where the robot is located. This can be represented as follows:

```
(define-action (open! d)
   #:prereq     (not (memq d open-doors))
   open-doors   (cons d open-doors))
```

To handle the constraint that the robot must be located in a room that is on one side or the other of the door, something like the following might be used:

```
(define-action (open! d)
   #:prereq   (and (not (memq d open-doors))
                   (connected? d (hash-ref location 'rob)))
   open-doors (cons d open-doors))
```

Here, the fexpr `(hash-ref location 'rob)` is being used to refer to the current location of the robot, and `connected?` is assumed to be a predicate (defined elsewhere) that tests if a door connects to a room. (Recall that the topology of the world is considered to be static.) A representation where each door state is a separate fluent is somewhat awkward for an action that opens an arbitrary door:

```
(define-action (open! d)
   door1-state (if (eq? d 'door1) 'open door1-state)
   door2-state (if (eq? d 'door2) 'open door2-state)
   door3-state (if (eq? d 'door3) 'open door3-state))
```

This says that the `open!` action affects the value of all three door fluents, but unless the argument *d* is the same as the name of the fluent, the new value is the same as the old.

Finally, to see an example of a more complex action, here is a representation of a generic action that pushes a box *b* into a room *rm*, where locations are represented using a hash-table `location`, and where door states are represented using a list of open doors:

```
;; push box b into room rm
(define-action (push-box! b rm)
   #:prereq (let ((rloc (hash-ref location 'rob)))
              (and (eq? rloc (hash-ref location b))
                   (not (eq? rloc rm))
                   (for/or ((d '(door1 door2 door3)))
                      (and (memq d open-doors)
                           (connected? d rloc) (connected? d rm)))))
   location (hash-set* location 'rob rm b rm))
```

The prerequisite for this action checks that the robot and the box *b* start in the same room, that this room is not the same as the destination room *rm*, and that there is an open door connecting the two rooms. The only fluent affected by the action is the `location` hash-table, but the action uses `hash-set*` to change the location of both the robot and the box.

* 3.3.2 Fluents with functional values

As noted above, any Scheme datum can be used as the value of a fluent. This means that we can have fluents whose values are *functions*. Here is an example:

```
(define-fluents
   open?      (lambda (d) (not (eq? d 'door1)))
   location   ...)
```

Here the `open?` fluent represents what is known about the state of all the doors, but this time as a function that returns `#f` for `door1` and `#t` for anything else. So, for example, the fexpr `(open? 'door2)` would have value `#t`, while `(and-map open? '(door1 door3))` would have value `#f`.

With a representation like the one for `open?`, it is still possible to consider an action that opens `door1`. Here is how such an action might be defined:

```
(define-action open1!
   #:prereq   ...
   open?      (lambda (d) (or (eq? d 'door1) (open? d))))
```

This action changes the `open?` fluent and gives it a new value. As usual, the variable `open?` within the fexpr refers to the value of the fluent before the action. So the new value of the `open?` fluent after the action is a function that returns true only if the door is `door1` or a door that was already open prior to the action. Generalizing this idea, a more generic action `open!` that opens an arbitrary door *d* can then be defined as follows:

```
(define-action (open! d)
   #:prereq (not (open? d))
   open?    (lambda (d*) (or (eq? d* d) (open? d*))))
```

3.4 Building a basic action theory

Putting all the pieces together, a complete basic action theory for the world depicted in Figure 3.1 is shown in Figure 3.2. There are a few programing points to notice.

3.4.1 Using abstract functions and predicates

A BAT like the one in Figure 3.2 may end up being used in a much larger *ERGO* program. So it is always a good idea to avoid "magic" constants like the list `(door1 door2 door3)` as much as possible within *ERGO* programs. It is better to use global variables (like the `all-doors` defined here) whose values can be adapted as necessary.

When it comes to the properties of the world, either static ones (like the connectivity of the rooms) or dynamic ones (like the locations of the boxes), it is almost always best to think of these in terms of *functions* and *predicates* in a way that abstracts from how the relevant information is actually represented.

For example, consider the open and closed state of the doors. In the BAT of Figure 3.2, this is represented by a fluent whose value is a list of the currently open doors. To test if a given door *d* is open, we could conceivably use the fexpr `(memq d open-door-list)` in *ERGO* programs. However, if we were to later change our mind and use a hash table representation instead (for some of the reasons discussed later in Section 3.6), we would have to go through the entire *ERGO* program and change the fexpr everywhere to something like `(hash-ref open-door-table d)` instead.

It is much clearer and less error-prone to first define a more *abstract predicate* like `(open? d)` that will then be used in *ERGO* programs:

```
(define (open? d) (memq d open-door-list))
```

Using an abbreviation like this (as in the BAT of the figure) has the advantage of isolating the *ERGO* program from the details of the fluent representation. Changing the representation then requires changing only the definition of this predicate and those actions that affect the fluent. Of course, we still have the option of defining an *abstract function* like

```
(define (open-doors) open-door-list)
```

should this turn out to be useful in the *ERGO* programs.

Similarly, consider the locations of the objects in rooms. The relevant abstract predicate in this case is whether a given object is in a given room, and the two abstract functions that might be considered are the list of objects that are located in a given room, and the room a given object is located in. Some or all of these may be useful in *ERGO* programs. In the BAT of Figure 3.2, only the last of these abbreviations is used:

```
(define (location o) (hash-ref location-table o))
```

Figure 3.2: Program file `Examples/house-bat.scm`

```
;;;  This is a specification of a world involving four rooms,
;;;  three doors, two boxes, and a robot.

;; the static part of the world
(define robot 'rob)
(define all-doors '(door1 door2 door3))
(define all-rooms '(room1 room2 room3 room4))
(define all-objects (cons robot '(box1 box2)))
(define door-connections               ; room topology
   (hasheq 'door1 '(room1 room2) 'door2 '(room2 room4) 'door3 '(room2 room3) ))

;;  the dynamic part of the world: two fluents
(define-fluents
   open-door-list  '(door2 door3)      ; a list of open doors
   location-table                      ; a hash-table of object locations
        (hasheq 'box1 'room1  'box2 'room4  'rob 'room1) )

;;  useful abbreviations
(define (connected? d rm) (memq rm (hash-ref door-connections d)))
(define (open? d) (memq d open-door-list))
(define (location o) (hash-ref location-table o))
(define (ok-door? d rm1 rm2)
   (and (open? d) (connected? d rm1) (connected? d rm2)))
(define (traversable? rm1 rm2)
   (and (not (eq? rm1 rm2)) (for/or ((d all-doors)) (ok-door? d rm1 rm2))))

;;  three actions
(define-action (open! d)               ; open a door
   #:prereq (and (not (open? d)) (connected? d (location robot)))
   open-door-list (cons d open-door-list))

(define-action (goto! rm)              ; go to a room
   #:prereq (traversable? (location robot) rm)
   location-table (hash-set location-table robot rm))

(define-action (push-box! b rm)        ; push a box into a room
   #:prereq (and (eq? (location robot) (location b))
                 (traversable? (location robot) rm))
   location-table (hash-set* location-table b rm robot rm))

;; to display the state of the world
(define (show-house)
   (printf "The open doors are ~a.\n" open-door-list)
   (printf "Box1 is in ~a. Box2 is in ~a. Rob is in ~a.\n"
           (location 'box1) (location 'box2) (location 'rob)) )
```

The main point again is to define abstract predicates and functions as necessary, and to avoid as much as possible concrete fluents like `location-table` in *ERGO* programs.

Once the fluents, the abbreviations, and the three actions are defined, a function called `show-house` is defined. For debugging purposes, it is always a good idea to write a function that displays all the values of the fluents in some convenient form.

3.4.2 Testing a basic action theory

There are some special *ERGO* functions that can be used to test the workings of a BAT in an interactive Scheme session: `legal-action?`, `change-state`, `save-state-excursion`, and `display-execution`. These functions are useful for debugging and are typically not used in working *ERGO* programs.

After loading a BAT into Scheme, the first thing to observe is that each fluent and each primitive action ends up as a global variable. For example, we can define a function that returns the current location of the robot:

```
> (define (rob-loc) (location 'rob))
> (rob-loc)
'room1
```

The `show-house` function defined in the BAT can be used to display all the fluent values:

```
> (show-house)
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.
```

Of course there is more than one state of the world that is of interest. States other than the initial one are the result of performing actions. The actions that take arguments end up as globally-defined functions that evaluate their arguments normally:

```
> (let ((x '(door1 door2 door3))) (open! (cadr x)))
'(open! door2)
```

The value returned is always a list whose first element is the action. Before performing an action, the *ERGO* function `legal-action?` can test if its prerequisite is satisfied in a state:

```
> (legal-action? (open! 'door2))
#f                ; false. Door2 cannot be opened because rob is in room1
> (legal-action? (open! 'door1))
'(room1 room2)    ; true. Door1 can be opened since it is close to rob
```

The value returned by `legal-action?` is the value of the prerequisite fexpr of the action. (The list `'(room1 room2)` is returned instead of a simple `#t` here because of the use of `memq` in the function `connected?` defined in the BAT.) To actually change the state of the world with an action, the *ERGO* function `change-state` can be used:

```
> (change-state (open! 'door1))
> (show-house)
The open doors are (door1 door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

> (change-state (push-box! 'box1 'room2))
> (show-house)
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room2.
```

```
> (change-state (goto! 'room1))
> (show-house)
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room1.
```

When testing a BAT, it is often convenient to perform some state changes, and then to restore the state of the world to what it was at the outset. This can be accomplished using `save-state-excursion`. It takes as its arguments some expressions that are evaluated in sequence for their effect. So, for example, starting in the initial state, we have

```
> (show-house)
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

> (save-state-excursion
     (change-state (open! 'door1))
     (change-state (push-box! 'box1 'room2))
     (show-house))
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room2.

> (show-house)
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.
```

It is sometimes useful to track the effects of a sequence of actions on the state of the world. The *ERGO* function `display-execution` takes as its arguments a function of no arguments (used to display states), and a list of actions to perform. For example, using this function starting from the initial state, we get the following:

```
> (display-execution show-house
      '((open! door1) (push-box! box1 room2) (goto! room3)) )

   The first state is:
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

   Executing action (open! door1).   The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

   Executing action (push-box! box1 room2).   The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room2.

   Executing action (goto! room3).   The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room3.

   That was the final state.
```

Like `save-state-excursion`, the `display-execution` function restores the state of the world to what it was just before it was called.

Once a BAT is found to be behaving properly, it can be included as part of an *ERGO* program (as discussed in the next chapter), or used for other purposes, such as for automated planning, which we now turn to.

3.5 Basic automated planning

In its most basic form, automated planning involves finding a sequence of actions that achieves a goal of interest. For example, starting from the initial state depicted in Figure 3.1, a planner might be asked to achieve the goal of having both boxes in Room2. One solution in this case would be the following sequence of actions:

> open Door1;
> push Box1 from Room1 to Room2 through Door1;
> go to Room4 through Door3;
> push Box2 from Room4 to Room2 through Door3.

This solution, of course is not unique; the robot could have gone for Box2 first, or it could have wasted some time by entering and leaving Room3.

To automatically generate a plan like this, the function `ergo-simplan` is used:

> `(ergo-simplan` *goal actions* `[#:prune` *prune*`] [#:loop?` *flag*`])`

The *goal* here is a function of no arguments that should return true in states where the goal has been achieved, and the *actions* is a list of actions previously defined by `define-action` to be considered in achieving the goal. (The two optional arguments will be explained later.) The value of an `ergo-simplan` expression is a shortest list of the actions that solves the planning problem, or `#f` when no plan can be found. A list of actions is a solution to a planning problem if it has two properties:

- each action in the list can be legally performed in the corresponding state (the first action in the initial state, the second action in the state that results from performing the first action, and so on);
- the state that results from performing all the actions in sequence satisfies the goal (that is, the given goal function returns true in that final state).

So, as a simple example, we have the following:

```
> (ergo-simplan
    (lambda () (eq? (location 'rob) 'room3))        ; the goal function
    '((open! door1) (open! door2) (open! door3)     ; the action list
      (goto! room1) (goto! room2) (goto! room3) (goto! room4)))
Plan found after 0 ms.
'((open! door1) (goto! room2) (goto! room3))
```

The goal here is for the robot to be in Room3, and a list of seven actions are given for consideration. The plan to achieve the goal has three actions to be executed in sequence.

As it turns out, the main complication in using the `ergo-simplan` function in practice is often constructing its second argument, the list of actions to consider in the planning. For the BAT above, the following list of all the defined actions might be used:

```
(define all-actions
   (append
      (map open! all-doors)
      (map goto! all-rooms)
      (for/append ((b '(box1 box2)))
        (for/list ((rm all-rooms)) (push-box! b rm)))))
```

This is a list with the seven actions above and eight `push!` actions (for two boxes × four rooms). To define the goal of getting all the objects into Room2, we can use this:

```
(define (my-goal) (for/and ((o all-objects)) (eq? (location o) 'room2)))
```

So `my-goal` is defined as a function of no arguments, as required for a goal. We can use it as the first argument to `ergo-simplan` to find a plan to achieve the goal:

```
> (ergo-simplan my-goal all-actions)
Plan found after 1 ms.
'((open! door1) (push-box! box1 room2) (goto! room4) (push-box! box2 room2))
```

The four step plan returned is the solution. We can examine how this solution works more closely using `display-execution`:

```
> (display-execution show-house (ergo-simplan my-goal all-actions))
Plan found after 1 ms.
    The first state is:
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

    Executing action (open! door1).  The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

    Executing action (push-box! box1 room2).  The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room2.

    Executing action (goto! room4).  The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room4. Rob is in room4.

    Executing action (push-box! box2 room2).  The resulting state is:
The open doors are (door1 door2 door3).
Box1 is in room2. Box2 is in room2. Rob is in room2.

    That was the final state.
```

We can generalize `my-goal` to one of getting all the objects into an arbitrary room:

```
> (define (all-in rm) (for/and ((o all-objects)) (eq? (location o) rm)))
> (ergo-simplan (lambda () (all-in 'room3)) all-actions)
Plan found after 3 ms.
'((open! door1) (push-box! box1 room2) (goto! room4) (push-box! box2 room2)
  (push-box! box1 room3) (goto! room2) (push-box! box2 room3))
```

Figure 3.3: Program file `Examples/PlanningExamples/farmer.scm`

```
;;; The classical fox, hen, and grain problem.
;;;    Fluent locs: the side of the river that the four objects are on
;;;    Action cross-with!: farmer crosses the river with a passenger or alone

;; Initially all four objects are on the left side
(define-fluents locs (hasheq 'farmer 'left 'fox 'left 'grain 'left 'hen 'left))

;; The farmer crosses with x (when x=farmer, then cross alone)
(define-action (cross-with! x)
  #:prereq  (same-side? 'farmer x)
  locs      (let ((other (if (eq? (loc 'farmer) 'left) 'right 'left)))
              (hash-set* locs 'farmer other x other)))

;;  Abbreviations
(define objects '(farmer fox hen grain))
(define (loc x) (hash-ref locs x))
(define (same-side? x y) (eq? (loc x) (loc y)))

;; Goal state: all four objects on the right side
(define (goal?) (for/and ((x objects)) (eq? (loc x) 'right)))

;; Unsafe state: the hen is with the grain or fox without the farmer present
(define (unsafe?)
  (and (or (same-side? 'hen 'grain) (same-side? 'fox 'hen))
       (not (same-side? 'farmer 'hen))))

;;  The main program: display a plan to achieve the goal
(define (main)
  (display-execution
    (lambda () (for ((x objects)) (printf "~a at ~a  " x (loc x))))
    (ergo-simplan goal? #:prune unsafe? (map cross-with! objects))))
```

Note that `ergo-simplan` requires a function of no arguments as its first argument. This is why the `lambda` is needed in this case, but was not needed with `my-goal` above.

3.5.1 The fox, hen, grain problem

As a second planning problem, and a second example of a complete basic action theory, consider the classical fox, hen, grain problem:

> A farmer has a fox, a hen, and a bushel of grain that he wants to transfer from one side of a river to the other using a boat that can carry at most one of them (plus himself) at a time. If the farmer leaves the fox alone with the hen, it will eat the hen; if the hen is left alone with the grain, it will eat the grain.

A representation of this problem as a basic action theory is shown in Figure 3.3.

In this version, the single fluent `locs` is used to represent the locations of the four objects in a hash-table. The action `cross-with!` represents a crossing of the boat by the farmer, with an argument indicating the passenger. (When the farmer crosses alone, the argument is the `farmer` itself.) A boat crossing is not possible when the farmer and the

Figure 3.4: The solution to the fox, hen, grain problem

```
    The first state is:
farmer at left  fox at left  hen at left  grain at left
    Executing action (cross-with! hen).  The resulting state is:
farmer at right  fox at left  hen at right  grain at left
    Executing action (cross-with! farmer).  The resulting state is:
farmer at left  fox at left  hen at right  grain at left
    Executing action (cross-with! fox).  The resulting state is:
farmer at right  fox at right  hen at right  grain at left
    Executing action (cross-with! hen).  The resulting state is:
farmer at left  fox at right  hen at left  grain at left
    Executing action (cross-with! grain).  The resulting state is:
farmer at right  fox at right  hen at left  grain at right
    Executing action (cross-with! farmer).  The resulting state is:
farmer at left  fox at right  hen at left  grain at right
    Executing action (cross-with! hen).  The resulting state is:
farmer at right  fox at right  hen at right  grain at right
    That was the final state.
```

passenger are on opposite sides of the river. The effect of the action is to move both the farmer and the passenger to the other side (using `hash-set*` on the `locs` fluent).

One requirement in this puzzle is that the hen must never be left alone with either the fox or the grain. One way to deal with this would be to change the prerequisite of `cross-with!` to ensure that only crossings that are safe for both the grain and the hen are considered possible. A cleaner way (used in the BAT of Figure 3.3) is to make use of the optional `#:prune` argument of `ergo-simplan`, whose value should be a function of no arguments. The idea is that `ergo-simplan` rejects any plan that attempts to pass through a state where the prune function returns true. So here we need only define `unsafe?` to formalize the safety requirement and use it as the prune argument.

The `main` procedure of the program produces the output shown in Figure 3.4. As can be seen, the farmer first brings the hen to the right side, then returns alone to get the fox, then brings the hen back to the left side, then brings the grain to the right side, and then returns alone to get the hen for the final crossing.

* 3.5.2 The jealous husband problem

As a final and much more challenging example problem, consider the so-called jealous husband problem, which also concerns a boat with limited capacity:

> Three married couples find themselves on a bank of a river wanting to get to the other side using a boat that can only hold two people at a time. However, each husband insists on being with his wife when other men are present.

Figure 3.5: Program file `Examples/PlanningExamples/jealous-bat.scm`

```
;;; The Jealous husband problem:
;;;   Three couples on the left side of the river
;;;   A boat that can hold 1 or 2 people.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Static facts about men, women and sides of river

(define men '(m1 m2 m3))                      ; 3 men
(define women '(w1 w2 w3))                    ; 3 women
(define people (append men women))            ; 6 people
(define (husband w) (case w ((w1) 'm1) ((w2) 'm2) ((w3) 'm3)))
(define (oppside x) (if (eq? x 'left) 'right 'left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Fluents and actions

;; two fluents: location of the boat and location of the 6 people
(define-fluents
  boat        'left
  people-loc  (hasheq  'm1 'left  'm2 'left  'm3 'left
                       'w1 'left  'w2 'left  'w3 'left))

;; a useful abbreviation
(define (loc p) (hash-ref people-loc p))

;; cross the river with one person in the boat
(define-action (cross-single! p1)
  #:prereq   (eq? boat (loc p1))
  boat       (oppside boat)
  people-loc (hash-set people-loc p1 (oppside boat)))

;; cross the river with two people in the boat
(define-action (cross-double! p1 p2)
  #:prereq   (and (eq? boat (loc p1)) (eq? boat (loc p2)))
  boat       (oppside boat)
  people-loc (let ((opp (oppside boat))) (hash-set* people-loc p1 opp p2 opp)))
```

A basic action theory for this is shown in Figure 3.5. The fluents and actions in this BAT are similar to those for the fox, hen, and grain problem, where the location of the boat is kept separate from the location of the six people. The goal will be the same: get all the objects from the left bank to the right bank. A complete program that loads this BAT and uses `ergo-simplan` to find a plan is shown in Figure 3.6. (To keep the files small, two files are used, one for the BAT, and one for the main program.) The jealousy constraint plays a role like the safety constraint from before. In this case, a state is considered "bad" if some woman is not on the same bank as her husband but is on the same bank as another man.

Much of the difficulty in actually solving this problem is due to the large number of actions to consider. The global variable `all-double-crossings` is defined as the list of all `cross-double!` actions with two people such that `can-boat?` is true and where the second person `p2` in the boat occurs later in the list of people than the first person `p1`. (The `can-boat?` predicate is used to ensure that we do not consider a boating trip involving a

Figure 3.6: Program file Examples/PlanningExamples/jealous-main.scm

```
;;; The Jealous husband problem:
;;;   Three couples on the left side of the river
;;;   A boat that can hold 1 or 2 people.
;;;   The goal: get everyone to the right side of the river.
;;;   Constraint: no woman can be with a man unless her husband is present

(include "jealous-bat.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   Goal state, bad state, and the list of all allowed actions

;; a goal state is one where all the people are on the right side
(define (goal?) (for/and ((p people)) (eq? (loc p) 'right)))

;; a bad state is one that violates the above jealousy constraint
(define (bad?)
  (for/or ((w women)) (and (not (eq? (loc w) (loc (husband w))))
                           (for/or ((m men)) (eq? (loc w) (loc m))))))

;; all single-person crossings
(define all-single-crossings (map cross-single! people))

(define (can-boat? p1 p2)                    ; p1 and p2 can boat together?
  (and (or (memq p1 men) (memq p2 women) (eq? p2 (husband p1)))
       (or (memq p2 men) (memq p1 women) (eq? p1 (husband p2)))))

;; all double-person crossings where the first person occurs before the
;; second in the list of people and where the two can boat together
(define all-double-crossings
  (for/append ((p1 people))
    (for/list ((p2 (for/only ((p2 (cdr (memq p1 people)))) (can-boat? p1 p2))))
      (cross-double! p1 p2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   The main program

(define (main)
  (ergo-simplan goal? (append all-single-crossings all-double-crossings)
           #:loop? #t
           #:prune bad?))
```

woman and a man who is not her husband. The memq is used to ensure that we do not consider pairs of people twice.)

Even with these restrictions, the space of potential plans is still huge. It will take ergo-simplan several seconds to find a plan even on a fast computer. A big part of the problem is that the planner will consider actions that undo previous actions, for example, a person crossing the river alone and then immediately returning.

To deal with this, the #:loop? optional argument to ergo-simplan can be used. When this *flag* is true, the planner rejects any plan that would pass through the same state twice. With this, we quickly get the following output, showing the required eleven actions:

```
Plan found after 72 ms.
'((cross-double! m1 w1) (cross-single! m1) (cross-double! w2 w3)
```

```
(cross-single! w1) (cross-double! m2 m3) (cross-double! m2 w2)
(cross-double! m1 m2) (cross-single! w3) (cross-double! w1 w2)
(cross-single! m3) (cross-double! m3 w3))
```

(Not to take this example too seriously or anything, but we might prefer a solution where the jealous husbands all end up on one side of the river without a boat, and the women have all gone off to find better husbands!)

3.6 Efficiency considerations

When writing a basic action theory for *ERGO*, how should a programmer decide what representation to use for the fluents? To represent knowledge about doors that may be open or closed, for example, is it best to use a list of open doors, an association list, a bit vector, a hash-table, a Scheme predicate, or something else?

In general, there is no easy answer to this question. When there are only a few doors, it does not really matter. A simple list representation is fine. Although the entire list may need to be scanned to determine the state of a door, the lists will be short. But as the number of objects in the world increases, the vector and hash-table representations are to be preferred since they allow constant time access to their elements.

In general, we can also distinguish between representations that are progressive and those that are regressive.

A *progressive representation* is one where the properties of objects are as easily computed after a large number of actions have been performed as they were in an initial state. For example, the hash-table representation of the locations of objects is progressive. No matter how many actions have taken place, a hash-table makes the current location of each object available as readily as it was in the initial state. The tradeoff is that each time the location of an object changes, a new hash-table must be constructed by copying some part of the old one. (Another example of a progressive representation is the one that manages doors using the fluent list `open-doors`.)

A *regressive representation* is one where this copying is not done, but where it may take more time to compute the properties of objects after a large number of actions have been performed. For example, suppose the locations of objects are represented using an association list, that is, a list of pairs (*obj loc*). When an object moves to a new location, one possibility is to insert a new pair at the front of the association list leaving the rest of the list unchanged. So the list does not need to be copied. The tradeoff is that after a large number of such actions, the list itself will become large and obtaining the location of an object can involve going through the entire list. (Another example of a regressive representation is the one that managed doors using the predicate fluent `open?` in Section 3.3.2.)

In general, for robots that are expected to be long-lived and perform a large number of actions, a progressive representation is the best. For robots with a short life or that live in a world with a very large number of changeable objects, a regressive representation might work better.

Another dimension worth noting is how global or local the fluents are. At one extreme, there are local fluents each of whose value is a single symbol. An example of a local representation was one where each door and each object was a different fluent. The advantage of these representations is that little copying is required and each action affects only a small

part of the overall state. The disadvantage is that the representation for parameterized actions can be cumbersome.

At the other extreme, there might be a single fluent that represents all aspects of the changing world. For example, we might use a single hash-table `state` that maps doors to their open/closed status and objects to their locations. The advantage is that generic actions are easily defined to change any aspect of the state. In fact, there can be a single action `change!` that updates any aspect of the state according to its arguments. The disadvantage is that the entire state would need to be copied when anything changes. (This copying can be reduced by using a more regressive representation, but that would lead to troubles of its own.)

Between these two extremes, there are representations like the one presented above, where all the door states are represented by one fluent and all the object locations are represented by a second fluent. This is a progressive representation, and should it lead to too much copying after an action (because there are too many doors or too many boxes), the solution would be to partition the boxes or the doors in some way and use more than one fluent for each.

* 3.7 How this all works

It is not necessary to understand the implementation details regarding fluents, states, and actions to be able to use the planner or the *ERGO* programming language of the next chapter. For completeness, however, the implementation is described in this section.

In the implementation, fluents are global Scheme variables. In addition, when the function `define-fluents` is used, a list of these fluent names is stored internally. A function `(current-state)` is used to make a vector of the current fluent values using this list, and a function `(restore-state` *vec*`)` is used to reassign the fluent to the values given by the vector *vec*. (The function `save-state-excursion` is defined in terms of these two.)

Actions are handled using three special hash-tables: `prereq-table`, `effect-table`, and `sensing-table`. (In this chapter, the sensing part was not used.) When an action is defined using `define-action`, entries are added to these tables. So, in the simple case where the action takes no argument, evaluating `(define-action` *name* `#:prereq` *pre* `...)` causes the following to take place:

```
(hash-set! prereq-table name (lambda () pre))
```

The function `legal-action?` works by looking for this function in the `prereq-table` for the given action and calling it.

The state change produced by an action is handled in a similar way. In the simplest case, `(define-action` *name fluent fexpr*`)` causes the following:

```
(hash-set! effect-table name (lambda () (set! fluent fexpr)))
```

So the function associated with the action changes the state to a new one where some of the fluents have new values according to the given fexprs. The function `change-state` works by looking for this function in the `effect-table` for the given action and calling it.

Turning now to the planner, the code for the `ergo-simplan` procedure is shown in Figure 3.7. It works by what is called *iterative deepening*. The idea is that `ergo-simplan`

Figure 3.7: A simple iterative deepening planner

```
(define (ergo-simplan goal acts #:steps [steps 30]
                  #:loop? [loop? #f] #:prune [prune (lambda () #f)])
  (define (simp* n h l)
    (define cur (current-state))
    (if (= n 0) (and (goal) (reverse h))
        (and (not (and loop? (member cur l))) (not (prune))
             (for/or ((a acts))
               (and (legal-action? a)
                    (save-state-excursion
                      (change-state a)
                      (simp* (- n 1) (cons a h) (if loop? (cons cur l) l))))))))
  (let loop ((n 0)) (or (simp* n '() '()) (and (< n steps) (loop (+ n 1))))))
```

first searches for a plan of length 0; failing this, it searches through all plans of length 1; failing this, it searches through all plans of length 2, and so on up to a bound (which is a third optional argument to `ergo-simplan`). It does this by calling a local function `simp*` repeatedly within a loop with an increasing value of n. Although it may seem wasteful on failing to find a plan of length n, to search from scratch for a plan of length $(n+1)$, it has been found in practice that the time spent looking for plans of length n or less is actually quite small compared to the time required to go through the plans of length $(n+1)$.

The `simp*` procedure within `ergo-simplan` does all the work. It receives as arguments the number of actions n to consider in a plan, the history h of actions performed so far, and a list l of the states encountered so far (in case the `#:loop?` parameter is used). If there are no actions to consider (that is, $n = 0$), it simply checks if the goal condition is currently satisfied and if so, returns h in reverse order. Otherwise, after checking the `#:loop?` and `#:prune` conditions, it tries to find (using `for/or`) an action in the given list of actions whose prerequisite is currently satisfied (using `legal-action?`) and such that `simp*` is satisfied in the state that results from performing that action (using `change-state`), with $(n-1)$ actions left to consider.

3.8 Exercises

1. Use `ergo-simplan` in the `house-bat` domain to find the shortest sequence of robot actions that gets two boxes into the same room.

2. Write a solution to the fox, hen, grain problem that does not use the `#:prune` optional argument, but instead uses a modified prerequisite for the `cross-with!` action as suggested in the text.

3. An early test domain for planning programs was the *blocks world*, where a robot would manipulate blocks on a tabletop with actions like picking up a block (off the table or off another block) and putting a block down (on the table or onto another block). A planning problem was specified in terms of an initial and a final layout of blocks, looking for minimal sequences of actions that transformed one to the other. Read up on this domain and formalize some parts of it in *ERGO*.

4. Formalize the Towers of Hanoi problem as a planning problem in *ERGO*. You may assume that the number of disks n is provided at the outset (unlike the version of the problem considered in Chapter 12). The shortest solution will take $2^n - 1$ moving actions. The well-known recursive program for this problem will find and print the actions in about 2^n time units (that is, its running time will scale linearly in the size of the plan). Investigate how long `ergo-simplan` takes to find a plan.

Chapter 4

From Planning to Programming

In this chapter, we turn our attention to *ERGO* programs. Running an *ERGO* program is not so different from the planning seen in Chapter 3. In both cases, the system is given an initial state of the world and some objective to be achieved. For planning, the objective is expressed as a goal condition to be made true using actions taken from a given list:

`(ergo-simplan` *goal actions*`)`

For programming, the objective will be expressed as an *ERGO* program to be executed:

`(ergo-do` *program*`)`

In both cases, the desired output will be the same: a sequence of actions that achieves the objective (or `#f` when no such sequence exists). The difference is that with a program, we get to consider not only *what* is to be achieved, but also *how* we expect it to be achieved. In *ERGO*, this can range from an explicit list of actions to perform, all the way to some guidelines about what should or should not be done.

This chapter explores how basic *sequential programs* can be defined, that is, programs where only one thing is happening at a time. It elaborates further on why it is useful to go beyond planning to programming. The more advanced *concurrent programs*, where more than one thing is happening at a time, are considered in Chapter 5.

4.1 Deterministic programming

This section deals with sequential programs that are fully *deterministic*, that is to say, where programs can be executed in just one way. These are the programs that are most like those seen in ordinary imperative programming languages.

4.1.1 The `:act` and `:begin` primitives

The most basic programming primitives in *ERGO* are `:act` and `:begin`. (Note that the programming primitives in *ERGO* all begin with a colon.) If *e* is an fexpr whose value is an action, then

`(:act` *e*`)`

is an expression whose value is an *ERGO* program, which can then be used as an argument to `ergo-do`. Returning to the example BAT of Figure 3.2, the following interaction can be observed:

```
> (ergo-do (:act (open! 'door1)))
'((open door1))
```

In this case, `ergo-do` is asked to find a sequence of actions that constitutes a successful execution of the given `:act` program, and it finds the obvious list with one element. (After execution, `ergo-do` restores the state to what it was at the outset.) On the other hand, the following program fails:

```
> (ergo-do (:act (goto! 'room2)))
#f
```

In this case, no sequence of actions can be found by `ergo-do` (since the door to Room2 is closed in the initial state).

To obtain a sequence of actions, the `:begin` primitive is used. If p_1, p_2, ... p_n are expressions that evaluate to *ERGO* programs, then

`(:begin` p_1 ... p_n`)`

is an expression that evaluates to the *ERGO* program that executes each subprogram in order:

```
> (ergo-do (:begin (:act (open! 'door1)) (:act (goto! 'room2))))
'((open! door1) (goto! room2))
> (ergo-do (:begin (:act (open! 'door1)) (:act (goto! 'room3))))
#f
> (ergo-do
     (:begin
         (:act (open! 'door1))
         (:begin (:begin) (:begin (:act (goto! 'room2))))
         (:act (goto! 'room1))  ))
'((open! door1) (goto! room2) (goto! room1))
```

As can be seen in the second example above, a `:begin` program fails (and `ergo-do` returns #f) if any of its subprograms fail. In this case, the robot cannot go directly from Room1 to Room3. The `(:begin)` that appears in the third example above is a program that does nothing. (The primitive `:nil` can be also used for this purpose.)

As with `ergo-simplan`, the sequence of actions returned by `ergo-do` can be used as an argument to the `display-execution` function, as shown in Figure 4.1.

4.1.2 Programs versus actions

There is a distinction in the *ERGO* language between programs and actions: while the form `(:act (open! 'door1))` evaluates to a *program* and can be used as an argument to `ergo-do`, the form `(open! 'door1)` evaluates to an *action* and cannot be used in this way. The `:begin` primitive (like most of the other language primitives to follow) expects to see *ERGO* programs as its arguments, and will produce an error otherwise:

Figure 4.1: Executing a sequence of actions

```
> (display-execution show-house
     (ergo-do (:begin (:act (open! 'door1)) (:act (goto! 'room2))) ))

   The first state is:
The open doors are (door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

   Executing action (open! door1).  The new state is:
The open doors are (door1 door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room1.

   Executing action (goto! room2).  The new state is:
The open doors are (door1 door2 door3).
Box1 is in room1. Box2 is in room4. Rob is in room2.

   That was the final state.
```

```
> (ergo-do (:begin (:act (open! 'door1)) (:act (goto! 'room2))))
'((open! door1) (goto! room2))
> (ergo-do (:begin (open! 'door1) (:act (goto! 'room2))))
procedure application: expected procedure, given: '(open! door1)
```

The error reported here is that `:begin` was given something other than an *ERGO* program as its first argument.

4.1.3 Using Scheme `define` with *ERGO* programs

Any Scheme form whose value is an *ERGO* program can be used wherever such a program is required. For example, we have this:

```
> (let ((pgm (:begin (:act (open! 'door1)) (:act (goto! 'room2)))))
    (ergo-do pgm))
'((open! door1) (goto! room2))
```

This means that the Scheme `define` can also be used. For example, to define a new *ERGO* procedure where the robot goes to some room and then back to Room1, we can use this:

```
(define (goback rm) (:begin (:act (goto! rm)) (:act (goto! 'room1)) ))
```

So `goback` is a function that returns an *ERGO* program as its value. We then get this:

```
> (ergo-do (:begin (:act (open! 'door1)) (goback 'room2) (goback 'room2)))
'((open! door1) (goto! room2) (goto! room1) (goto! room2) (goto! room1))
```

4.1.4 The `:if` and `:test` primitives

The *ERGO* `:if` primitive is used to execute a program conditionally. The expression

`(:if` e p q`)`

evaluates to a program that executes p if the fexpr e is true, and q otherwise.

```
> (ergo-do (:if (open? 'door1) (:act (goto! 'room2)) (:act (open! 'door1))))
'((open door1))
```

(See below for the difference between the *ERGO* `:if` and the Scheme `if`.) There are some additional primitives that expand to versions of `:if`: the primitive

`(:when` e p_1 ... p_n`)`

behaves just like `(:if` e `(:begin` p_1 ... p_n`) :nil)`, and the primitive

`(:unless` e ...`)`

is an abbreviation for `(:when (not` e`)` ...`)`.

The primitive

`(:test` e`)`

succeeds or fails according to whether or not the fexpr e evaluates to true. In sequential programming, it is equivalent to `(:if` e `:nil :fail)`, where `:fail` is the (rarely useful) *ERGO* program that always fails.

4.1.5 The `:while` and `:for-all` primitives

To perform iteration, the primitive

`(:while` e p_1 ... p_n`)`

can be used. This behaves just like `(let loop () (:when` e p_1 ... p_n `(loop)))`, that is, repeatedly perform the p_i in sequence until the fexpr e evaluates to false. Note that this program will fail (and `ergo-do` will return `#f`) if any of the p_i fail before e is false. The primitive

`(:until` e ...`)`

is an abbreviation for `(:while (not` e`)` ...`)`.

The second primitive for iteration is

`(:for-all` v e p_1 ... p_n`)`,

where v is a symbol, e is an fexpr that evaluates to a list, and the p_i evaluate to subprograms. This is executed by executing the p_i in sequence for variable v assigned to each element of e. So, for example,

```
(:for-all x '(room1 room2 room3) (get-to-room x))
```

behaves just like

```
(:begin (get-to-room 'room1) (get-to-room 'room2) (get-to-room 'room3))
```

performing a (user-defined) *ERGO* procedure `get-to-room` for each of the three rooms. For convenience, the e can also evaluate to a number n in which case the v is assigned to the numbers $0, 1, \ldots, n-1$.

Another way to give a variable a value in *ERGO* is by using `:let` which is the *ERGO* version of the Scheme `let` special form.

∗ 4.1.6 Evaluation versus execution

Expressions such as (:begin p_1 ... p_n) or (:act a) evaluate to *ERGO* programs, but do not actually execute those programs until they are passed to ergo-do. This distinction between when an expression is evaluated and when an *ERGO* program is executed accounts for the difference between the *ERGO* :if and the Scheme if (and between :let and let).

The expression (:if e q r) evaluates to a conditional program which, when executed, chooses between q and r based on the value of e at the time of execution. The expression (if e q r), on the other hand, evaluates to q or to r based on the value of e at the time of evaluation. In many cases, the two expressions behave the same and can be used interchangeably. However, there can be a difference between the two when the e refers to fluents whose values will be changed in the execution.

Suppose, for example, that Door1 is closed initially. Then the expression

```
(:begin (:act (open! 'door1)) (:if (open? 'door1) q r))
```

evaluates to a program that opens the door and then performs q (which is what would be expected in normal sequential execution). On the other hand, the expression

```
(:begin (:act (open! 'door1)) (if (open? 'door1) q r))
```

evaluates to a program that opens the door and then performs r (which is almost certainly not what was intended). The problem is that (open? 'door1) will be false when the :begin expression is evaluated, but true after the first action is executed.

The moral is that when the normal Scheme evaluation requires the value of a fluent, we need to be careful about when the fluent will be evaluated. In almost all cases, we will want to use :if and :let in *ERGO* programs instead of their Scheme counterparts. Instead of a program like (:begin p (my-function f)) (where f is a fluent), we will want to use something more like (:begin p (:let ((x f)) (my-function x))) in cases where the *ERGO* program p can change the value of f.

4.2 Nondeterministic programming

From a programming perspective, the main difference between *ERGO* and ordinary imperative programming languages is the possibility of *nondeterminism*. A program is nondeterministic when it can be executed successfully in more than one way. It is then the job of the *ERGO* processor to search for one such execution, that is, to find a sequence of actions that constitutes one of the successful executions of a program.

Nondeterminism makes it possible to write *high-level programs*, where many of the details are left to the *ERGO* system to determine depending on what else needs to be done. For example, we can say something like "exit the room and go to the left or right as appropriate to enter into the kitchen" or "pick up one of the packages as needed and take it into the next room." Nondeterministic programs are assumed to have *choice points* like this in them, and the *ERGO* processor must find the right choice that ensures a successful completion of the entire program.

4.2.1 The :choose primitive

The simplest nondeterministic primitive is

(:choose p_1 ... p_n),

where the p_i evaluate to subprograms. Conceptually, a :choose program is executed by selecting one of those subprograms and executing it. The one to be executed is chosen nondeterministically, depending on what else needs to be done.

To see this in action, imagine a scenario where there are four actions, a, b, c, and d, where d alone has a prerequisite which is false initially, and where the actions a and c have no effect, but where the action b makes the prerequisite of action d true. The program

```
(:choose (:act a) (:act b) (:act c))
```

is executed by executing one of the three actions. By default, ergo-do simply returns the first successful execution it can find:

```
> (ergo-do (:choose (:act a) (:act b) (:act c)))
'(a)
```

This mode of execution is most useful for deterministic programs. By using the optional #:mode argument with the value 'offline, all three executions can be seen:

```
> (ergo-do #:mode 'offline (:choose (:act a) (:act b) (:act c)))

(a)
OK? (y or n)  n

(b)
OK? (y or n)  n

(c)
OK? (y or n)  n
#f
```

In this 'offline mode, ergo-do displays a sequence of actions and then asks for confirmation from the user. If the user enters y, that sequence is returned as the value of the entire expression; if the user enters n, ergo-do searches for another execution of the program and then asks again. A value of #f is returned only when no further executions can be found. (The default behaviour of ergo-do when no mode is specified is called 'first mode. There is also a third 'count mode that returns the total number of successful executions. For example,

```
> (ergo-do #:mode 'count
    (:begin
       (:choose (:act a) (:act b) (:act c))
       (:choose (:act a) (:act b) (:act c))
       (:choose (:act a) (:act b) (:act c))
       (:choose (:act a) (:act b) (:act c)) ))
81
```

Here the program has $3 \times 3 \times 3 \times 3 = 81$ possible executions.)

Search enters the picture when some of the nondeterministic choices may lead to a failure later in the program. For example, the program

```
(:begin
  (:choose (:act a) (:act b) (:act c))
  (:act d) )
```

has just one execution: the sequence consisting of the action `b` followed by the action `d`. The other two executions, involving the actions `a` and `c`, must be discarded since they do not allow the final `d` action to be executed successfully. Similarly,

```
(:begin
  (:choose (:act a) (:act b) (:act c))
  (:choose (:act a) (:act b) (:act c))
  (:act d) )
```

has exactly five executions: the four executions without a `b` action must be discarded since they do not allow the rest of the program to complete successfully.

4.2.2 Backtracking search

There are two ways to think of this `:choose` primitive. For small enough problems, it is sufficient to imagine *ERGO* as magically choosing the correct subprogram to execute. In the examples above, the program just knows that action `b` is the correct choice needed later.

For larger problems, we have to think of `:choose` in terms of search. The `:choose` attempts the first subprogram, and if the rest of the program can be executed successfully, it stops there; if it cannot, *ERGO* backtracks to the state it was in at the time of that choice, but this time it attempts the second subprogram; this process continues until either one of the subprograms leads to success, or `:choose` runs out of options, in which case it fails.

Note that this backtracking search is depth-first. Consider a program like the following:

`(:begin (:choose` p_1 p_2`) (:choose` q_1 q_2`)` r`)`

In this case, p_1 followed by q_1 is considered first. If program r then fails, backtracking reconsiders the second `:choose`, so that p_1 followed by q_2 ends up being considered next. If r fails again, the second `:choose` runs out of options and fails. At that point, backtracking reconsiders the first `:choose`, which then tries its second option, and so p_2 followed by q_1 is considered next, and finally p_2 followed by q_2, if necessary. If none of these allow r to succeed, the entire `:begin` program fails.

4.2.3 The :>> and :<< primitives

To help follow the backtracking execution of an *ERGO* program, there are two special *ERGO* primitives:

`(:>>` e_1 ... e_n`)`

and

`(:<<` e_1 ... e_n`)`,

where the e_i are fexprs. Both of these programs always succeed, but evaluate their e_i for effect. They can therefore be used to display debugging information. They differ in the following way: `:>>` evaluates the e_i when the program is executing normally, but `:<<` evaluates the e_i only when the program is backtracking as the result of a failure. So, for example, the program

```
(:begin
   (:<< (printf "Out of options\n"))
   (:>> (printf "Starting\n"))
   (:choose :nil :nil)
   (:>> (printf "So far so good\n"))
   (:<< (printf "Failing\n"))
   :fail)
```

produces the following output and then fails:

```
Starting
So far so good
Failing
So far so good
Failing
Out of options
```

ERGO also provides two additional primitives, `::act` and `::test` which behave just like `:act` and `:test` except that they use `:>>` and `:<<` to display information on success and on failure for debugging purposes.

4.2.4 The `:for-some` primitive

The second nondeterministic primitive in *ERGO* is

`(:for-some` v e p_1 ... p_n`)`,

where v is a symbol, e is an fexpr that evaluates to a list, and the p_i evaluate to subprograms. Conceptually, the `:for-some` is executed by setting a variable v to one of the elements of the list denoted by e, and then executing all the subprograms in sequence. The element of the list is chosen nondeterministically and depends on what else needs to be done. As with `:for-all`, the e can also evaluate to a number n, in which case the subprograms must succeed for one of the values $0, 1, \ldots, n-1$.

The `:for-some` primitive can be thought of as the dual of the `:for-all` primitive: whereas a `:for-all` program can be executed successfully when the subprograms can be executed successfully for *all* values of its variable, a `:for-some` program can be executed successfully when the subprograms can be executed successfully for *some* (at least one) value of its variable. (Similarly, the `:choose` primitive can be thought of as the dual of `:begin`.) So, for example, the program

```
(:for-some d '(door1 door2 door3) (:act (open! d)))
```

behaves exactly like

```
(:choose (:act (:open! 'door1))
         (:act (:open! 'door2))
         (:act (:open! 'door3)))
```

with three possible executions (assuming each door can be opened). Similarly,

```
(:for-some a act-list (:act a) (:act a) (:act a) (:act a))
```

will have as many executions as there are actions in the list `act-list` that can be legally performed four times. (This idea of nondeterministically choosing an action from a list of actions will allow a convenient form of planning, as discussed in Section 4.4.)

4.2.5 The `:star` primitive

The final nondeterministic primitive in *ERGO* is

`(:star` p_1 ... p_n`)`,

where the p_i evaluate to subprograms. Conceptually, a `:star` program is executed by repeatedly executing all the subprograms in sequence some number of times. The number of times is chosen nondeterministically depending on what else needs to be done, and can be zero.

So, for example, the program

```
(:star (:for-some obj object-list
          (:test (on-floor? obj))
          (:act: (pickup! obj))  ))
```

picks up some number of objects that are on the floor. (The `:test` is not necessary if the `pickup!` action has a prerequisite that the object be on the floor.) It is left unspecified by this program which objects are to be chosen or even how many are to be picked up. In terms of backtracking, `:star` first attempts to do its subprograms zero times, and if that later fails, then once, and if that fails, then twice, and so on.

The `:star` primitive can be duplicated using recursion and other features of *ERGO*:

`(let loop ((n 0)) (:choose (:for-all i n` p_1 ... p_n`) (loop (+ n 1))))`

This is perhaps not too surprising, since `:star` is like a `:while` loop with no termination condition, and a `:while` loop can also be simulated using recursion. (With no termination condition, a program like `(:begin (:star` p`) :fail)` will simply run forever.)

4.2.6 Searching through rooms

The `:choose`, `:for-some`, and `:star` constructs of *ERGO* are especially useful in programs where actions are required but the exact details are not known in advance. In this case, the programmer can leave the details to be sorted out at runtime using nondeterminism.

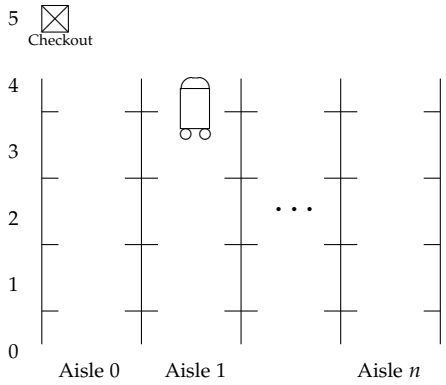
Consider, for example, having the robot find its way to some other room *rm* in a building similar to the one in Figure 3.2. Finding a path through the rooms is most easily done using a backtracking search. The code is shown in Figure 4.2.

In the procedure `get-to-room`, the program performs transitions from room to room some number of times using the `:star` primitive. It then confirms using `:test` that at the end, the robot is located in room *rm*. To perform a room transition, a door is selected to go through using the `:for-some` primitive. This door must be connected to the room currently occupied by the robot, and it may need to be opened. Once an appropriate door is chosen,

Figure 4.2: A search through the rooms

```
;; the room that is connected to room rm by door d (or #f)
(define (connecting-room rm d)
  (for/or ((rm* all-rooms)) (and (not (eq? rm rm*)) (connected? d rm*) rm*)))

;; find a way of getting to a room rm by a backtracking search
(define (get-to-room rm)
  (:begin
     (:star                                          ; as often as necessary
        (:for-some d all-doors                          ; choose a door
           (:test (connected? d (location robot)))      ; it must be nearby
           (:unless (open? d) (:act (open! d)))         ; open it, if necessary
           (:act (goto! (connecting-room (location robot) d))) )) ; go through it
     (:test (eq? (location robot) rm)) ))           ; end up in goal room
```

Figure 4.3: The grocery store world



the robot needs to go through that door to the adjoining room, which is calculated using the `connecting-room` function.

The backtracking search needed to execute this *ERGO* program is actually very similar to planning. The relationship between the two will be further explored in Section 4.4.

4.3 A grocery store example

Let us now consider another scenario. Imagine that a robot is placed in a grocery store like the one depicted in Figure 4.3. The robot can go up and down an aisle to pick up items on the shelves. It can also go left and right from aisle to aisle, provided it is located at one of the ends of an aisle. Items in the grocery store each have a location (a, s), where a is the number of the aisle and s is the number of the shelf along that aisle. A BAT for this world is presented in Figure 4.4. The fluents are the following: `location`, the location (a, s) of the robot; `grocery-list`, a list of items that the robot is shopping for; and `trolley-contents`, a list of items that the robot has already picked up and removed from its grocery list.

Figure 4.4: Program file `Examples/grocery-bat.scm`

```
;;;  Basic action theory for a grocery world

;; Fixed world parameters
(define MIN-SHELF 0)                    ; min shelf on an aisle
(define MAX-SHELF 11)                   ; max shelf on an aisle
(define CHECK-OUT-LOCATION '(0 0))      ; location of checkout counter
(define STORE-LOCATION                  ; locations of items on shelves
   (hasheq 'milk '(2 4) 'chocolate '(7 8) ))

;; Fluents:  location of robot, desired items, contents of grocery cart
(define-fluents
  location  '(0 0)
  grocery-list  '(milk chocolate)
  trolley-contents  '())

;; Auxilliary functions
(define (grocery-loc item) (hash-ref STORE-LOCATION item))
(define (aisle loc) (car loc))
(define (shelf loc) (cadr loc))
(define (aisle-end?) (memq (shelf location) (list MIN-SHELF MAX-SHELF)))

;; Actions
(define-action (right! n)               ; go right to aisle n
  #:prereq  (and (aisle-end?) (< (aisle location) n))
  location  (list n (shelf location) ))

(define-action (left! n)                ; go left to aisle n
  #:prereq  (and (aisle-end?) (> (aisle location) n))
  location  (list n (shelf location) ))

(define-action (up! n)                  ; go up to shelf n in current aisle
  #:prereq  (< (shelf location) n)
  location  (list (aisle location) n))

(define-action (down! n)                ; go down to shelf n in current aisle
  #:prereq  (> (shelf location) n)
  location  (list (aisle location) n))

(define-action (pickup! item)           ; add item on grocery list to trolley
  #:prereq  (equal? location (grocery-loc item))
  grocery-list (remove item grocery-list)
  trolley-contents (cons item trolley-contents) )

;; to display the state of the grocery store
(define (show-store)
   (printf "The grocery list contains ~a.\n" grocery-list)
   (printf "The robot is located in aisle ~a at shelf ~a.\n"
           (aisle location) (shelf location))
   (printf "The trolley contents are ~a.\n" trolley-contents) )
```

The goal of the robot in this world is to navigate the grocery store, picking up all the items on its list, and then to proceed to the checkout counter. A program that does this is shown in Figure 4.5. The program works by first iterating through the shopping list one

Figure 4.5: Program file `Examples/grocery-main.scm`

```
;;; This is a version of a grocery shopping robot.
;;; The goal is to obtain all the items on a shopping list.

(include "grocery-bat.scm")

;; get to a given shelf on the current aisle: up or down or nothing
(define (get-to-shelf s)
  (:choose (:act (up! s)) (:test (= (shelf location) s))  (:act (down! s))))

;; get to a given aisle: get to the end of the aisle, then right or left
(define (get-to-aisle a)
  (:unless (= (aisle location) a)
      (:choose (get-to-shelf MIN-SHELF) (get-to-shelf MAX-SHELF))
      (:choose (:act (right! a)) (:act (left! a))) ))

;; get to a goal location: get to that aisle and then to that shelf
(define (get-to-location goal)
  (:begin (get-to-aisle (aisle goal)) (get-to-shelf (shelf goal))))

;; the robot procedure: pick up every item and then go to checkout
(define (get-groceries)
   (:begin
      (:until (null? grocery-list)
         (:for-some item grocery-list
            (get-to-location (grocery-loc item))
            (:act (pickup! item)) ))
      (get-to-location CHECK-OUT-LOCATION)  ))

;; the main program
(define (main) (display-execution show-store (ergo-do (get-groceries))))
```

item at a time, going to where that item is located in the store and putting it in the trolley, and then going to the location of the checkout counter. To get to a desired location, the robot first goes to the desired aisle using `get-to-aisle`, and then up or down the aisle using `get-to-shelf`. To get to a desired aisle, the robot first goes to an end of the current aisle, then moves right or left as needed.

While this program gets the job done, it has some limitations. For one thing, if the robot is located at position (a, s) and wants to get to position (a', s') where a' is not a, it nondeterministically chooses which end of the current aisle to exit. This can involve needless travel on its part. For example, assuming there are 11 shelves on each aisle, if the robot is located at shelf 10 and needs to go to shelf 9 on another aisle, it should exit the current aisle at the top (shelf 11) and not at the bottom (shelf 0). A better version of the `get-to-location` program appears in Figure 4.6. In this version, `get-to-aisle` is not used, and the decision about which end of the aisle to exit is based on the values of the current and goal shelves.

A more serious problem with the program is that the items on the grocery list are considered in the order they appear on the grocery list. Unless the grocery list has been prepared with care, this can result in a lot of zigzagging though the store. A better strategy is to always choose the item on the grocery list that is closest to the current location:

Figure 4.6: Better aisle navigation

```
(define (get-to-location goal)
  (:let ((ag (aisle goal)) (sg (shelf goal))           ; goal coords
         (an (aisle location)) (sn (shelf location)))  ; current coords
     (:begin
        (:unless (= ag an)
           (get-to-shelf (if (> (+ sg sn) MAX-SHELF) MAX-SHELF MIN-SHELF))
           (:choose (:act (right! ag)) (:act (left! ag))))
        (get-to-shelf sg))))
```

```
(:let ((item (closest-item location grocery-list)))
   (get-to-location (grocery-loc item))
   (:act (pickup! item)))
```

The function `closest-item` would go through the current grocery list returning the item whose distance to the current location was minimal.

While this "greedy" strategy might work well in this grocery store, there are cases where choosing the closest item repeatedly can fail to produce a trajectory that is minimal overall. An even better strategy is to choose an arbitrary item from the grocery list, but on the way there, to watch for and pick up any object that is on the grocery list. (The `:monitor` primitive presented in Chapter 5 is well-suited to this type of programming. See the delivery agent example in Section 5.2.1.) In fact, a simple related strategy employed by many shoppers with big grocery lists is to traverse each of the aisles in sequence looking for items that are on the grocery list.

4.4 Programming versus planning

As a matter of programming style, *ERGO* programs can range from very detailed deterministic programs (like those of the form `(:begin (:act` a_1`) ... (:act` a_n`))` all the way to nondeterministic ones where many of the execution details are left out. An example of the latter is the `get-to-room` program of Figure 4.2, where the necessary transitions from room to room were not specified in the program, but had to be discovered by `ergo-do` during execution by searching the topology of the building.

At its most extreme, a nondeterministic program might only specify a final condition that needs to be satisfied. Consider the following program:

```
(define (achieve-simple goal? actions)
   (:begin
      (:star (:for-some a actions (:act a)))
      (:test (goal?))   ))
```

This *ERGO* procedure says very little about how it should be executed: repeatedly choose some action in the given list and perform it, after which the given goal condition must be satisfied. To execute this program, `ergo-do` ends up having to find a sequence of actions from the list that can be legally executed starting in the initial state and that terminates in a state satisfying the goal condition. In other words,

```
(ergo-do (achieve-simple goal? actions))
```

is really just the same as

```
(ergo-simplan goal? actions).
```

The conclusion: the execution of nondeterministic *ERGO* programs completely subsumes the basic planning seen in Chapter 3 as a special case.

The advantage of *ERGO* programming over planning as way of specifying what a robot should do is that a program can easily include additional control details appropriate for the task at hand. It can thus serve as a powerful plan filter. For example, by using

```
(:star (:for-some a actions (:act a))
       (:test (not (prune?))) )
```

a pruning condition given by `prune?` can be used to ensure that some condition is satisfied throughout the execution of the plan (like safety in the fox, hen, grain planning problem). Alternatively, a program like

```
(:star (:for-some a actions (acceptable? a) (:act a)))
```

could be used to filter the choice of action according to some given criteria (for example, only use actions that make a loud noise when nobody will be disturbed). Another tactic is to only look for plans that are limited in some way (before trying something else):

```
(define (achieve-bound cost bound goal? actions)
  (let loop ((c 0))
    (:choose (:test (and (<= c bound) (goal?)))
             (:for-some a actions
                (:act a)
                (loop (+ c (cost a)))))))
```

This procedure searches for a plan to achieve the goal but whose total action cost (according to a given cost function) does not exceed a given maximum bound.

As we come to know more about the domain, our instructions about what to do can become more specific. In the nondeterministic `get-to-room` procedure of Figure 4.2, instead of a search for *actions*, the program searches for *doors* to go through. Once these door choices have been made, the rest of the program follows deterministically. This is significant since there may be a very large set of legal actions that could be considered next, and the unrestricted planning problem might be very difficult to solve.

Consider the grocery store example from Section 4.3. For a large enough grocery list, it would be quite difficult to solve the grocery shopping as a simple planning problem: the size of the search tree would simply be too big. On the other hand, a well-constructed *ERGO* program can easily deal with hundreds of grocery items. Moreover, the *ERGO* program can provide a fine-grained control of the behaviour. It would be quite challenging to formulate planning problems for each of the various ways of going through the shopping aisles. It is much simpler to be able to say how to navigate the aisles in a program.

4.5 Efficiency considerations

For the most part, the implementation of the *ERGO* system allows programs to be written without too much concern for how well they will scale. To see where programs will run into efficiency issues, it is useful to consider a simple test BAT with these definitions:

```
(define-fluents afluent 0 bfluent 0)
(define-action a+ afluent (+ afluent 1))
(define-action b+ bfluent (+ bfluent 1))
```

There are two numeric fluents, `afluent` and `bfluent`, which are incremented by two actions, `a+` and `b+` respectively.

It is possible to generate extremely long sequences of actions in a fraction of a second using a deterministic program like this:

```
(let loop ((n (expt 2 20))) (:when (> n 0) (:act a+) (loop (- n 1))))
```

The `ergo-do` function will quickly produce a list of 2^{20} actions. (To avoid seeing them all, pass the value returned by `ergo-do` through a function like `length`.)

Accessing the values of fluents is also very efficient, and so the following program takes about the same amount of time:

```
(:while (< afluent (expt 2 20)) (:act a+))
```

It is not hard to confirm that the time required for the execution of this program scales linearly (as to be expected), so that

```
(:while (< afluent (expt 2 25)) (:act a+))
```

takes about 32 times as long to execute. (Note that a significant proportion of the time it takes to run this program is spent in Scheme garbage collection. For very time-critical applications, it may be useful to find out how to turn off this garbage collection.)

Where care is required in programming is with nondeterminism. The program

```
(:begin
  (let loop ((n (expt 2 20)))
     (:when (> n 0) (:choose (:act a+) (:act b+)) (loop (- n 1))) )
  (:test (= afluent (expt 2 20))) )
```

is similar to the one above and will generate a list of 2^{20} actions. This will take somewhat longer to execute, but almost all of the extra time is due to garbage collection. For this program, the first action selected inside the `:choose`, the `a+` action, is always the correct one; no backtracking is needed. This can be contrasted with the following program:

```
(:begin
  (let loop ((n 20))
     (:when (> n 0) (:choose (:act a+) (:act b+)) (loop (- n 1))) )
  (:test (= bfluent 20)))
```

In this case, the first action selected is *never* the correct one; the only way to pass the final test is to always choose `b+` actions. Although the final list of actions has only 20 elements, `ergo-do` must explore the entire search tree of 2^{20} elements to eventually find that correct final sequence. This takes about the same time as generating a list with 2^{20} elements, discounting garbage collection. Consequently, taking into account the Million-Billion Rule mentioned on page 35, we can predict that the program

```
(:begin
  (let loop ((n 30))
     (:when (> n 0) (:choose (:act a+) (:act b+)) (loop (- n 1))) )
  (:test (= bfluent 30)))
```

which will take 2^{10} times longer to finish, will have unacceptable execution time.

Similar considerations apply to the backtracking that results from the `:star` primitive. Execution of the program

```
(:begin
  (:star (:act a+))
  (:test (= afluent 1000)))
```

will find the successful list of 1000 actions quite quickly. This is because only 1000 possibilities will be considered. This is in contrast with the following program:

```
(:begin
  (:star (:choose (:act a+) (:act b+)))
  (:test (= afluent 20)))
```

In this case, before `ergo-do` can find the correct list with 20 elements, it must first go through entire trees containing the sequences of `a+` and `b+` actions of length $1, 2, \ldots, 19$. Again, we can predict that the program

```
(:begin
  (:star (:choose (:act a+) (:act b+)))
  (:test (= afluent 30)))
```

will take too long to execute. Blind search has its limitations.

As noted above, one of the major advantages of *ERGO* programming over planning is that the search does not have to be blind. We can see this very dramatically, by inserting conditions to be satisfied at various points in the search above:

```
(:begin
  (:star (:choose (:act a+) (:act b+)))
  (:test (= afluent 10))
  (:star (:choose (:act a+) (:act b+)))
  (:test (= afluent 20))
  (:star (:choose (:act a+) (:act b+)))
  (:test (= afluent 30)))
```

This program is similar to the previous one, except that it insists on progress being made along the way. The net effect is that `ergo-do` only has to blindly explore very small trees (no larger than 2^{10}) to then assemble the final sequence of length 30. This program terminates almost immediately.

* 4.6 How this all works

As with basic action theories in the previous chapter, it is not necessary to understand how the `ergo-do` function is implemented in Scheme to be able to use it. This is just as well,

since the implementation relies heavily on macros (not covered in this book) to do much of its work. Nonetheless, a rough overview of the implementation is presented here for those who may be interested in extending or revising it.

The macro-defining operator used in the implementation is `define-macro` so that

```
(define-macro (foo x) `(car ,x))
```

causes `(foo '(a b c))` to behave like `(car '(a b c))`. Macros are more flexible than functions since they need not evaluate their arguments, or can do so selectively.

Program expressions in *ERGO* like `(:begin` p q`)` are evaluated before they are used. They each evaluate to a function that takes three arguments:

- `hist`, a list of actions performed so far, which will be (reversed and then) returned by `ergo-do` on successful completion;
- `fail`, a function of no arguments called the failure continuation, to call if the *ERGO* program cannot take a successful step;
- `succ`, a function of three arguments called the success continuation, to call if the *ERGO* program can take a successful step. (The arguments are explained below).

The last thing an *ERGO* program must do is to call one of these continuations or ask some other *ERGO* program to do so.

For example, the *ERGO* program construct `:if` can be defined as follows:

```
(define-macro (:if cond p1 p2)
   `(lambda (hist fail succ)
       (if ,cond
          (,p1 hist fail succ)
          (,p2 hist fail succ) )))
```

Note first of all that `(:if` *expr* pgm_1 pgm_2`)` expands to a function of three arguments (via `lambda`), as required. When called, this function evaluates the given *expr* (using `if`) and then continues with either pgm_1 or pgm_2. These must be *ERGO* programs themselves and so are expected to take the three arguments as above and to call `fail` or `succ` as appropriate.

(One minor complication in the above scheme are the actual names `hist`, `fail`, and `succ`, which must not collide accidentally with user variables used in *expr*, pgm_1, or pgm_2. To ensure this, the implementation actually uses `dm-ergo` instead of `define-macro`, to generate new variables different from all the variables that appear in user programs.)

The `:test` construct is a simple example of an *ERGO* program that calls one of the continuations directly:

```
(define-macro (:test cond)
   `(lambda (hist fail succ)
       (if ,cond
          (succ hist fail #f)
          (fail))))
```

Either succ or fail is called depending on whether the condition is true. The fail continuation takes no arguments. The arguments for the succ continuation are the history, the failure continuation, and the steps remaining in the program under consideration. (The #f indicates no further steps needed for a :test.) Note that fail is passed as an argument to succ. This is because the success continuation may later decide that something has failed and need to backtrack to the failure continuation.

To see why intuitively, consider a program like (:begin (:choose pgm_1 pgm_2) pgm_3). If pgm_1 fails, execution continues by trying pgm_2. If pgm_1 succeeds, on the other hand, execution continues with pgm_3. However, if this pgm_3 subsequently fails, the execution must backtrack and try the pgm_2, just as if pgm_1 had originally failed.

The handling of primitive actions via :act is also done by either calling the failure or success continuation. The definition is (approximately) as follows:

```
(define-macro (:act a)
   `(lambda (hist fail succ)
       (if (legal-action? ,a)
           (begin (change-state ,a) (succ (cons ,a hist) fail #f))
           (fail))))
```

If the action is not legal, fail is called; otherwise, the state is changed using change-state, and succ is called with an updated history of actions. (The history of actions is not used for online execution, discussed in Chapter 6.)

The basics of backtracking in *ERGO* can be seen in the definitions of the :choose and :begin program constructs. Here is how a simpler version of :choose with just two arguments can be defined:

```
(define-macro (:choose2 p1 p2)
   `(lambda (hist fail succ)
      (let ((w (current-state)))
        (,p1 hist
             (lambda () (restore-state w) (,p2 hist fail succ))
             succ))))
```

This says that to execute (:choose2 pgm_1 pgm_2), the state of the world is first saved, and then the program pgm_1 is executed with succ as the success continuation, but with a new failure continuation: if pgm_1 fails, rather than simply calling fail, the state is restored, and then the program pgm_2 is be attempted, which in turn will call fail or succ as appropriate. In other words, try pgm_1, and if that succeeds, execution continues normally; but if it fails, pgm_2 is attempted next.

The :begin construct is similar except that it is the success continuation that is modified. A two argument version can be defined as follows:

```
(define-macro (:begin2 p1 p2)
   `(lambda (hist fail succ)
       (,p1 hist fail
          (lambda (h f c) ((if (not c) ,p2 (:begin2 c ,p2)) h f succ)))))
```

This says that to execute (:begin2 pgm_1 pgm_2), the program pgm_1 is executed with fail as the failure continuation, but with a new success continuation: if pgm_1 is able to take a step

successfully, execution continues not with `succ`, but with the remaining steps from pgm_1 (if any) followed by pgm_2. (Extra complications arise in the case of concurrent programs where steps from some other program may be interleaved with this execution.)

The last piece of the implementation puzzle concerns how the interpreter `ergo-do` itself is defined. In the simple case, when the execution is `'first` mode, the definition is (roughly) as follows:

```
(define (ergo-do p)
   (define (succ h f c) (if c (c h f succ) (reverse h)))
   (save-state-excursion (p '() (lambda () #f) succ)))
```

So `(ergo-do` *pgm*`)` calls the *ERGO* program *pgm* with an empty history of actions, a failure continuation that simply returns `#f`, and a success continuation that continues through each step of the program until there are no steps left, in which case, it simply returns the final history of actions in reverse order.

4.7 Exercises

1. Using the `house-bat` of Chapter 3, write an *ERGO* program that has the robot visiting every room of the house.

2. Imagine a robot in a much larger house than the one in Chapter 3. Write an *ERGO* program to get to an arbitrary room by first calculating the shortest path.

3. As noted, the `:star` primitive of *ERGO* is a feature that can be duplicated using recursion. Compare the version given in the text to this simpler recursive one:

   `(let loop () (:choose :nil (:begin` p_1 ... p_n `(loop))))`

   When would this second version be preferable, if ever?

4. The `achieve-bound` procedure given in the text could be improved by making it greedier: sort the list of actions by cost, and always try actions of lower cost first. Rewrite the procedure to do this. Discuss where it would make sense to sort the actions once only, and where it would be better to sort the actions each time through the loop. (Note: The procedure does an exhaustive search through all sequences of actions, which is fine when there are a small number of them. The problem of efficiently finding a minimal cost plan otherwise is considerably harder and requires something more like dynamic programming.)

Chapter 5

Concurrent Programming and Reactivity

In more complex cognitive robotic systems, a robot must not only deliberate about what to do, but also react in an appropriate way to the dynamic world it inhabits. It often helps to visualize the robot as performing more than one task at the same time. For example, a robot may be going from room to room picking up garbage, but also closing any open window it finds, and stopping occasionally to recharge its battery. A program with this type of behaviour can certainly be written using just the programming primitives seen in Chapter 4, but it would tend to be messy. An otherwise simple program for picking up garbage would be interspersed everywhere with extra code to deal with windows and battery levels. To avoid this, it is better to program each of these simple behaviours as independent programs and then to ask *ERGO* to execute them *concurrently*.

5.1 The `:conc` and `:atomic` primitives

There are two main program primitives in *ERGO* for concurrency,

`(:conc` $p_1 \ldots p_n$`)`

and

`(:monitor` $p_1 \ldots p_n$`),`

where the p_i evaluate to subprograms that are to be executed concurrently. The concurrency here does not mean true simultaneity, but rather an *interleaving* of steps. In other words, each subprogram is assumed to go through a number of steps in order, where a step is either `(:act` a`)`, which involves the execution of some legal action, or `(:test` e`)`, which involves testing the truth of some condition. When two or more programs are executed concurrently, each subprogram will go though its steps in the order specified, but between any two steps, steps from other subprograms may appear.

For example, suppose that `a`, `b`, `c`, and `d` are actions that have been defined as having no prerequisites. Consider the program

```
(:conc (:begin (:act a) (:act b)) (:begin (:act c) (:act d)))
```

The first subprogram asks for `a` then `b` in sequence; the second subprogram asks for `c` then `d` in sequence. When executed concurrently, there are six possible executions:

```
> (ergo-do #:mode 'offline
       (:conc (:begin (:act a) (:act b)) (:begin (:act c) (:act d))))

(a b c d)
OK? (y or n)  n

(a c d b)
OK? (y or n)  n

(a c b d)
OK? (y or n)  n

(c d a b)
OK? (y or n)  n

(c a b d)
OK? (y or n)  n

(c a d b)
OK? (y or n)  n
#f
```

Note that in all these cases, the `a` step precedes the `b` step and the `c` step precedes `d` step, as required. The `:conc` primitive, in other words, searches for a legal interleaving that satisfies the ordering given by the subprograms, backtracking as required.

Suppose now that one of the effects of action `b` is make the prerequisite of action `d` false. In this case, three of the executions above would be ruled out:

```
(a b c d)
(a c b d)
(c a b d)
```

These would no longer be legal since they require `d` to be executed in a state where its prerequisite is false. On the other hand, if the `c` action also happens to make the prerequisite of `d` true, the first of the these three would again be a legal interleaving of the actions.

Steps that involves the `:test` primitive are similar to those involving `:act`. Consider a program of the following form: (`:conc` *p* (`:begin` *q* (`:test` *e*) *r*)) The first subprogram generates the steps for *p*, while the second first generates the steps for *q*, then a test that *e* is true, and then more steps for *r*. The steps from the two subprograms are then to be interleaved. The effect of the (`:test` *e*) is that the interleaving must be such that in some state after *q* is done but before *r* begins, the condition *e* must be satisfied. This can impose constraints on where the steps of *p* can appear. For example, *e* might be (`> f 5`), where `f` is some fluent, and *p* might be a loop that increments the value of `f` from 0. In that case, *r* will only be executed after *p* has incremented it six times. In other words, in the presence of concurrency, (`:test` *e*) can be read as "wait until condition *e* is true," where the waiting is for other concurrent subprograms.

It is worth noting that `:act` and `:test` are the only programming constructs that generate steps. For example, when (`:if` e p q) is interleaved with some other program, the evaluation of the condition e is *not* considered a step. In other words, it cannot happen that e evaluates to true, some other program does something (for example to change e), and then p begins to execute in the changed state. So the p will always begin execution in a state where the e is true.

There may be cases where it is actually useful to treat the evaluation of e as a step and to allow interleaving. This can be achieved by using `:test` explicitly instead of the `:if` primitive, as in the following:

(`:choose` (`:begin` (`:test` e) p) (`:begin` (`:test` (`not` e)) q))

Other programming constructs like `:when` and `:while` can also be rewritten to use `:test` in an analogous way.

On the other hand, it may sometimes be desirable to avoid any interleaving with certain programs. For this, the `:atomic` primitive can be used. The expression

(`:atomic` $p_1 \ldots p_n$)

behaves just like (`:begin` $p_1 \ldots p_n$), except that the resulting program is treated as an indivisible operation with no interleaved steps. So in the end,

(`:atomic` (`:choose` (`:begin` (`:test` e) p) (`:begin` (`:test` (`not` e)) q)))

behaves once again like (`:if` e p q).

5.1.1 A table lifting example

An example of a full program using `:conc` appears in Figure 5.1. This is a program for two robots (or a single robot with two arms) to safely lift a table. The idea is that both ends of the table must reach a certain goal height, but that at no point should one end of the table be more than a certain amount higher than the other (so that nothing will fall off).

There are different ways to program this, but one way is to imagine that each robot will be executing an identical subprogram called `lifter`, and that the main program will run these two subprograms concurrently. The `lifter` for each robot does the following: it grabs one end of the table (the first robot gets to choose either end, and the second robot gets what is left); then repeatedly, until the table has been completely raised, the robot lifts its end by a small amount. The action `lift!` has as a prerequisite that the table be safe to lift at that point, that is, that the new height to be attained not exceed the height of the other end by more than the fixed `tolerance` amount.

Running the `main` program produces the following list of actions:

```
((grab! r1 e1) (lift! r1 1) (grab! r2 e2) (lift! r2 1) (lift! r2 1)
 (lift! r1 1)  (lift! r1 1) (lift! r2 1)  (lift! r2 1) (lift! r1 1)
 (lift! r1 1)  (lift! r2 1) (lift! r2 1)  (lift! r1 1))
```

The first robot grabs an end and lifts it, but then must wait, since a second lifting is not allowed (it would exceed the tolerance). So the second robot now grabs the other end and does two liftings, after which it must wait. Execution continues, alternating between the two robots until the table has been safely raised.

Figure 5.1: Program file `Examples/lift-table.scm`

```
;;;  Two robots must lift two ends of a table in increments up to a height
;;;  while ensuring that the table remains level (to a tolerance)

;;;  Two fluents:  position of table ends, and holding status of each robot
;;;  Two actions:  grab an end of a table, and move vertically

(define robots '(r1 r2))                ; the robots
(define ends '(e1 e2))                  ; the table ends

(define goal-height 6)                  ; the desired height
(define amount 1)                       ; the increment for lifting
(define tolerance 1)                    ; the tolerance

(define-fluents
   pos-table  (hasheq 'e1 0 'e2 0)      ; vertical pos of table end
   held-table (hasheq 'r1 #f 'r2 #f))   ; what robot is holding (#f = nothing)

;; some useful abbreviations
(define (pos e) (hash-ref pos-table e))
(define (held r) (hash-ref held-table r))
(define (table-is-up?)                  ; both ends higher than goal-height?
  (for/and ((e ends)) (>= (pos e) goal-height)))
(define (safe-to-lift? r z)             ; ok for robot to lift its end?
  (let ((e (held r)))
    (let ((e* (for/or ((d ends)) (and (not (eq? e d)) d))))
      (<= (pos e) (+ (pos e*) tolerance (- z))))))

;; action of robot r grabbing table end e
(define-action (grab! r e)
   #:prereq    (and (for/and ((r* robots)) (not (eq? e (held r*))))
                    (not (held r)))
   held-table  (hash-set* held-table r e))

;; action of robot r moving his end of the table up by z units
(define-action (lift! r z)
   #:prereq    (and (held r) (safe-to-lift? r z))
   pos-table   (let ((e (held r))) (hash-set pos-table e (+ (pos e) z))))

;; the main lifting program for robot r
(define (lifter r)
   (:begin (:for-some e ends (:act (grab! r e)))              ; grab an end
           (:until (table-is-up?) (:act (lift! r amount)) ))) ; lift repeatedly

;; the main program: run both robots concurrently
(define (main) (ergo-do (:conc (lifter 'r1) (lifter 'r2))))
```

5.2 The `:monitor` primitive

It is most often the case when two tasks are to be handled concurrently that one of them has higher priority than the other. Consider the case of a robot that is moving from room to room (for some purpose), but is also keeping track of its battery level to avoid running out of power. While the battery level is high, the behaviour is the normal one. But if

the battery level drops below some amount, the normal behaviour is suspended, and the charging behaviour takes on a higher urgency.

The *ERGO* program (:monitor $p_1 \ldots p_n$) is executed as follows: the program p_n is fully executed; however, before every step of p_n, the program p_{n-1} is fully executed; similarly, before every step of p_{n-1}, the program p_{n-2} is fully executed; and so on up to p_1. In other words, p_1 has the highest priority since no step takes place without it first being fully executed, and p_n has the lowest priority, since it only gets to execute a step when all the other programs have finished.

Typically, these p_i (except possibly for p_n) involve either a :when or a :while: if a certain condition is true, then do something; otherwise do nothing so that a lower priority program will get a chance to do its part. For example, a program of the form

```
(:monitor (:when e p) q)
```

says to do program q except that p should be done before every step of q but only when e is true. Similarly, the program

```
(:monitor (:while e p) q)
```

says to do program q except that p should be done repeatedly before every step of q while e is true. Consequently, for the program

```
(:monitor (:while #t p) q)
```

the program q would never get a chance to execute any steps.

Here is a sample program that moves boxes to Room3 while dealing with a battery that can only be charged in Room1:

```
(:monitor
   (:when (< battery-level 10)
       (get-to-room 'room1))
   (:when (and (< battery-level 15) (eq? (location robot) 'room1))
       (:act charge-battery)))
   (:for-all box box-list (fetch box) (transfer box 'room3)))
```

The main task of the robot is to go get each box and carry it to Room3 using the (user-defined) programs fetch and transfer. However, if at any point in the execution of this task the robot finds itself in Room1 with a battery level below fifteen, it interrupts the main task to charge its battery. But before any of this, if the battery level ever drops below ten, the robot interrupts everything and gets to Room1 using get-to-room.

5.2.1 A delivery agent example

A more complex use of :monitor is presented in Figure 5.2. This example involves a robot on a two dimensional grid that must pick up a number of objects and deliver them to goal destinations. The actions are: picking up an object, putting down an object, and moving by one unit in one of the four directions on the grid (taking along any objects being carried).

This problem can be solved nicely using :monitor. The main loop does all and only the motion actions. Concurrently, :monitor watches to see if any objects should be put down or picked up along the way. The program produces the following list of actions:

Figure 5.2: Program file `Examples/delivery-bot.scm`

```
;;; This is a program for a delivery agent on a 2d grid using :monitor
;;; The objects needing delivery and their goal destinations are fixed.

(define objects '(a b c d))
(define (goal x) (case x ((a) '(3 4)) ((b) '(4 2)) ((c) '(0 3)) ((d) '(1 1))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  The basic action theory

(define-fluents
    rob-loc  '(0 0)                                              ; rob loc
    loc-table (hasheq 'a '(0 0) 'b '(4 0) 'c '(3 3) 'd '(3 3)))   ; obj locs

;;  abbreviations
(define (obj-loc x) (hash-ref loc-table x))       ; obj loc or #f
(define (carried? x) (not (obj-loc x)))           ; obj is being carried
(define (home? x) (equal? (obj-loc x) (goal x)))  ; obj is at goal?

(define-action (move! dir)           ; dir is (0 1) (0 -1) (1 0) or (-1 0)
    rob-loc (list (+ (car rob-loc) (car dir)) (+ (cadr rob-loc) (cadr dir))))

(define-action (pickup! x)
    #:prereq (equal? rob-loc (obj-loc x))      ; prereq: rob and x co-located
    loc-table (hash-set* loc-table x #f))      ; the #f says x is being carried

(define-action (putdown! x)
    #:prereq (carried? x)                      ; prereq: x is being carried
    loc-table (hash-set* loc-table x rob-loc)) ; x goes on the grid at rob-loc

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  The main program for the robot

(define (go-to d)                              ; go to loc d by a seq of moves
  (:begin (:while (< (car rob-loc) (car d)) (:act (move! '(1 0))))
          (:while (> (car rob-loc) (car d)) (:act (move! '(-1 0))))
          (:while (< (cadr rob-loc) (cadr d)) (:act (move! '(0 1))))
          (:while (> (cadr rob-loc) (cadr d)) (:act (move! '(0 -1)))) ))

(define (main) (ergo-do
   (:monitor
      ; pickup or dropoff an object as required during motion
      (:for-all x objects
           (:when (and (equal? (goal x) rob-loc) (carried? x))
               (:act (putdown! x)))
           (:when (and (equal? (obj-loc x) rob-loc) (not (home? x)))
               (:act (pickup! x)) ))
      ; move to the location of an object or carry it to its goal
      (:until (and-map home? objects)
           (:for-some x (for/only ((x objects)) (not (home? x)))
               (:let ((dest (or (obj-loc x) (goal x))))
                  (go-to dest))))                        )))
```

```
((pickup! a) (move! (1 0)) (move! (1 0)) (move! (1 0)) (move! (0 1))
 (move! (0 1)) (move! (0 1)) (pickup! c) (pickup! d) (move! (0 1))
```

```
(putdown! a)  (move! (1 0)) (move! (0 -1)) (move! (0 -1)) (move! (0 -1))
(move! (0 -1)) (pickup! b) (move! (0 1)) (move! (0 1)) (putdown! b)
(move! (-1 0)) (move! (-1 0)) (move! (-1 0)) (move! (-1 0)) (move! (0 1))
(putdown! c) (move! (1 0)) (move! (0 -1)) (move! (0 -1)) (putdown! d))
```

The robot begins by picking up the first object, `a`, that happens to be located where it is, at `(0 0)`. Then it begins moving towards the goal destination for object `a` which is `(3 4)`. It does three moves north, and three moves east along its way, picking up objects `c` and `d` at location `(3 3)`, before reaching `(3 4)`, where it drops off `a`. It continues this way until all four objects have reached their goal destination.

Note that this delivery behaviour is completely *opportunistic*. The robot does not compute in advance which objects it will pick up and in what order. If it happens to be where an object is, it picks it up and deals with it. This means that if the world changes while the program is executing, for example, if the objects move by themselves or new objects enter the grid, the robot can react appropriately to the objects at hand. This sort of reactivity is the topic of the next section.

5.3 Reactivity

Consider a very simple world with a single Boolean fluent `stopped?` (which starts out being false), an action `stop!` which makes the fluent `stopped?` true, and an action `ding!` which does nothing. By itself, the *ERGO* program `(:until stopped? (:act ding!))` is not very interesting; it simply repeats the `ding!` action forever. However, if we imagine that there are other agents at work performing actions concurrently, the behaviour of the program must be reinterpreted: the *ERGO* program repeats the `ding!` action until it is stopped by the `stop!` action. Actions performed by other agents are called *exogenous*.

This then is the essence of reactivity: the robot performs some task, while monitoring and reacting to conditions that may be changed exogenously. More generally, the robot will be performing one or more tasks, while monitoring one or more conditions that may change as the result of its actions or the actions of other agents. When one of these conditions becomes true, the robot may suspend what it is doing, deal with the condition, and then resume its normal tasks, perhaps in some modified form. The `:monitor` control construct, seen in the previous section, is an ideal way to program this sort of reactivity.

5.3.1 A reactive elevator example

Let us consider an elevator controller that can react to exogenous events that may occur:

- The elevator will be located on some floor, and will have actions to go from floor to floor. For simplicity, let us assume that the action `up!` moves the elevator up one floor, and that `down!` moves it down one floor.
- There are a certain number of call buttons that are on, and the goal of the elevator is to turn them all off using the action `turnoff!`, which turns off the call button of the current floor, and finally park the elevator on the first floor. (We put aside complications such as actual passengers on the elevator for now.)

Figure 5.3: Program file `Examples/reactive-elevator-bat.scm`

```
;;;  This is an adaptation of the elevator that appears in the IJCAI-97
;;;  paper on ConGolog.  It uses exogenous actions for temperature, smoke,
;;;  and call buttons.  The actions and fluents are described below.

(define-fluents
   floor  7                              ; the floor the elevator is on
   temp   0                              ; the temperature inside the elevator
   fan?   #f                             ; is the fan on?
   alarm? #f                             ; is the smoke alarm on?
   on-buttons '(3 5) )                   ; the floors that are being called

;; the normal actions
(define-action up!                       ; go up one floor
   #:prereq (< floor 10)  floor (+ floor 1))
(define-action down!                     ; go down one floor
   #:prereq (> floor 1)   floor (- floor 1))
(define-action toggle-fan!               ; toggle the fan
   fan? (not fan?))
(define-action turnoff!                  ; turn off button for current floor
   on-buttons (remove floor on-buttons))
(define-action ring!)                    ; ring the alarm bell

;; the exogenous actions
(define-action (turnon! n)               ; call button n is turned on
   on-buttons (if (memq n on-buttons) on-buttons (cons n on-buttons)))
(define-action heat!  temp (+ temp 1))   ; the temperature rises
(define-action cold!  temp (- temp 1))   ; the temperature falls
(define-action smoke! alarm? #t)         ; smoke activates alarm
(define-action reset! alarm? #f)         ; alarm is deactivated
```

- Now let us further assume that there is an exogenous action `(turnon! `n`)`, whose effect is to turn on the call button of floor n. In essence, these are external requests that the elevator is being asked to process. Before the elevator on floor one can actually stop running, it must ensure that there are no pending requests.

- While the elevator is doing its job, an alarm condition may become true as a result of an exogenous `smoke!` event. If an alarm condition is detected, the elevator must stop moving, and react by repeatedly ringing its bell using the `ring!` action until the alarm is finally turned off exogenously by a `reset!` event (from a firefighter).

- Finally, the elevator has an internal temperature. No matter what else the elevator is doing (serving floors or sounding alarms), if the temperature becomes too hot (as a result of exogenous `heat!` events), it must react by turning on its fan via the `toggle-fan!` action; if the temperature becomes too cold (as a result of exogenous `cold!` events), it should turn off the fan via the `toggle-fan!` action.

A simple basic action theory for an elevator like this is shown in Figure 5.3. Note that exogenous actions are presented right along with the normal ones (called *endogenous*). From the point of view of the BAT, they are indistinguishable.

Figure 5.4: Program file `Examples/reactive-elevator-run.scm`

```
;;;  This is the main program for the elevator that appears in the IJCAI-97
;;;  paper on ConGolog.  The BAT appears in reactive-elevator-bat.scm

(include "reactive-elevator-bat.scm")

;; get to floor n using up and down actions
(define (go_floor n)
   (:until (= floor n) (:if (< floor n) (:act up!) (:act down!))))

;; the main elevator program as a priority driven monitor
(define control
  (:monitor
     (:when (and (< temp -2) fan?) (:act toggle-fan!))      ; handle cold
     (:when (and (> temp 2) (not fan?)) (:act toggle-fan!)) ; handle heat
     (:while alarm? (:act ring!))                  ; stop and ring the bell
     (:until (null? on-buttons)                    ; main serving behaviour
        (:for-some n on-buttons                             ; choose a floor
             (go_floor n) (:act turnoff!)))                 ; serve it
     (go_floor 1)                                  ; default homing behaviour
     ;; (:while #t (:wait))        ; to keep the elevator running when online
))
```

The main procedure for the elevator is called `control` and is shown in Figure 5.4. It consists of a `:monitor` with five separate tasks in order of priority. The "normal" behaviour of the robot is the one in the `:until` loop: until there are no longer floors to be served, select a floor, go to that floor, and turnoff the call button. (In the program here, the selection of a floor is trivial. In a more complex setting the `on-buttons` list would be reordered, for example, to give certain floors priority, or to take into account which floor has been waiting the longest, or to minimize elevator motion, *etc.*) The rest of the `:monitor` program serves to augment this normal task, to react to events that happen during the execution, and to park the elevator when the normal task is complete.

In performing the normal elevator task, a number of `up!` and `down!` actions will be generated. Before each such action, the higher priority `:while` loop will be executed. In case the fluent `alarm?` is false, this loop terminates immediately and the normal behaviour continues. However, imagine that a `smoke!` event occurs exogenously, triggering the alarm. In this case, the `alarm?` fluent becomes true and the `:while` loop executes the `ring!` action repeatedly. Because this action does not change any fluent, this ringing may continue indefinitely. However, if a `reset!` action occurs exogenously, it will set the `alarm?` fluent to false, and the `:while` loop will terminate, allowing the normal behaviour to resume.

Before any alarm or normal behaviour takes place, the higher priority `:when` programs will be executed. When the temperature is within range, no action will be needed, and the moving or ringing will continue as before. However, if enough `heat!` or `cold!` actions take place exogenously, the ringing or moving will be interrupted momentarily so that the `toggle-fan!` action can be executed. This is a `:when` and not a `:while`, so the interruption consists of a single action only, after which the interrupted behaviour resumes.

Finally, after all the fan toggling, bell ringing, and floor serving is complete, the lowest priority task can start execution. This involves parking the elevator on the first floor.

This typically requires a number of `down!` actions. But before each `down!` action step, `:monitor` must execute the higher priority tasks. So if the call button for a new floor is pushed exogenously, the `on-buttons` fluent will become non-empty and the normal elevator behaviour will once again resume. Only when the elevator is on the first floor and no further action is required will the program terminate.

If this program is used in the simple offline mode seen so far, none of this interesting reactivity takes place:

```
> (ergo-do #:mode 'first control)
'(down! down! down! down! turnoff! up! up! turnoff! down! down! down! down!)
```

What happens here is that the elevator goes down four floors to serve floor 3, then up two floors to serve floor 5, then down four floors to park on the first floor. Nothing else happens exogenously to cause it to behave differently. It is only when this program is used in online mode (in Chapter 6) that a more interesting range of behaviour will emerge.

5.4 Playing strategic games

A very simple case of an agent reacting to exogenous actions is in game playing. Consider a game like chess. The actions available to the player are the legal moves of the game. Knowledge about the state of the game (that is, whose turn it is and where the various pieces are located on the board) is always complete. The goal of the game is to capture the other player's king. But a plan to achieve this goal cannot be a sequence of actions. Instead, each time a player makes a move, the opponent gets a turn to play and will make a move exogenously to try to stop the player from attaining the goal. The player must then decide what to do next, and the process iterates.

In terms of cognitive robotics, a game like chess is a dynamic world where the robot must choose an action to perform, then wait until an exogenous action occurs, then choose another move to perform, then wait again, and so on. It is this simple synchronized turn-taking behaviour that makes game-playing a special case of reacting to exogenous events. So *ERGO* includes a simple two-person game player, as explained below.

5.4.1 The game of tic-tac-toe

A complete basic action theory for the well-known game of tic-tac-toe appears in Figure 5.5. The fluents are the `player`, either X or O, and the `board`, a vector of nine positions, each of which will be one of X, O, or `#f` (for unoccupied). (The vector is of length ten, but position 0 is not used.) The positions are as shown below:

1	2	3
4	5	6
7	8	9

There is a single action `move!` whose parameter is the board position to be occupied by the current player. The idea is that when it is the robot's turn to play, `move!` will be a normal,

Figure 5.5: Program file `Examples/GameExamples/ttt.scm`

```
;;;   The game of tic tac toe using a general game player.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   The action theory: 2 fluents and 1 action

(define-fluents
   board  (vector #f #f #f #f #f #f #f #f #f)  ; the initial board
   player 'x)                                  ; player who plays next

;; the current player occupies the board at position sq
(define-action (move! sq)
   #:prereq (not (vector-ref board sq))         ; square is not occupied
   board    (vector-set board sq player)        ; player occupies square
   player   (if (eq? player 'x) 'o 'x))         ; the turns alternate

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   Auxiliary definitions

(define squares '(1 2 3 4 5 6 7 8 9))

(define lines  '((1 2 3) (4 5 6) (7 8 9) (1 4 7) (2 5 8) (3 6 9)
                 (3 5 7) (1 5 9)))

(define (occ sq) (vector-ref board sq))

(define (has-line? pl)                       ; player pl owns some line?
  (for/or ((ln lines)) (for/and ((sq ln)) (eq? pl (occ sq)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   The game playing

(define (winner)                             ; the winner of a board or #f
   (if (has-line? 'x) 1                                    ; player1
      (if (has-line? 'o) -1                                ; player2
          (and (and-map occ squares) 0))))                 ; a tie

(define (print-board)                        ; display 3 lines of the board
   (define (printsq sq) (display (or (occ sq) "-")))
   (define (println ln) (display "   ") (for-each printsq ln) (display "\n"))
   (for-each println '((1 2 3) (4 5 6) (7 8 9))))

;; X moves first via minimax; O plays via read
(define (main) (ergo-play-game winner (map move! squares) print-board))
```

endogenous action; when it is the opponent's turn to play, `move!` becomes an exogenous action. Either way, the action changes the board and who is to play next.

The ultimate goal of a tic-tac-toe robot is to fully occupy one of the eight horizontal, vertical, or diagonal lines on the board. However, as mentioned above, there can be no simple offline plan to achieve this goal since a player must respond to the moves of an opponent. From an online point of view, we can think of the goal of the robot as choosing a single good move to make starting in some state.

There are two top-level *ERGO* functions that can be used for playing strategic games: `ergo-generate-move` and `ergo-play-game`. The first of these returns a player (a function of no arguments that when called generates a single next move), while the second function

Figure 5.6: The start of a game of tic-tac-toe

```
Player 1 moves will be generated (endogenous).
Player 2 moves must be entered (exogenous).
The current state is as follows:
   ---
   ---
   ---
Action (move! 1) is chosen.
The current state is as follows:
   x--
   ---
   ---
The following actions are now possible:
  0  (move! 2)
  1  (move! 3)
  2  (move! 4)
  3  (move! 5)
  4  (move! 6)
  5  (move! 7)
  6  (move! 8)
  7  (move! 9)
Enter a number from 0 to 7 to choose one:
```

plays an entire game by generating and reading moves repeatedly. The format for the `ergo-generate-move` function is as follows:

> `(ergo-generate-move` *win actions max?*
> `[#:static` *st*`]` `[#:depth` *n*`]` `[#:infinity` *k*`])`

Leaving aside the optional arguments for now, this is similar to the `ergo-simplan` function of Chapter 3. As before, *actions* is a list of actions to be considered in deciding how to move. Instead of a binary goal condition, however, the function *win* of no arguments indicates when the game is over. The value returned should be false in those states where the game is not over, but otherwise, it should have a numeric value: 1, if the player who is moving first (according to the initial state) has won: −1, if the player who is moving first has lost; and 0, for a tie. The `winner` function in Figure 5.5 does this for tic-tac-toe, where X is assumed to move first. The *max?* argument should be `#t` or `#f` according to whether the generated player will be playing first or second. The format for `ergo-play-game` is similar:

> `(ergo-play-game` *win actions show*
> `[#:static` *st*`]` `[#:depth` *n*`]` `[#:infinity` *k*`])`

The extra argument *show* is a function of no arguments whose effect is to print the state of the board. This argument is handled by `print-board` in Figure 5.5. The `ergo-play-game` function works by calling the value of `ergo-generate-move` to get a move, updating the state, showing the board and a numbered list of legal actions available to the opponent, reading the opponent's choice (as a number), updating the state again, and then iterating until the game is over. (The fact that this function obtains moves from an opposing player

makes this an online controller, as we will see in Chapter 6, but one that is rudimentary enough to consider here.) The start of a game of tic-tac-toe is shown in Figure 5.6.

The game player returned by `ergo-generate-move` uses minimax with alpha-beta pruning to decide on a next move, but those details are beyond the scope of this book.

* 5.4.2 The game of pousse

A basic action theory for the much more challenging game of Pousse is shown in Figure 5.7. This is a game played on a 4×4 board between two players, X and O. Like tic-tac-toe, the object is to own a horizontal or vertical line on the board (no diagonals). However, tokens are placed on the board in a different way: they are pushed onto the board from the top, bottom, left or right of the board. As a token enters the board from one of the four edges, other tokens on the board may have to move to make room, falling off the board at the other edge if the row or column was full. (The word "Pousse" is French for "push.") All sixteen push moves are legal throughout, although a player loses if a move results in a board that duplicates an earlier one.

While the bookkeeping details are certainly more complex, the structure of the final program for Pousse shown in Figure 5.8 is similar to that of tic-tac-toe. One extra complication concerns the size of the minimax search tree. For small games like tic-tac-toe, the entire tree can be explored. But for bigger games like Pousse, it is necessary to truncate the search at a certain depth.

The three optional arguments for `ergo-generate-move` and `ergo-play-game` are used for this purpose: the `#:static` argument is a function of no arguments whose value in a state is a number estimating how good the state is from the point of view the player who is moving first (as with *win*); the `#:infinity` argument is an upper bound on the numbers that will be generated by the static evaluation function; and finally, the `#:depth` argument is how deep the search should proceed before using the static evaluation function.

For Pousse, the static evaluation function used in Figure 5.7 involves summing occupancy (the squares of the numbers of rows and columns occupied) and centrality (the sums of distances of tokens to corners). Using this static evaluation function at a depth of six results in quite reasonable playing behaviour. (There are sixteen moves per board, so a depth of six leads to a game tree of size 2^{24}, small enough to be tolerable.)

5.5 Exercises

1. Change the reactive elevator so that there are priority floors that must be served before any non-priority ones.

2. Redesign the delivery agent robot so that multiple robots can be working simultaneously on the grid. If we assume that the passageways are wide enough to accommodate multiple robots (unlike in the next exercise), the main control issue will be to split up the work to avoid more than one robot traveling to pick up the same object.

3. Consider a world where multiple robots (like the delivery agent) are moving on a two-dimensional grid towards various destinations. Arrange the BAT so that no location on the grid can be occupied by more than one robot at a time. Consequently,

Figure 5.7: Program file `Examples/GameExamples/pousse-bat.scm`

```
;;;   A basic action theory for the game Pousse.
;;;   Details on the game itself can be found here:
;;;      http://web.mit.edu/drscheme_v424/share/plt/doc/games/pousse.html

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   The action theory: 3 fluents and 1 action

;; game constants
(define N 4)                                ; the size of the board
(define L '(0 1 2 3))                       ; the rows / cols as a list
(define D '(t b l r))                       ; the directions to push
(define N* (- N 1))                         ; last row and column

(define-fluents
  board   (build-array (list N N) (lambda (i j) #f)) ; the game board
  player  'x                                ; the player who plays next
  boards  '())                              ; a list of previous boards

(define-action (push! dir k)                ; push in direction dir on row/col k
  board  (move (trans dir k))               ; get new board from current one
  player (if (eq? player 'x) 'o 'x)         ; the turns alternate
  boards (cons board boards))               ; save current board

;;  Useful abbrevs for access to board
(define (occ i j) (array-ref board (list i j)))
(define (move l) (apply array-set* board l))

;; print the board row by row
(define (print-board)
  (define (printrc i j) (display (or (occ i j) "-")))
  (for ((i L)) (display "  ") (for ((j L)) (printrc i j)) (display "\n")))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   How the board is transformed after a pushing move

;; get a list (l1 p1 ... ln pn) where li is array indices and pi is player
(define (trans dir k)
  (case dir ((t) (pushl player 0  k +1  0))
            ((b) (pushl player N* k -1  0))
            ((l) (pushl player k  0  0 +1))
            ((r) (pushl player k  N* 0 -1)) ))

(define (pushl pl i j di dj)
  (define edge? (or (and (= i 0) (= di -1)) (and (= i N*) (= di +1))
                    (and (= j 0) (= dj -1)) (and (= j N*) (= dj +1))))
  (cons (list i j) (cons pl (if (or edge? (not (occ i j))) '()
                                (pushl (occ i j) (+ i di) (+ j dj) di dj)))))
```

there can be a standoff when two robots are trying to get past each other. Program a scheme (such as a right-of-way convention, or one-way passageways, or something else) to eliminate such standoffs.

4. Write a program to play Qubic, that is, 3D tic-tac-toe on a $4 \times 4 \times 4$ board, using the

Figure 5.8: Program file Examples/GameExamples/pousse-main.scm

```
;;;   The game Pousse in Ergo using the general game player.

(include "pousse-bat.scm")

;; number of cols (resp rows) occupied by player pl in a row (resp col)
(define (num-cols pl i) (for/sum ((j L)) (if (eq? pl (occ i j)) 1 0)))
(define (num-rows pl j) (for/sum ((i L)) (if (eq? pl (occ i j)) 1 0)))

;; sum over all rows (resp cols) of fn of #cols (resp rows) for X and for O
(define (row-counter fn)
  (for/sum ((i L)) (fn (num-cols 'x i) (num-cols 'o i))))
(define (col-counter fn)
  (for/sum ((j L)) (fn (num-rows 'x j) (num-rows 'o j))))

(define (occupancy)                              ; occupancy squared
  (define (sq xs os) (- (* xs xs) (* os os)))
  (+ (row-counter sq) (col-counter sq)))

(define (centrality)                             ; manhattan dist to corners
  (for/sum ((i L))
    (for/sum ((j L))
      (* (case (occ i j) ((x) 1) ((o) -1) (else 0))
         (+ (min i (- N i)) (min j (- N j)))))))

(define (static) (+ (occupancy) (centrality)))   ; static evaluation

(define (winner)                                 ; the winner of a board
  (define (owned xs os) (if (= xs N) 1 (if (= os N) -1 0)))
  (if (member board boards) (if (eq? player 'x) 1 -1)
      (let ((ownership (+ (row-counter owned) (col-counter owned))))
        (if (> ownership 0) 1 (if (< ownership 0) -1 #f)))))

(define all-moves (for/append ((dir D)) (for/list ((k L)) (push! dir k))))

;; X moves first via minimax; O plays via read
(define (main) (ergo-play-game winner all-moves print-board
                               #:static static #:depth 6 #:infinity 512))
```

ergo-play-game function. Considering the Million-Billion Rule on page 35, choose a depth parameter n so that no more than a million nodes of a game tree will be considered per move, and compare the play to having a depth of $(n+1)$.

Chapter 6

Providing Online Control

Cognitive robotics is not just about computing sequences of actions, as with the basic planning seen in Chapter 3 or the programming in Chapters 4 and 5. What cognitive robotics is really about is controlling robots through cognition. The reason we care about producing a sequence of actions is that we intend to use these actions to actually drive a robot or software agent.

In this chapter, we examine the issues that arise in using the output of a planner or an *ERGO* program to control a robot. The architecture of a total system based on *ERGO* is shown in Figure 6.1. There are three main components:

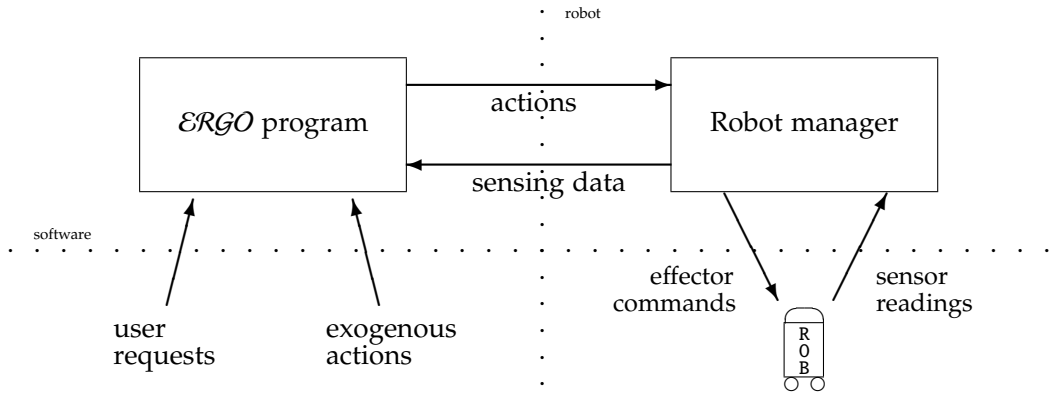
- The *ERGO* component is made up of an *ERGO* program and its associated basic action theory. The expected output of this component is a stream of actions to be performed by the robot. In the simplest case, this component takes no input other than an invocation by the user. In a more complex setting, it receives a stream of sensing data from the robot manager as well as reports of other exogenous actions that have taken place, which may include additional user requests.

- A *robot* is a piece of machinery with effectors and sensors. Once it is turned on, its effectors can be activated by a robot manager, and it can report the state of its sensors to the manager. (A robot may be replaced by a software simulator that displays in some way what its effectors would have done and generates ersatz sensing data.)

- A *robot manager* is a program that sits between the *ERGO* component and the robot. Its job is twofold: it translates requests for action from the *ERGO* program into commands for the robot effectors; it translates readings from the robot sensors into sensing reports for the *ERGO* program as needed. This manager can be written in any computer language that supports the necessary communication. (Typically, it is the need for communication with specialized robotic hardware that ends up determining the required computer language for the manager.)

So far, in all the previous chapters, the basic action theories and *ERGO* programs have been completely shut off from the outside world. The decisions about the actions to perform use information available to the program at the outset, but nothing else. (Although the planning of Chapter 12 also deals with sensing, the actual state of a sensor will not be available to the planner. The best that can be done there will be to plan for all possible sensing results, according again to what is known at the outset.)

Figure 6.1: The architecture of a cognitive robotic system



This is what is called *offline execution*. The cognitive robotics architecture is simple: before the robot does anything, the *ERGO* program (or a planning program) is asked to compute an entire plan, a complete sequence of actions; then the *ERGO* program is put away, and the plan is passed to the robot manager for autonomous execution.

In *online execution*, an *ERGO* program only computes the next action to be performed by the robot manager; the manager then executes this action, possibly returning sensing data to the running *ERGO* program. The *ERGO* program then computes the following action, possibly using information acquired in previous steps, possibly looking ahead to future actions. The process then iterates, perhaps indefinitely. While this is going on, the *ERGO* program may also find out about actions performed exogenously by other agents.

In the first section of this chapter, a complete cognitive robotic system that runs in an offline mode will be presented. But this is mostly just for contrast with the main focus of this chapter, which is online control.

6.1 A complete offline example

We now turn our attention to a simple but complete cognitive robotics system, a simulated elevator, first described by Ray Reiter and presented in Chapter 1. This is not a reactive elevator like the one in Section 5.3.1, as it only attends to call buttons. The elevator can move up or down in one step from a current floor to a destination floor. Call buttons for some of these floors are initially on, and the job of the elevator is get to each of them in some order, turn the call buttons off, and then finally park the elevator on the first floor.

Following the architecture in Figure 6.1, the complete cognitive robotics system for this elevator is made up of three pieces:

- The *ERGO* component, called `basic-elevator.scm`, consists of an *ERGO* program with its associated basic action theory.
- The actual elevator is simulated using a displayed elevator image and call buttons.

Figure 6.2: Program file `Examples/basic-elevator.scm`

```
;;; This is a version of the elevator domain, formalized as in Ray Reiter's
;;; original in Golog, where the actions take numeric arguments.

;; The basic action theory: two fluents and three actions
(define-fluents
   floor 7                    ; where the elevator is located
   on-buttons '(3 5))         ; the list of call buttons that are on

(define-action (up n)         ; go up to floor n
   #:prereq  (< floor n)
   floor n)

(define-action (down n)       ; go down to floor n
   #:prereq  (> floor n)
   floor n)

(define-action (turnoff n)    ; turn off the call button for floor n
   on-buttons (remove n on-buttons))

;; Get to floor n using an up action, a down action, or no action
(define (go-floor n)
   (:choose (:act (up n)) (:test (= floor n)) (:act (down n))))

;; Serve all the floors and then park
(define (serve-floors)
   (:begin
      (:until (null? on-buttons)
         (:for-some n on-buttons (go-floor n) (:act (turnoff n))))
      (go-floor 1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Main program: run the elevator using the above procedure

(define (main) (display (ergo-do #:mode 'first (serve-floors))))
```

- The robot manager, called `elevator-server.scm`, drives the simulation. (In this case, the manager happens to be written in Scheme.)

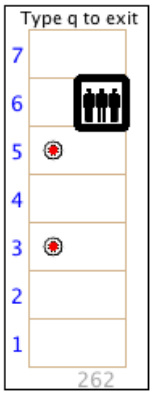
The *ERGO* program `basic-elevator.scm`, first seen in Chapter 1, is displayed again in Figure 6.2. Because the system makes no attempt to serve the floors in an intelligent way, it is much simpler than the programs seen so far. Note that the `main` function calls `ergo-do` using the `'first` mode, meaning that it returns the first successful sequence of actions it finds, which it then prints using `display`, sending the result as a list to standard output:

```
racket -l ergo -f basic-elevator.scm -m
((down 3) (turnoff 3) (up 5) (turnoff 5) (down 1))
```

The actions it generates are intended to send an elevator down to floor 3, turnoff the call button there, then up to floor 5, turnoff the call button there, and finally down to floor 1.

Since the overall system is intended to work in offline mode, the only real requirement of the robot manager is that it be able to "consume" the actions produced by the *ERGO* program and do something interesting with them. For our purposes here, it will read the

Figure 6.3: The elevator server



actions as a single list at the outset. (This protocol would not work well if there were thousands or millions of actions to perform offline.)

Here is how the robot manager `elevator-server.scm` works. It begins by opening a graphical window showing a simple image of an elevator. Then, rather than sending instructions to a real physical elevator in response to the actions it receives as input, it modifies the graphical display as necessary, moving the elevator, turning off call buttons, and so on. (The program uses a number of simulation and graphical facilities provided by Racket Scheme that are beyond the scope of this book. Note that this elevator manager is written in Scheme but has nothing to do with *ERGO*.)

To test this elevator manager manually, load the program into Scheme calling its `main` function, and provide it with a list of actions to read:

```
racket -f elevator-server.scm -m
((up 10) (down 2))
```

Here is what should happen: A narrow window opens showing an elevator shaft with blue numbers along the left side indicating the floor, like the image shown in Figure 6.3. Red call buttons are visible at floors 3 and 5. In the simulation, the elevator begins at floor 7, moves up to floor 10, then down to floor 2, and then the simulation stops. (The simulation can also be ended at any time by typing a `q` in the simulation window.)

The same simulation behaviour can be achieved non-interactively by piping the list of actions through standard input:

```
echo "((up 10) (down 2))" | racket -f elevator-server.scm -m
```

So the simplest way to put this elevator manager together with the *ERGO* control program, is to pipe the actions computed by *ERGO* to the manager through standard input:

```
racket -l ergo -f basic-elevator.scm -m | racket -f elevator-server.scm -m
```

This should cause the simulated elevator to perform the actions determined by the *ERGO* program. Here, for the very first time in this book, all three pieces of a cognitive robotic system from Figure 6.1 have come together in a (simulated) robot.

6.2 The robotic interface for online control

So far, *ERGO* programs have been run in offline mode: given a program, `ergo-do` generates a sequence of actions, a complete legal execution of the entire program. What this means is that a robot cannot get started on even the first action in the program until `ergo-do` has considered the execution of the entire program from start to end. For a realistic program written on many pages of text and requiring thousands of actions, this is impractical.

In online execution, `ergo-do` generates a sequence of actions *incrementally*, sending each action to the robot manager one at a time, allowing it to decide when it is ready to receive the next one. In between these steps, exogenous actions can occur that may change the values of some of the fluents.

6.2.1 The reactive elevator in online mode

Let us now reconsider the reactive elevator presented in Section 5.3.1. The behaviour of this program in offline mode was already discussed. Running this same program in online mode, by using the `'online` argument of `ergo-do`, produces the following

```
> (ergo-do #:mode 'online control)
Starting monitors for 0 exogenous and 0 endogenous interfaces.
Stopping all interface monitors.
'done
```

So nothing happens yet. To make something happen, a source for exogenous actions and a target for endogenous actions must be declared using the function `define-interface`:

`(define-interface` *direction function*`)`

The *direction* argument should evaluate to `'in` for incoming exogenous actions or to `'out` for outgoing endogenous actions (that is, actions produced by *ERGO* programs). The *function* argument should evaluate to a function of no arguments (like `read`) in the case of `'in`, and to a function of one argument (like `display`) in the case of `'out`. For exogenous actions, the input function will be used by `ergo-do` to receive actions from the outside world, and for endogenous actions, the output function will be used by `ergo-do` to send actions to the outside world. (An *ERGO* program can have multiple sources of exogenous actions as well as multiple targets for endogenous ones.)

As a very simple example of an output function, we might use `displayln` itself, as shown in Figure 6.4. In this case, `ergo-do` displays the actions produced by the program one at at time. Of course, `displayln` is not actually getting an elevator to move; it is only simulating sending the actions to an outside world. Moreover, the `displayln` takes almost no time to execute, and so the simulated elevator does all its work in a fraction of a second. A better simulation output function to use is the *ERGO* function `write-endogenous`. This behaves much like `displayln` except that it also pauses for 1.5 seconds, simulating the time it might take for each action to take place on a real robot.

The `'in` interface for exogenous actions is similar. The function `read` can be used to enter simulated exogenous actions: `(define-interface 'in read)`. But a better simulation input function to use is the *ERGO* function `read-exogenous`, which is like `read`, but also displays a convenient prompt.

Figure 6.4: Using `define-interface`

```
> (define-interface 'out displayln)
> (ergo-do #:mode 'online control)
Starting monitors for 0 exogenous and 1 endogenous interfaces.
down!
down!
down!
down!
turnoff!
up!
up!
turnoff!
down!
down!
down!
down!
Stopping all interface monitors.
```

Figure 6.5: Program file `Examples/reactive-elevator-tcp1.scm`

```
;;;  This is an adaptation of the elevator that appears in the IJCAI-97
;;;  paper on ConGolog.  The code is in reactive-elevator-run.scm.

(include "reactive-elevator-run.scm")

;;;  In this version, ERGO is used with TCP in a basic way
;;;     - actions generated by the program are simply printed
;;;     - exogenous actions arrive over TCP port 8234

;;;  To use this elevator, run this program, and in another window, do
;;;     > telnet localhost 8234
;;;  from which exogenous actions can be entered

(define-interface 'out write-endogenous)

(define-interface 'in
  (let ((ports (open-tcp-server 8234)))
    (displayln "Ready to receive exogenous actions!" (cadr ports))
    (lambda () (display "Act: " (cadr ports)) (read (car ports)))))

(define (main) (ergo-do #:mode 'online control))
```

6.2.2 Using a TCP interface

Using both `write-endogenous` and `read-exogenous` as the interface functions is quite messy since the same terminal window is used for both. A better solution is to obtain exogenous actions from a source other than the target for the endogenous ones.

An example is shown in Figure 6.5. In this program, exogenous actions are expected to appear over TCP on port number 8234. What this means is that when `ergo-do` is run online, it becomes a *TCP server*: client processes (such as `telnet`) can connect to it over TCP on port number 8234. The client will then see a prompt `Act:` indicating that the server is

Figure 6.6: Program file `Examples/reactive-elevator-tcp2.scm`

```
;;;  This is an adaptation of the elevator that appears in the IJCAI-97
;;;  paper on ConGolog.  The code is in reactive-elevator-run.scm.

(include "reactive-elevator-run.scm")

;;;  In this version, ERGO is used with a separate TCP action server program
;;;     - exogenous actions for ERGO can be typed in here
;;;     - ordinary and exogenous action are sent to the server on port 8123

;;;  Note the action server program must be running before calling ergo-do.
;;;  It reads actions generated by ERGO and must acknowledge with "ok"

(define-interface 'in read-exogenous)

(define-interface 'out
  (let ((ports (open-tcp-client 8123)))
    (lambda (act)
      (displayln act (cadr ports))
      (or (eq? (read (car ports)) 'ok) (error "bad read from server")))))

(define (main) (ergo-do #:mode 'online control))
```

ready to receive exogenous actions. If the client then produces an exogenous action, that action will be read by the input function, returned to the `ergo-do` server, and executed concurrently with the running *ERGO* program. The client will then be prompted for the next exogenous action.

Note that the value passed as the second argument to `define-interface` is a function of no arguments, as required, but that the `let` expression first opens TCP port 8234 using `open-tcp-server`. In general, the *function* argument of `define-interface` must do whatever initialization is necessary for the input or output function to work properly. (In the implemenattion, the robotic interfaces created by `define-interface` are run as separate processes within Racket during the execution of `ergo-do`.)

To try out the program in Figure 6.5 by hand, call the `main` function of this elevator program, and immediately enter the following in another command-line window:

```
> telnet localhost 8234
Connected to localhost.
Escape character is '^]'.
Ready to receive exogenous actions!
Act:
```

At this point, exogenous actions can be typed in by hand for as long as the main program is running. For example, if `smoke!` is entered, the *ERGO* program will stop the elevator and start producing `ring!` actions, until the `reset!` action is entered. If `(turnon! 8)` is entered, the elevator will eventually go up to floor 8 to turn off the call button there. The `telnet` connection is closed when `ergo-do` terminates (but see `:wait` below).

A different robotic interface for the reactive elevator is shown in Figure 6.6. In this case, exogenous actions are read at the interaction terminal using `read-exogenous`. The output interface, however, is over TCP. This means that when `ergo-do` is run online, it becomes a

TCP client: a server process must already be running and waiting for connections over TCP port 8123. When the *ERGO* program produces an action (such as `up!` or `toggle-fan!`), that action is sent to the server using `displayln`. The *ERGO* client then waits until it receives an `ok` from the server using `read`. At that point, the output function returns, and the execution of the *ERGO* program resumes.

6.2.3 The `:wait` primitive

The reactive elevator stops execution when the elevator is parked on the first floor with no unresolved floor requests, smoke or temperature issues. However, in some online applications, it useful to think of the elevator as running forever. If the elevator has nothing to do, it should simply pause until the next floor request arrives.

One way to achieve this would be to place a loop like

```
(:while #t (:test #t))
```

as the final subprogram in the `:monitor`, after the `(go_floor 1)`. The effect of this would be to have `ergo-do` test the true condition repeatedly until an exogenous action presents something better to do. But this is a form of *busy waiting*: an idle elevator might end up evaluating the `#t` condition millions of times.

A better alternative is offered by the `:wait` primitive. The *ERGO* program

```
(:wait x)
```

behaves just like `(:test #t)` except that it only succeeds after the next exogenous action occurs or x seconds have passed, whichever comes first. (The time argument may be omitted, in which case it may wait forever.) Until then, it does nothing but wait. So if

```
(:while #t (:wait))
```

is included as the last subprogram of `:monitor`, the effect is to do nothing until an exogenous action occurs. At that point, the step is ready to be taken, but it will be preceded by any other higher priority steps necessitated by the exogenous action. Once these are all taken care of, the step is taken and the loop continues by waiting again.

Of course, such a program will never terminate. By using a loop such as

```
(:until stopped? (:wait))
```

as the last subprogram in a `:monitor`, a suitably defined exogenous action `stop!` can make the `stopped?` fluent true and terminate the program. (Care must be taken when using a `:wait` with no timeout in this way since the program will hang if the exogenous action occurs before the `:wait` is executed.)

6.3 Exogenous actions

At this point, it is worthwhile stepping back and reexamining the reactive elevator program in Figure 5.4. As can be seen, the elevator serves the floors whose call buttons are on while monitoring the temperature and smoke alarm. In particular, if the temperature gets too hot, the *ERGO* program will toggle the fan in the elevator, and if it gets too cold, it will do the same.

6.3.1 Sensing via exogenous actions

But in what sense does the *ERGO* system know the temperature in the elevator? In the initial state of the world, the value of the `temp` fluent is given as `0`. The system has complete knowledge of this initial state. And yet, the temperature is expected to change, even though none of the elevator actions change it. So how does the system maintain its complete knowledge about the current temperature? The answer is that it expects to be *told* about temperature changes via exogenous actions. That is, without having to perform any sensing activities on its own, the *ERGO* system is told via an exogenous action when the temperature goes up or down. The effect of a `heat!` event is precisely to inform an *ERGO* program that the `temp` fluent has gone up by one unit. (The `cold!`, `smoke!`, and `reset!` events are similar.) This is a form of *passive sensing* which depends on the occurrence of exogenous actions, in contrast to a more active form of sensing considered below.

In the reactive elevator example, the temperature changes by one unit at a time. But there might just as easily have been an exogenous action that models an arbitrary change in temperature:

```
(define-action (thermometer-response! x)    ; exogenous temperature change
   temp x)
```

In this case, an occurrence of the exogenous event `(thermometer-response! 4)` would signal a running *ERGO* program that the `temp` fluent had changed to `4` regardless of its previous value.

It is, of course, outside the control of an *ERGO* program when exogenous events such as these take place. In the simplest case, the external world might send an exogenous event as soon as a fluent of interest changes. But the world might instead send an exogenous event reporting the current value of a fluent only at regular timed intervals.

Another possibility is that these exogenous events only happen when they are *requested* by a running *ERGO* program. This would be a form of *active sensing*. We might imagine a robot with a number of onboard sensors, such as a thermometer, a sonar, a battery level, and others. An endogenous action such has `thermometer-request!` would change no fluents, but would tell the robot manager that a thermometer reading was requested. The robot manager would then deal with the robot's sensors and eventually cause an exogenous action like `(thermometer-response!` x`)` to happen for some value of x, which would have the effect of changing the temperature fluent to x as above. If there are no other exogenous actions to worry about, an *ERGO* program might include something like

```
(:atomic (:act thermometer-request!) (:wait))
```

to perform the request and wait for the response. (The `:atomic` sequencing ensures that other concurrent tasks do not execute between the request and the `:wait`, during which the exogenous response might occur, causing the `:wait` to then hang.)

6.3.2 Starting and stopping lengthy behaviours

The idea of producing an endogenous action that causes exogenous actions to happen later is a common and very useful feature of online programs. Many of the behaviours that either take time to perform on a robot or that can have a number of possible outcomes are best modeled in this way.

Consider, for example, the motion of a robot from room to room in a building, as in Chapter 3. In the simplest case, there might be a single action `goto!` whose effect is to change the location of the robot from one room to another. This change happens instantaneously in the sense that no other actions (endogenous or exogenous) can happen between the time the motion starts and the time it stops. This is the model of robot motion reflected in the BAT of Figure 3.2.

But of course the motion of an actual robot is far from instantaneous, and in an online context, it might be better to say that there are two or more instantaneous actions that start and stop the behaviour:

```
(define-action (start-goto! dir dest)       ; endogenous start moving
    #:prereq       (eq? status 'stopped)
    status       'moving
    goal-dest     dest)

(define-action (stop-goto! flag)            ; exogenous stop moving
    status       'stopped
    arrived?     flag
    location     (if flag goal-dest location))
```

The first action, `start-goto!`, is an endogenous action that, when it is performed by a program, sets the `status` fluent to `moving` and tells the robot manager to get the robot moving in the proper direction. At this point, the robot is in a "moving" state, and can now consider doing other actions while in that state (via `:conc` or `:monitor`). Note that the `location` fluent is not changed, meaning that it is recording the robot's last *known* location. However, because the motion may go astray, it is not up to the program to simply "decide" to arrive at its destination. Instead, arrival (and termination of the motion) is modeled as something that *happens to the robot*, signalled by an exogenous `stop-goto!` event. So overall, the `goto!` action can be replaced by a sequence like the following:

```
(:begin (:act (start-goto! dir dest))           ; start the robot motion
        (:while (eq? status 'moving) (:wait)))  ; wait for expected arrival
```

In the best case, the robot will arrive at its intended destination, and the `location` fluent will then be updated. But in some cases, the robot may have been stopped by the robot manager because the motion was taking too long, or because the robot bumped into something, or for some other reason. In such a case, the fluent `arrived?` would be false and it would then be up to the *ERGO* program to engage in some sort of failure recovery. (Another way of modeling this is for there to be more than one type of exogenous action that can terminate the motion and change the value of the `status` fluent.)

In this example of robot motion, the behaviour is initiated by an endogenous action and terminated by an exogenous one. Interestingly, user requests can be thought of as just the opposite of this. For example, serving a floor in the reactive elevator is also a behaviour, but one that is initiated by an exogenous action (the `turnon!` action, which requests service) and terminated by an endogenous one (the `turnoff` action, which indicates completion). In this case, between the starting and stopping actions, the elevator is in a "serving a floor" state, reflected by that floor being an element of the `on-buttons` fluent, and is free to be engaged in other concurrent actions

* 6.4 Backtracking and the `:search` primitive

This final section is concerned with backtracking. In fact, nothing has been said so far about how online execution interacts with backtracking. Consider the following program:

```
(:begin
    (:choose (:act a) (:act b))
    (:act c))
```

Suppose that action `b` makes the prerequisite of action `c` true, but that action `a` does not.

In offline execution, `ergo-do` would consider action `a` first, then consider `c`, which would fail, and then backtrack and consider action `b` instead, then `c`, which would now succeed. So there is a single legal execution: action `b` followed by action `c`.

In online execution, however, `ergo-do` generates actions one at a time and sends them to the outside world for execution. Once `:choose` has decided to try action `a`, that choice is final. The action takes place in the world and there may be no way to undo its effect. Without looking ahead to see action `c`, however, there is no way of knowing that action `a` was a bad choice that will later lead to failure.

Why not simply defer the execution of action `a` until it can be determined that there is a successful completion of the entire program with `a` as its first action? One of the motivations of online execution is to generate actions *incrementally* without having to first run through the entire program. Given a program like `(:begin (:act a)` p`)`, the goal is to get the robot started on action `a` before trying to find a successful execution of all of p, which could be gigantic.

This means that in an online context, backtracking is handled in a different way. The program `(:begin` $p_1 \ldots p_n$`)` generates actions incrementally starting at p_1 without waiting to see if there is a path all the way through to the end of p_n. Consequently, an online execution of a program may *fail* in cases where an offline execution would succeed, as in the simple program above.

To guard against this possibility, the *ERGO* primitive

`(:search` $p_1 \ldots p_n$`)`

can be used. In online mode, `:search` behaves like an offline `:begin`. In other words, an action is produced by `:search` and sent to the output interfaces only if a successful execution of all the p_i subprograms in sequence can be found. So the online execution of the program

```
(:search
    (:choose (:act a) (:act b))
    (:act c))
```

will generate `b` as its first action, unlike `:begin`.

Therefore, wrapping an entire program within a `:search` would certainly avoid any online failure. But it would also completely defeat the purpose of incrementality. A better solution is to use the `:search` primitive more selectively.

Consider two subprograms p and q. The program `(:search` p q`)` forces the online execution of the two pieces to be linked together. Before any actions are generated for

program p, it must be the case that a successful execution of both p then q can be found. But the program `(:begin (:search` p`)` q`)`, on the other hand, allows the online execution of the two pieces to be detached. Before any actions are generated for program p, it must be the case that a successful execution of all of p can be found. But the consideration of q is deferred. If we imagine, for example, that q is a very large program, with hundreds of pages of code, perhaps containing `:search` operators of its own, this can make an enormous difference. So by using the `:search` primitive selectively, we obtain the advantages of offline search (or planning) without having to deal with impossibly large search spaces.

One caveat concerns exogenous actions. The `:search` primitive uses `ergo-do` in offline mode to search for an appropriate sequence of actions, which it then sends online to the output interfaces. No exogenous actions are considered during the offline search. However, if an exogenous action occurs while the actions are being sent to the output interfaces, the computed sequence of actions may no longer be suitable, and failure may still occur. This means that the `:search` primitive should only be used in contexts where exogenous actions cannot invalidate the results of the offline search. (A more complex search operator would restart the search after an exogenous action to find a replacement sequence of actions.)

Projects

Chapter 7

A Simulated World

A popular sort of software system consists of a simulated world or environment of some sort inhabited by independent agents that move around and perform actions in that world. Typically, the world has a number of treasures or valued items to be obtained by the agents, as well as dangers or obstacles to avoid. The agents operate in real time but under program control: a program connects to the world somehow, and then decides on the actions the agent will perform, occasionally asking for reports on the current state of the world. (The complete state of the world may also be displayed graphically for external observers to watch.) The simulated world is sometimes presented as a game where the first agent to attain some goal is declared the winner.

In this chapter, we consider a project involving a software agent controlled by *ERGO* working in such a simulated environment. The constraint of operating in real time makes it difficult to engage in deliberative behaviour such as long-term planning, but online *ERGO* remains useful even as a scripting language in settings like these.

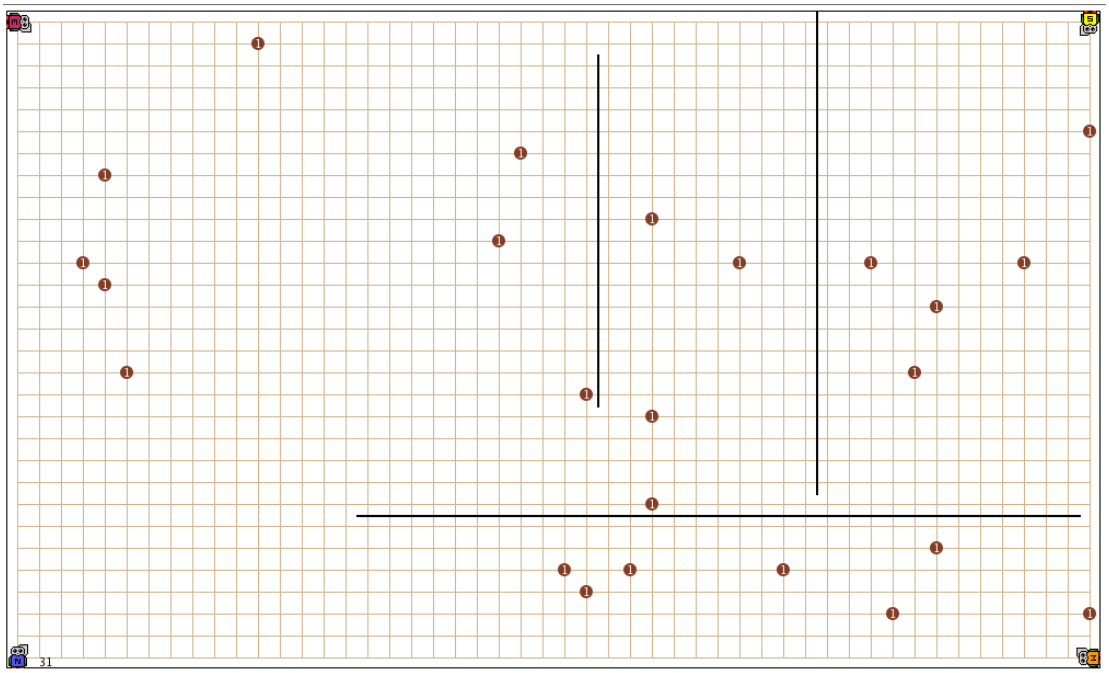
7.1 The Squirrel World

In the Squirrel World (SW) environment, agents are imagined to be squirrels living on a two-dimensional grid. Squirrels have effectors (to do things in the world) and sensors (to gather information). Everything is known to the squirrels at the outset except for the locations of acorns, wall obstacles, and other squirrels. The goal is to be the first squirrel to gather four acorns. (See the SW documentation on how to run this environment.)

Figure 7.1 depicts a typical initial state of the world. As can be seen, the world is divided into a 30×50 grid, and there are a total of four squirrels (called Nutty, Skwirl, Edgy, and Wally) that start out positioned symmetrically at the four corners of the grid

Throughout the game, each acorn and squirrel is located at some point on the grid, and each point on the grid can contain any number of squirrels and acorns. (Acorns are depicted by brown circles with the number of acorns written inside.) Between any two adjacent grid points there may be a wall (depicted with a bold black line) that blocks the passage between the two points. A wall also surrounds the entire world. Note that each squirrel begins with a world boundary to its left and behind it. (The only real difference in the initial locations is that relative to their initial orientation, Nutty and Skwirl deal with a 30×50 grid, whereas Edgy and Wally work in a 50×30 grid.)

Figure 7.1: The Squirrel World



Acorns and walls are distributed randomly on the grid and are completely passive. Squirrels on the other hand have a number of actions at their disposal which are either ordinary actions (that change the world) or sensing actions (that provide information to the squirrel). Each action has a fixed duration. Squirrels have an energy level, the number of time units they have left before they expire. Squirrels start at the maximum level, which is 3000 time units. Energy levels decrease as time advances, but can be increased by eating an acorn. Squirrels can pick up acorns and carry up to two of them. They can also drop an acorn they are holding. Finally, squirrels can build additional wall segments. The first squirrel to be located at a position where there are four acorns wins the game.

7.1.1 The agent actions

Full details on the actions available to a squirrel can be found in the SW documentation. Here is the list of ordinary actions:

action	duration	effect
`left`	1	turn to the left 90 degrees
`right`	1	turn to the right 90 degrees
`forward`	1	move ahead one unit if possible, or be penalized
`pick`	30	pick up an acorn if possible
`drop`	30	drop an acorn if possible
`eat`	60	eat an acorn, boosting energy if possible
`build`	40	build a wall segment

Note that an action may have no effect. For example, if a squirrel attempts a `pick` action and either there is no acorn there or the squirrel is already holding two acorns, then the action still has a duration of 30 time units, but has no effect. (To be able to find out if there is an acorn present, the squirrel can use the `smell` action described below.) Of particular note is the `forward` action: if a squirrel attempts this action and there is a wall segment directly ahead, the action has no effect, but the squirrel is stunned and loses 750 units of energy. (The `look` action described below can be used to check for a wall.)

Here is a summary of the four sensing actions:

action	duration	sensing information
`feel`	1	energy of the squirrel on a scale from 0% to 100%
`look`	10	is there a wall one unit directly ahead?
`smell`	4	are there acorns or other squirrels here?
`listen`	40	coordinates of other nearby squirrels

In addition, the ordinary actions like `pick` return a sensing result which is either `ok` when the action has its intended effect, or `fail` when the action has no effect.

7.2 Interacting with *ERGO*

Once the SW server is running, other programs can connect to it, send it actions, and receive responses. (See the SW documentation for how to do this.) A squirrel agent written in *ERGO* will need to interact with the SW using `define-interface`. So *ERGO* programs will typically contain a BAT that includes actions like `forward` and `look` which can be sent directly to the SW server. But obtaining sensing results requires some fiddling since the SW server does not return the exogenous actions expected by `define interface`.

To facilitate the interaction in both directions, a special Racket program `sw-bridge.scm` shown in Figure 7.2 is loaded at the start of the *ERGO* program. This program sets up the communication between *ERGO* and a running SW server (using `define-interface`), establishes the initial connection over TCP, and defines a global variable `sw-name`, the name of the squirrel agent assigned by the SW server. This bridge should be of use for any project where an *ERGO* program interacts with the SW server.

The basic action theory of the *ERGO* program may define any number of actions as needed. The interface function in `sw-bridge` will send only the following actions to the SW server (and ignore the rest): `forward`, `left`, `right`, `pick`, `drop`, `eat`, `build`, `feel`, `look`, `smell`, `listen`, `quit`. The first seven of these are the ordinary SW actions. The next four actions (`feel`, `look`, `smell`, and `listen`) are the SW sensing actions that `sw-bridge` treats as requests for exogenous reports (as discussed in Section 6.3.1). The argument to the exogenous action will be the sensing result returned by the SW server. Here is a list of the sensing actions and the exogenous actions they produce as defined in `sw-bridge`:

sensing action	exogenous action	argument
`feel`	`(set-energy!` x`)`	a number from 0 to 100
`look`	`(set-view!` x`)`	either `wall` or `nothing`
`smell`	`(set-aroma!` x`)`	a list with two numbers
`listen`	`(set-sound!` x`)`	a list of relative coordinates

Figure 7.2: Program file `Projects/Squirrels/sw-bridge.scm`

```
;;; This is interface code that can be used for any ERGO agent that interacts
;;;   with a Squirrel World server.

;; The SW interface parameters (change as necessary)
(define portnum 8123)                       ; port for SW server
(define machine "localhost")                ; machine for SW server
(define tracing? #f)                        ; print action info for debugging?

;;; Initializing the TCP connection to the SW server
(eprintf "Connecting to the Squirrel World\n")
(define sw-ports (open-tcp-client portnum machine))
(define sw-name (read (car sw-ports)))      ; the first output of SW server
(eprintf "Squirrel ~a ready to go\n" sw-name)

(define sw-acts                             ; acts to send to SW server
  '(feel look smell listen left right forward pick drop eat build quit))
(define sw-responses (hasheq                ; exog responses for sensing acts
  'feel 'set-energy! 'look 'set-view! 'smell 'set-aroma! 'listen 'set-sound!))

;; Define the two ERGO interfaces (using a channel for exog actions)
(let ((chan (make-channel)) (iport (car sw-ports)) (oport (cadr sw-ports)))
  (define (sw-read) (channel-get chan))     ; get exog from sw-write (below)
  (define (sw-write act)                    ; send act over TCP and get response
    (and (memq act sw-acts)
      (let ()
        (displayln act oport)
        (let ((ans (read iport)) (exog (hash-ref sw-responses act #f)))
          (and (eof-object? ans) (error "No response from Squirrel World"))
          (and tracing? (eprintf "Sending: ~a. Receiving: ~a\n" act ans))
          (and (eq? ans 'fail) (eprintf "Warning: Action ~a failed\n" act))
          (and exog (channel-put chan (list exog ans)))))))
  (define-interface 'in  sw-read)
  (define-interface 'out sw-write))
```

In other words, when a `look` action is produced by an *ERGO* program, it will be followed by an exogenous `(set-view! wall)` or `(set-view! nothing)` action, depending on what is in front of the squirrel. It is then up to the *ERGO* basic action theory to decide which fluents change as the result of these exogenous actions.

The communication method used by `sw-bridge` is worth noting as it may be useful in other contexts. The main thing to observe is that the `'in` interface (for exogenous actions) defined by the internal procedure `sw-read` cannot simply read from the TCP port. Instead, `sw-read` attempts to extract an entry from an internal channel, blocking when that channel is empty. It will be up to the other internal procedure `sw-write` to put exogenous actions into that channel. So `sw-write` really does all the communication work. Given an action performed by an *ERGO* program, `sw-write` first decides using `sw-acts` whether the action should be sent to the SW server. If so, it sends the action with `displayln` and obtains the response immediately with `read`. If the action was a sensing action according to `sw-responses`, it then constructs an appropriate exogenous action (as described above) and puts that action in the channel for `sw-read`.

Figure 7.3: Program file `Projects/Squirrels/simple-main.scm`

```
;;;  This program provides agent behaviour in the Squirrel World (SW)
;;;  The squirrel finds a wall ahead, and then runs back and forth.

(include "sw-bridge.scm")              ; define the interfaces to SW
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Basic action theory

(define-fluents
   steps  0                            ; the number of steps taken
   seen   'nothing)                    ; the last thing seen by "look"

;;  squirrel actions
(define-action left)                   ; turn left
(define-action look)                   ; look ahead
(define-action forward                 ; go forward
   steps  (+ steps 1))

;;  exogenous action
(define-action (set-view! x)           ; report from last look action
   seen   x)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Control program

(define (count)                        ; go forward until a wall is seen
   (:until (eq? seen 'wall) (:act forward) (:act look) (:wait)))

(define (run n)                        ; forever do 2 lefts and n forwards
   (:while #t (:act left) (:act left) (:for-all i n (:act forward))))

(define (main)                         ; overall: count then run
   (ergo-do #:mode 'online (:begin (count) (:let ((n steps)) (run n)))))
```

7.2.1 A simple squirrel

A simple but complete *ERGO* program constructed along these lines is shown in Figure 7.3. This is for a squirrel that ignores acorns and other squirrels and just runs back and forth, like the `wall-follower` sample program that comes with the SW.

The basic action theory only uses the `left`, `look`, and `forward` endogenous actions, as well as the exogenous `set-view!` action. The effect of the `left` action is not modelled (that is, the squirrel does not need to keep track of what its effect is). The `forward` action is modelled as incrementing a fluent `steps`, but this is only used in the first part of the *ERGO* program. The `look` action has no effect, but it does cause a subsequent `set-view!` exogenous action that updates the value of the `seen` fluent to `wall` or to `nothing`. (Note that in this BAT, only `set-view!` changes the `seen` fluent. So, for example, after doing a `left` action, the fluent is unchanged. This suggests that the fluent should be read as "the last thing seen by a `look` action" rather than "what is directly ahead.")

The *ERGO* program itself uses two procedures:

- `(count)` is an *ERGO* program that repeatedly sends the squirrel forward (in its initial orientation) until a wall is seen. Note that the program performs the `look` action and then waits for the exogenous report before testing the value of the `seen` fluent.

- `(run` n`)` is an *ERGO* program that repeatedly turns the squirrel around (by performing two `left` actions) and then does n `forward` actions. Note that this program does not look for any walls, and so the squirrel moves at its top speed.

The `main` program performs the `(count)` procedure, then the `(run` n`)` procedure where n is the value of the `steps` fluent immediately after the `(count)`. (Note that `:let` is used to ensure that n has the value of `steps` after the `count`. See Section 4.1.6.) The effect is to have the squirrel walk cautiously up to the first wall ahead of it (perhaps the boundary at the other side of the world), and then to run quickly back and forth between that wall and the origin. The `main` program runs until the squirrel dies, and so the only way to stop it before then is to stop the SW server (by typing a `q` in the SW server window).

Once the SW server is running, the simple squirrel program can be run using

```
> racket -l ergo -f simple-main.scm -m
```

The actual behaviour observed in the SW window will of course depend on which squirrel ends up being connected to the program and on the locations of the walls.

7.3 A foraging squirrel project

The file `systematic-main.scm` contains the main *ERGO* program for a second much more complex squirrel. This one does a systematic search for acorns near its home, stashing them in a special nest area, while monitoring its hunger, until it has found four of them.

7.3.1 The main program

The program has many pieces, but the essence of the behaviour can be seen in the `main` procedure displayed in Figure 7.4. The behaviour is controlled by a `:monitor` which, at the lowest priority, searches for acorns along two dimensions of the grid.

A note about orientation: The absolute orientation of the grid is not used anywhere. The squirrel identifies its initial orientation as "north" and the orientation to its right as "east." (So there are boundary walls to its south and to its west initially.) The directions north and east are not equivalent: one of them has 30 units maximum and the other has 50. (The `sw-name` defined by `sw-bridge` determines whether north has 30 or 50 units.)

Within the `:monitor` of the `main` procedure, there are three subprograms: at the highest priority, `low-energy` handles what to do when the energy of the squirrel gets low; at the intermediate priority, `acorns-present` does what is necessary when the the squirrel detects an acorn; and finally, at the lowest priority, there is the search behaviour. The search subprogram repeatedly strolls along its north direction watching for walls and acorns; when it sees a wall, it runs back to the x-axis (like the squirrel of the previous section), takes one step east, and the search continues. (More details on these programs below.)

Once the SW server is running, the file `systematic-main.scm` can be loaded and run just like `simple-main.scm`. Unless the squirrel finds itself in an unfortunate environment (with too few acorns or too many walls), it should eventually collect its four acorns.

Figure 7.4: Program file `Projects/Squirrels/systematic-main.scm`

```
(include "sw-bridge.scm")
(include "systematic-bat.scm")
(include "systematic-procs.scm")

(define (run-to x y)         ; run to (x,y) but east/west on x-axis only
  (:begin
    (:for-all i yposition (go-dir 'south))
    (:if (< x xposition) (:for-all i (- xposition x) (go-dir 'west))
        (:for-all i (- x xposition) (go-dir 'east)))
    (:for-all i y (go-dir 'north))))

(define (do-at-nest prog)    ; run to (1,1), do prog, then run back
  (:let ((x xposition) (y yposition)) (run-to 1 1) prog (run-to x y)))

(define (stroll-north)       ; walk north, smelling and looking for walls
  (:until (or (eq? seen 'wall) (> yposition (/ ymax 2)))
    (go-dir 'north)
    (check smell) (check look)))

(define (main)
  (ergo-do #:mode 'online
    (:monitor (low-energy) (acorns-present)
      (:while #t             ; systematic search of grid
         (stroll-north)             ; walk north as far as possible
         (run-to xposition 0)       ; run back to x-axis
         (go-dir 'east)             ; take one step east
         (check feel) (check smell) (status-report)))))
```

7.3.2 The basic action theory

The fluents and actions used by this program are shown in Figure 7.5. The `energy`, `carrying`, `stashed`, `xposition`, `yposition`, and `direction` fluents have the obvious values. The fluents `seen` and `smelled` are used for sensing: `seen` is `wall` or `nothing` after a `look`, and `smelled` is the number of acorns after a `smell`. (The number of colocated squirrels after a `smell` is ignored.)

The actions used by the program are a subset of those recognized by the `sw-bridge`. The action `forward` changes the `xposition` and `yposition` fluents; the `left` and `right` actions change the `direction` fluent (using global alists `dirL` and `dirR` defined below); the `eat` and `drop` actions decrement the `carrying` fluent, while `pick` increments it; the `stashed` fluent is incremented by a `drop` since that action only takes place at the nest site; finally, a `pick` action at the nest site decrements the `stashed` fluent.

A note about the nest: A squirrel can only carry two acorns, so in this program, the squirrel deposits those it cannot carry at a fixed location, its nest. This nest could be at the origin $(0, 0)$. However, this has the minor disadvantage that another squirrel casually following a boundary wall could easily find the acorns and steal them. So the squirrel nest is positioned at $(1, 1)$. (There are other options for hiding a stash discussed later.)

Figure 7.5: Program file `Projects/Squirrels/systematic-bat.scm`

```
(define-fluents
  seen 'nothing smelled 0 energy 100 carrying 0 stashed 0
  xposition 0 yposition 0 direction 'north)

;;  motion actions
(define-action quit)
(define-action forward
   xposition (+ xposition (case direction ((east) 1) ((west) -1) (else 0)))
   yposition (+ yposition (case direction ((north) 1) ((south) -1) (else 0)))
   smelled   0
   seen      'nothing)
(define-action left
   direction (cadr (assq direction dirL))
   seen      'nothing)
(define-action right
   direction (cadr (assq direction dirR))
   seen      'nothing)

;;  acorn actions
(define-action eat
   carrying  (- carrying 1))
(define-action drop
   carrying  (- carrying 1)
   stashed   (+ stashed 1))
(define-action pick
   carrying  (+ carrying 1)
   stashed   (if (and (= xposition 1) (= yposition 1))
                 (- stashed 1)  ; picking from stash
                 stashed))      ; picking from elsewhere

;;  sensing actions
(define-action feel)
(define-action look)
(define-action smell)

;;  exogenous actions
(define-action (set-energy! x) energy x)
(define-action (set-view! x)  seen x)
(define-action (set-aroma! x) smelled (car x))
```

7.3.3 Procedures used by the squirrel

All the remaining procedures and abbreviations used by the squirrel are shown in Figure 7.6. (Some already appeared in Figure 7.4.) Here are the details:

- The alists `dirL` and `dirR` are used to map one direction into another direction, the one that results from turning left or right.
- All the sensing done by the squirrel passes through the procedure `(check` *a*`)`, which performs the sensing action *a* and waits for an exogenous report.

Figure 7.6: Program file `Projects/Squirrels/systematic-procs.scm`

```
;;  position or direction abbrevs
(define dirL '((north west) (west south) (south east) (east north)))
(define dirR '((north east) (west north) (south west) (east south)))
(define ymax (case sw-name ((Edgy Wally) 50) (else 30)))

;; do sensing action a and wait for exogenous response
(define (check a) (:atomic (:act a) (:wait)))

;; action procedures
(define (face-dir dir)       ; rotate to face dir
  (:if (eq? dir direction) :nil
       (:if (eq? dir (cadr (assq direction dirR))) (:act right)
            (:if (eq? dir (cadr (assq direction dirL))) (:act left)
                 (:begin (:act right) (:act right))))))

(define (go-dir dir)         ; face dir, then one step forward
  (:begin (face-dir dir) (:act forward)))

(define (dump-at-nest)       ; drop all carried acorns at nest
  (do-at-nest (:for-all i carrying (:act drop))))

(define (status-report)
  (:>> (printf "Energy=~a Carrying=~a Stashed=~a\n" energy carrying stashed)))

(define (acorns-present)     ; behaviour when acorns are smelled
  (:while (> smelled 0)
     (:when (= carrying 2) (dump-at-nest))
     (:act pick)
     (:when (= 4 (+ carrying stashed)) (dump-at-nest))  ; I win!
     (check feel) (check smell)))

(define (low-energy)         ; behaviour when feeling hungry
  (:when (< energy 25)
     (:>> (printf "*** Low energy! ")) (status-report)
     (:when (= carrying 0)
        (:if (> stashed 0) (do-at-nest (:act pick)) (:act quit)))   ; I lose!
     (:act eat) (check feel)))
```

- The procedure `face-dir` turns as necessary to face a given direction.
- The procedure `go-dir` faces in a given direction, then takes a single step forward.
- The procedure (`run-to` x y) gets to position (x, y) by moving quickly through the grid (using previously travelled routes). It goes south until it is on the x-axis, then west or east until it is at $(0, y)$, and then north until it is at (x, y).
- The procedure (`do-at-nest` p) does the following: it remembers its current location (x, y), it runs to $(1, 1)$, executes p, and then runs backs to (x, y).
- The procedure `dump-at-nest` uses `do-at-nest` to get to the nest, and there drops all the acorns being carried.
- The procedure `status-report` prints information about the state of the squirrel.

The remaining procedures determine the top-level behaviour of the squirrel:

- The procedure `stroll-north` walks north using `go-dir` and at each step, smells for an acorn and checks for a wall. It continues doing this until the squirrel sees a wall or gets outs of its quadrant (when its `yposition` exceeds half of `ymax`).
- The procedure `acorns-present` does the following when at least one acorn has been detected: If the squirrel is already carrying two acorns, it first dumps both of them in the nest. Then the squirrel picks up an acorn and, if it now has four in total, it goes to the nest to claim victory. Otherwise, it checks its health and smells for more acorns at the same spot.
- The procedure `low-energy` does the following when the energy of the squirrel is found to be low: If the squirrel is not carrying any acorns, it first goes to get one from the nest. (If there are none in the nest either, the squirrel gives up and quits.) It then eats one of the acorns it is carrying.

7.4 Extensions and variants

While the *ERGO* program above performs reasonably well, there are a number of design decisions that could be reconsidered in a project.

First of all, consider the idea of a nest. If there happens to be a wall blocking passage to $(1,1)$, the squirrel will die without getting its four acorns. Of course, some other nest location could be used. Perhaps it would be best if a nearby nest site were chosen *randomly* at runtime (and remembered by the squirrel, of course), so that other squirrels would have no way of finding the stash without searching. (Note that the squirrel needs to skip over its own nest site when it is out looking for new acorns.) Another way to protect the stash is to build some sort of wall structure around the nest. A two-segment L-shaped wall formation would block simple searches from nearby boundary axes.

Of course nothing stops our own squirrel from attempting to raid other stashes. The current program spends most of its time in its own quadrant. It is easy to imagine a squirrel that first strolls north from $(0,0)$, then from $(1,0)$, from $(2,0)$, and so on as before, but then at some point, goes east as far as it can and then continues strolling north from $(x,0)$, then from $(x-1,0)$, from $(x-2,0)$, and so on. Without losing much time, this could give the squirrel a chance of finding acorns stashed by its eastern competitor.

What might be best, however, is to do without a nest completely. When the squirrel finds an acorn, it could leave the acorn there, note the current location in a list, and continue searching. When the squirrel is hungry or has seen four acorns, it could then go through that list, picking up and dropping acorns as necessary. This would have the advantage of minimizing those somewhat expensive `pick` and `drop` operations.

A more difficult issue concerns the search program itself. After strolling north, the squirrel returns to the x-axis and moves one step east. But what if there is a wall blocking the passage from $(x,0)$ to $(x+1,0)$? The current program will simply bump into that wall, pay a large penalty, and eventually die. Is there something better? One possibility is to try to get around the wall. But this can be quite challenging in the most general case. For example, a wall might be the vertical stem of a T-shaped wall formation. Furthermore, the horizontal part of the T-shape might be the stem of yet another T-shaped formation on its

Figure 7.7: Program file `Projects/Squirrels/random-main.scm`

```
(include "sw-bridge.scm")
(include "systematic-bat.scm")
(include "systematic-procs.scm")

(define-fluent path '())
(define-action push-path! path (cons direction path))
(define-action pop-path! path (cdr path))

(define (revdir dir)             ; the 180 direction from dir
  (cadr (assq (cadr (assq dir dirL)) dirL)))

(define (do-at-nest prog)        ; go to nest, do prog, then return
  (:begin (:for-all d path (go-dir (revdir d)))
          prog
          (:for-all d (reverse path) (go-dir d))))

(define (take-step r)            ; try a step in a random dir according to r
  (define (try-in-order dirs)
    (:begin (face-dir (car dirs)) (check look)
            (:if (eq? seen 'nothing)
                 (:begin (:act push-path!) (:act forward) (check smell))
                 (try-in-order (cdr dirs)))))
  (try-in-order
    (if (< r .48) '(north east west south)
        (if (< r .96) '(east north south west) '(south west north east)))))

(define (path-restart)           ; go back to nest and start a new path
  (:until (null? path) (go-dir (revdir (car path))) (:act pop-path!)))

(define (main)
   (ergo-do #:mode 'online       ; random search of grid
     (:begin (go-dir 'north) (go-dir 'east)        ; start from (1,1) location
        (:monitor (low-energy) (acorns-present)
          (:while #t
             (path-restart)                        ; start search over
             (:for-all i 35 (take-step (random)))  ; take 35 random steps
             (check feel) (status-report))))))
```

side. In other words, the squirrel might actually be in some sort of wall *maze* that might be quite hard to exit (without starving).

One radical solution to this problem is to perform a *non-systematic* search of the grid, where a squirrel moves somewhat randomly, always endeavouring to visit new territory, while looking for walls and smelling for acorns as necessary. This requires a redesign of the procedure and its basic action theory.

A simplified version of a randomized searching squirrel is shown in Figure 7.7. It uses the same BAT and procedures as the systematic squirrel, but with a few modifications. It uses a new fluent `path` to keep track of all the directions it has taken since the last time it left the nest at $(1,1)$. It uses the same `low-energy` and `acorns-present` procedures as before, but this time, the `do-at-nest` procedure uses the saved path (which is known to be clear of walls). The search procedure in this case involves getting the squirrel to take 35

steps using `take-step`, and then running back to the nest and starting on a new path. The `take-step` procedure uses `try-in-order` to attempt one step in a list of possible directions, taking the first one that is not blocked by a wall. The order of directions is determined randomly: 48% of the time it tries north, then east, then west, then south; 48% of the time it tries east, then north, then south, then west; 4% of the time it tries something different: south, then west, then north, then east. Overall, this means the squirrel heads somewhat to the north and to the east, unless there are walls blocking it. Unlike the systematic squirrel, it will not be stymied by walls, including walls that touch the x-axis (or any boundary).

Assuming the SW server is running, the `random-main` program can be run the same way as `systematic-main`. There are cases where it will do better. But overall, it appears to spend too much time looking for walls and smelling for acorns at the same nearby locations. Much better would be to somehow keep track of all the "clear space" it has visited, and to return to its nest only when it has too many acorns, and then to run back to a perimeter of that clear space. While the search for acorns can be random, the goal should always be to expand the clear space. This would preserve the advantages of the systematic search, while avoiding its drawbacks.

Chapter 8

A LEGO Robot

After all this time talking about cognitive robotics, we are finally ready to work on an actual robot. Perhaps the simplest programmable robot is the LEGO Mindstorms, available as an inexpensive kit. The kit includes plastic LEGO pieces to build a variety of robots, as well as a collection of sensors and motors. In this chapter, we consider a project that involves a LEGO robot controlled by *ERGO* that moves on a small surface, picking up items at various locations and delivering them to other locations, somewhat like the delivery example of section 5.2.1. In terms of hardware, all that is needed for this robot beyond the LEGO kit is a micro-SD card (to install the EV3dev operating system) and a USB Wifi dongle (to allow wireless TCP communication with *ERGO*).

8.1 The EV3 Brick and EV3 Python

The LEGO Mindstorms kit comes with what is called the EV3 Brick, a small battery-operated computer that can be connected to sensors and to motors and can be programmed to control them. A picture showing an EV3 Brick connected to the three motors (at the top) and four sensors (at the bottom) appears in Figure 8.1. Typically, a robot is assembled out of LEGO pieces and includes the Brick with some connected sensors and motors, like the two vehicles shown in Figure 8.2. Code for the EV3 Brick is normally prepared on an auxiliary computer, and then downloaded to the Brick for execution.

There are a variety of programming languages that can be used with the Brick (including a graphical one provided by LEGO), but here we use Python with a special EV3 library that is included with EV3dev, a Linux-based operating system for the Brick. So in this chapter, a robot manager written in Python will be interacting with an *ERGO* program over TCP. (It is possible to use Bluetooth as well, but TCP is much easier.)

A full account of EV3 Python and the EV3dev operating system are beyond the scope of this book, but there is considerable material online, including tutorials and many examples. See `http://www.ev3dev.org/` for information about EV3dev, including how to install it on an micro-SD card to boot the EV3 Brick, and the sites

```
http://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/
https://sites.google.com/site/ev3python/
```

for details on the EV3 Python library.

Figure 8.1: The LEGO EV3 Brick

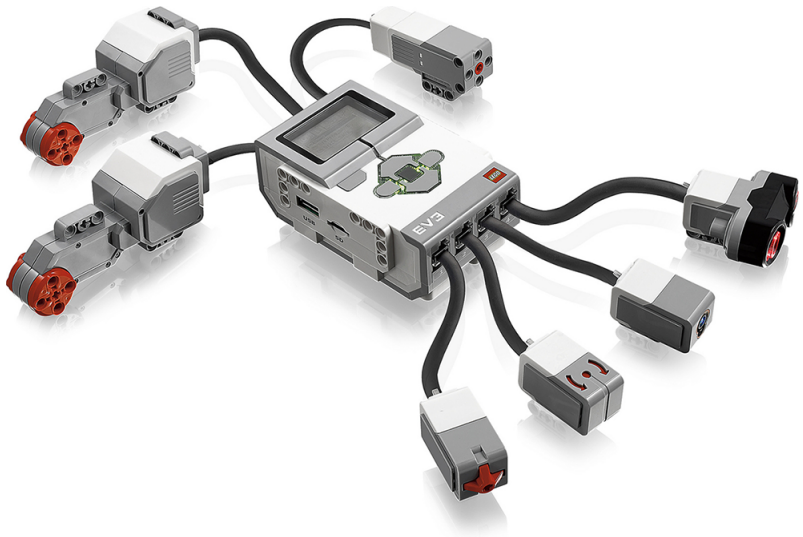
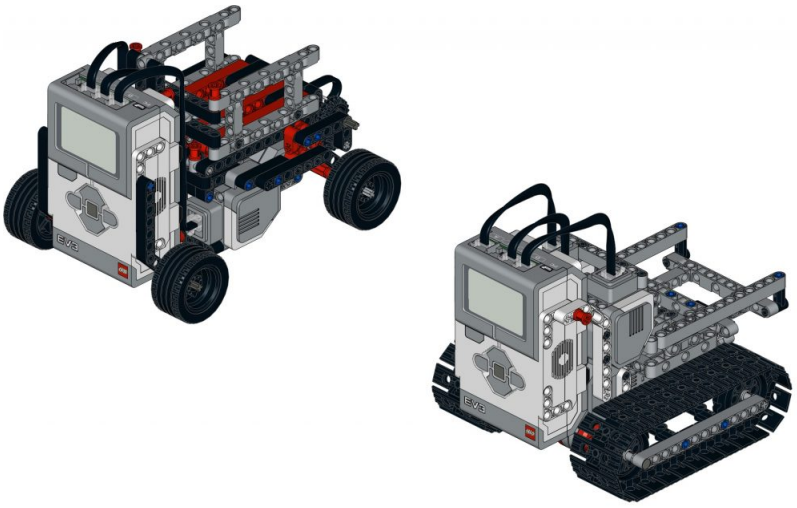


Figure 8.2: Two LEGO vehicles



To give a sense of what an EV3 Python program looks like, two small ones are shown here. Figure 8.3 shows EV3 Python code that gets a vehicular robot to drive in a square pattern (or close) by dead reckoning, assuming motors on two wheels are connected to out ports on the Brick. Figure 8.4 shows Python code that gets the vehicle to continue driving forward until a sensor detects reflected light brighter than a certain threshold, again assuming the light sensor is connected to an in port on the Brick.

Figure 8.3: Program file Servers/EV3/misc/square.py

```
import time
import ev3dev.ev3 as ev3

# motors must be connected to out ports A and D
left_wheel = ev3.LargeMotor('outA')
right_wheel = ev3.LargeMotor('outD')

# drive both motors for ms milliseconds
def straight(ms):
        left_wheel.run_timed(time_sp=ms,speed_sp=500)
        right_wheel.run_timed(time_sp=ms,speed_sp=500)
        time.sleep(ms/1000.0)

# turn either left (dir=1) or right (dir=-1)
def turn(dir):
        left_wheel.run_timed(time_sp=360,speed_sp=-500*dir)
        right_wheel.run_timed(time_sp=360,speed_sp=500*dir)
        time.sleep(.5)

# drive a squarish pattern
def square(ms):
        for i in range(0,4):
                straight(ms)
                turn(1)

square(1500)
```

Figure 8.4: Program file Servers/EV3/misc/threshold.py

```
import time, ev3dev.ev3 as ev3

# motors must be connected to out ports A and D
left_wheel = ev3.LargeMotor('outA')
right_wheel = ev3.LargeMotor('outD')

# the color sensor must be connected to an in port
color_sensor = ev3.ColorSensor()
color_sensor.mode = 'COL-REFLECT'    # report a value from 1 to 100

# drive both motors at speed sp until told otherwise
def straight_forever(sp):
        left_wheel.run_forever(speed_sp=sp)
        right_wheel.run_forever(speed_sp=sp)

# drive forward while color sensor < t
def threshold(t):
        straight_forever(200)
        while (color_sensor.value() < t): time.sleep(.1)
        straight_forever(0)

threshold(50)
```

Figure 8.5: Program file `Projects/LEGO/tag-bridge.scm`

```
;;; This is interface code that can be used for an ERGO agent that
;;;   uses tagged actions for online interactions

;; this calls define-interface after modifying readfn and printfn to use tag
(define (define-tagged-interfaces tag readfn printfn)
  (define (read-add-tag)
    (let ((r (readfn)))
      (if (symbol? r) (list r tag) (cons (car r) (cons tag (cdr r))))))
  (define (print-detag a)
    (and (not (symbol? a)) (not (null? (cdr a))) (eq? (cadr a) tag)
         (printfn (cons (car a) (cddr a)))))
  (define-interface 'in read-add-tag)
  (define-interface 'out print-detag))

;; setup in and out interfaces over TCP
(define (tag-tcp-setup tag portnum IPaddress)
  (eprintf "Setting up interfaces over TCP for ~a\n" tag)
  (define tcp-ports (open-tcp-client portnum IPaddress))
  (define-tagged-interfaces tag
    (lambda () (read (car tcp-ports)))
    (lambda (act) (displayln act (cadr tcp-ports))))
  (eprintf "~a is ready to go\n" tag))

;; setup in and out interfaces with standard IO
(define (tag-stdio-setup)
  (eprintf "Setting up interfaces over stdin and stdout\n")
  (define-tagged-interfaces 'user read-exogenous write-endogenous))
```

8.2 Interacting with *ERGO*

To interact with *ERGO*, a robot manager needs to be able to perform the endogenous actions generated by the *ERGO* program as well as to signal when exogenous actions occur. The ability to use TCP for this interaction simplifies the job considerably. In a sense, the only real design decision left is what those primitive actions should be.

However, we do want to allow for the possibility of primitive actions in the BAT that are not meant for the EV3 Brick. (They may be used internally by *ERGO*, or for interacting with a user, or for some other purpose.) For this reason, all the actions of the BAT will take a distinguished first argument, a tag indicating where the action should be sent over an interface (for endogenous actions) or from where the action was received over an interface (for exogenous ones). For example, if a robot called `robo1` signals `wall_detected!`, the interface we use will translate this to the exogenous action `(wall_detected! robo1)` for *ERGO*. If *ERGO* produces the endogenous action `(go-location! robo1 loc5)`, the interface will send `(go-location! loc5)` to `robo1`. If the action produced is `(tell! user done)`, then nothing will be sent to `robo1`, but `(tell! done)` will be sent to `user` instead.

So to summarize, *ERGO* will deal with primitive actions that are *tagged*, that is, have a first argument indicating the source or destination of the action. The destinations will receive untagged actions, and the sources will be expected to send untagged actions.

Figure 8.6: Program file `Projects/LEGO/test-manager1.scm`

```
;;; This program uses the EV3 Robot Manager 1 to perform the actions below.
;;; Once the robot manager is running, this ERGO program can be run by
;;;     racket -l ergo -f test-manager1.scm -m <IPaddress>
;;; where <IPaddress> is the address of the EV3 machine.

(include "tag-bridge.scm")

(define-fluents light 100)

;; endogenous
(define-action (run_motor! r x))
(define-action (req_sensor! r))
;; exogenous
(define-action (reply_sensor! r z) light z)

(define (main . args)
  (and (null? args) (error "Must supply an IP address for EV3"))
  (tag-tcp-setup 'my-EV3 8123 (car args))       ; port 8123 assumed
  (ergo-do #:mode 'online
     (:begin (:act (run_motor! 'my-EV3 2000))   ; run the motor for 2000 ms
             (:act (req_sensor! 'my-EV3))       ; ask for a sensor reading
             (:wait)                            ; wait for value returned
             (:>> (printf "The light had value ~a\n" light)))))
```

8.2.1 Programming the *ERGO* side

Any *ERGO* program intending to interact with the EV3 Brick using tagged actions should load the file `tag-bridge.scm` shown in Figure 8.5. This defines a function `tag-tcp-setup` which makes it easy to connect to a running EV3 server and deal with tagged actions. The *ERGO* program can then run online in the usual way, sending endogenous actions and receiving exogenous ones. Note that `tag-tcp-setup` needs to be called with three arguments: a tag identifying the EV3 being used, the TCP port (something like `8123`) and the IP address of the running EV3 Brick (like `"192.168.0.140"` or `"EV3robot.cs.pedley.edu"`). The function `tag-stdio-setup` in this file can also be used for interacting with the user on the terminal.

A very simple *ERGO* program using this file is shown in Figure 8.6. Of course this program will not work properly until a robot manager is running on the EV3.

8.2.2 Programming the EV3 side

On the EV3 side, it is the job of the programmer to decide what to do with each of the endogenous actions it will receive from *ERGO*, as well as when it should consider signaling an exogenous action. The actual communication with *ERGO* over TCP is always the same and is provided here by a library called `ergo_communication.py`. Any robot manager written in Python should import two functions from this file:

- `signal_exogenous(actName,args)`
  This function takes two arguments, `actName`, which is a string, like `'respond!'` or `'wall-detected!'`, and a list of arguments for the action, each of should be a number

or a string. The function sends the action (converted into a Scheme list) over TCP to the *ERGO* system.

- `ergo_tcp_session(portnum,initialize,handle_endogenous)`
  This function starts a TCP server on the EV3 waiting for a connection on the port given by the `portnum` argument, and terminates when the session closes. The other two arguments are functions that are called after a connection is made:
  - `initialize()` must be a function of no arguments defined somewhere in the robot manager program that will be called at the start of the TCP session;
  - `handle_endogenous(actName,args)` must be a function of two arguments defined in the robot manager that will be called every time an endogenous action is received by the server. The arguments to this function are like those of `signal_exogenous`, but this time for endogenous actions.

Note that everything that the robot manager does during a TCP session is as a result of its `initialize` or `handle_endogenous` arguments. So it is up to the `handle_endogenous` argument to do what it takes for every possible endogenous action it can receive. Typically, the body of this function will be something like this:

```
if actName == 'act1': proc1()
elif actName == 'act2': proc2(args[0],args[1])
elif actName == 'act3': proc3('***')
          ...
elif actName == 'act10': proc10(17,args[0])
else: print('WARNING: action ignored')
```

where the $\texttt{act}_i$ are the names of all the endogenous actions that can be received from *ERGO*, and the $\texttt{proc}_i$ are Python functions defined in the robot manager. These Python functions will go on to use the EV3 library to control the motors, read from the sensors, and then call the function `signal_exogenous` as necessary. It is a good idea when developing a robot manager to begin with stubs for these $\texttt{proc}_i$ functions that simply use `print` and `input` operations instead of motors and sensors.

8.2.3 Putting the two pieces together

Before trying out a robot manager, it perhaps useful to ensure that TCP is working as expected on the EV3 Brick. (First, of course, the EV3 must be connected to the local area network using the USB Wifi dongle, and the IP address of the EV3 Brick must be known.) The program `test_server.py` can be used to test the workings of TCP. When run by Python on an EV3 Brick, it should say that it is listening for a connection on port 8123 and then pause. In a terminal window on a different machine in the same local area network, it should then be possible to connect to the EV3 manually using `telnet`:

```
> telnet 192.168.0.140 8123
Connected to 192.168.0.140.
Escape character is '^]'.
```

Figure 8.7: Program file `Servers/EV3/manager1.py`

```
#  This is a robot manager for an EV3 brick interacting with ERGO over TCP
#  It accepts the following endogenous actions:
#       - (run_motor! t), where t is in milliseconds
#       - req_sensor!
#  It generates the following exogenous actions:
#       - (reply_sensor! n), where n is between 1 and 100

import sys, ev3dev.ev3 as ev3
from ergo_communication import signal_exogenous, ergo_tcp_session

moto = ev3.LargeMotor('outA')    # need a motor on out port A
colo = ev3.ColorSensor()         # need a color sensor on an in port
colo.mode = 'COL-REFLECT'        # color sensor returns 1 to 100

# Run the motor on port A for t milliseconds
def run_motor(t):
    moto.run_timed(time_sp=t, speed_sp=500)

# Return a number from 1 to 100 in a reply_sensor! exogenous action
def req_sensor():
    signal_exogenous('reply_sensor!',[colo.value()])

#########################################################################
###   The two procedures needed for ergo_tcp_session

def ini():
    ev3.Sound.speak('Starting Manager One').wait()
    print("Starting Manager 1")

def dispatch(actName,args):
    if actName=='run_motor!': run_motor(args[0])
    elif actName=='req_sensor!': req_sensor()
    else: print('No dice')

ergo_tcp_session(8123,ini,dispatch)
```

At this point, every time a number n is typed in the `telnet` session, the EV3 should respond with the number $n + 7$, until the session is terminated.

We are now ready to try the program `manager1.py`, a simple robot manager for the EV3 shown in Figure 8.7. Note that the two function arguments to `ergo_tcp_session` here are `ini` and `dispatch`. The `dispatch` function is prepared to deal with the two endogenous actions produced by the *ERGO* program `test-manager1.scm` from Figure 8.6, and to send to that program the exogenous action it is expecting. So once Python is running `manager1.py` on the EV3, we can test its operation by running `test-manager1.scm` under *ERGO* on a different computer in the same local area network and, if all is working well, we get a LEGO motor to spin for two seconds and the LEGO sensor value printed.

This, then, is a complete cognitive robotic system of the sort shown in Figure 6.1. All that is left to do is to give it something more interesting to think about.

8.3 A delivery robot project

The project we want to consider in this chapter involves a robot delivery service. The idea is that the LEGO robot is a vehicle of some sort that can move around on the roads over a given terrain. There will be packages for it to pick up at certain designated locations along the roads, and packages to deliver at others. Most of the robot's time will actually be spent moving on a road from one location to another, possibly turning at intersections, which are also designated locations. We assume that the robot itself does not put packages on its cart or take them off; at the appropriate locations, it signals for customers there to do so.

So in the simplest case, we are considering three endogenous actions that the LEGO robot can perform:

- `(leave-location!` *rob*`)`
  The robot starts moving along the road from it current location towards the next one. It continues following the road until an `arrive-at-location!` action happens.
- `(turn!` *rob dir*`)`
  The robot turns left, right, or around at its current location.
- `(req-customer-action!` *rob*`)`
  The robot announces its presence at its current location and waits for someone to take parcels off or put parcels on. It does not leave until the `customer-action-done!` action below happens.

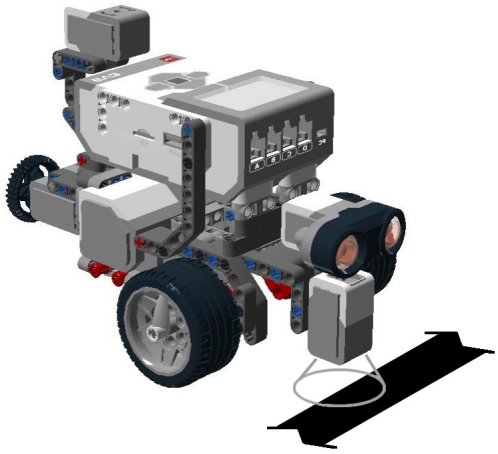
We are also considering three exogenous actions that can happen:

- `(arrive-at-location!` *rob*`)`
  This action happens when a moving robot arrives at the next location on the road.
- `(customer-action-done!` *rob*`)`
  This action happens after someone signals that the parcels to be picked up have been placed on the robot and the parcels to be delivered have been removed.
- `(request-delivery!` *user pkg from to*`)`
  This action happens when a customer requests a delivery, where the given package is to be taken from one location to another.

Note that we are *not* treating "getting from one location to another" as a single primitive action. This is an operation that can take several seconds to perform, and so, following the discussion in Section 6.3.2, the primitive action `leave-location!` is used to get the robot moving towards the next location, but then actually arriving is something that happens later, signaled by the exogenous `arrive-at-location!` action.

As to what this terrain should actually look like, perhaps the simplest idea is to imagine a robot assembled with a colour sensor aimed down like the one shown in Figure 8.8, and to construct a world where the robot will be able to distinguish the brightness of the normal terrain, from the brightness of the roads, from the brightness of the designated locations. One way is to use a dull non-reflective surface for the terrain, a black non-reflective tape for the roads, and a bright shiny tape on the roads for the designated locations. Some advance calibration with the robot will be needed so that the three ranges of light intensities to use for terrain, roads, and the designated locations can be settled in advance.

Figure 8.8: A LEGO Robot with a downward light sensor



8.3.1 A road map

While the robot needs to be able to go from one location to another, it is up to *ERGO* to keep track of the locations more globally. Roads can be curved and do not have to meet at right angles. What matters in *ERGO* is the connectivity of the roads, that is, which designated locations are connected to which others, without regard to whether the road is long or short, straight or curved.

We will use a table `adjacency` that maps each location x to four locations (u_0, u_1, u_2, u_3) as a sequence ordered clockwise. The idea is that if you arrive at location x from location u_0, then the next location you can get to will be as follows: u_1 if you turn left, u_2 if you go straight, u_3 if you turn right, and of course, u_0 if you turn around. This means that if you arrive at x from u_1, then the next location you get to will be u_2 if you turn left, u_3 if you go straight, and u_0 if you turn right. Arriving at x from u_2 and from u_3 are similar. To indicate that it is impossible to head in a certain direction from x, the corresponding location u_i can be `#f`. The connectivity is as follows: if none of the u_i are `#f`, the location is a normal four-way intersection; if one of the u_i is `#f`, the location is a T or Y intersection; if two or more of the u_i are `#f`, the location is not an intersection at all; if three of the u_i are `#f`, the location is a dead end; and if all four are `#f`, the location is an isolated point.

To keep track of where the robot is on the map, we need to know its location and relative orientation. For orientation, we will use a number $i = 0, 1, 2$, or 3, with reference to the adjacencies u_i described above. If the robot is at location x with orientation i, we take this to mean that location u_i is the one directly ahead of the robot. (Note that if $i \neq j$, then orientation i is different from j even when $u_i = u_j$.)

The Scheme code for the map functionality is shown in Figure 8.9. The functions `next-location` and `next-orientation` refer to the location and orientation the robot would have after moving straight ahead. The function `shift-orientation` refers to the orientation after turning left, right, or around. The function `get-direction` returns the turn direction it would take to go from one location to an adjacent one (or `#f`, if no turn

Figure 8.9: Program file `Projects/LEGO/delivery-map.scm`

```
;;;  This is the code that defines the map of roads for the LEGO robot

;; The table below is for a map that looks like this:
;;     1 --- 2 --------- 7 ---8
;;          /            |     \
;;          |            |      \
;;  3 ----- 4 --- 5 ---- 6 ----- 9
;;         /             |
;;       10             11
(define adjacency
  (hasheq 1 '(#f 2 #f #f)  2 '(#f 7 4 1)    3 '(#f 4 #f #f)  4 '(3 2 5 10)
          5 '(4 #f 6 #f)   6 '(5 7 9 11)    7 '(2 #f 8 6)    8 '(7 #f #f 9)
          9 '(6 8 #f #f)  10 '(#f 4 #f #f) 11 '(#f 6 #f #f)))

(define (adjs x) (hash-ref adjacency x #f))

;; the next location after leaving x with orientation ori
(define (next-location x ori) (list-ref (adjs x) ori))

;; the next orientation after leaving x with orientation ori
(define (next-orientation x ori)
  (let ((locs (adjs (next-location x ori))))
    (modulo (+ (for/or ((i 4)) (and (eq? x (list-ref locs i)) i)) 2) 4)))

;; the orientation that results from turning dir = left, right or around
(define (shift-orientation ori dir)
  (modulo (+ ori (case dir ((left) -1) ((right) 1) ((around) 2))) 4))

;; the dir that is needed to go from x and ori to an adjacent location y
(define (get-direction x ori y)
  (let ((locs (adjs x)))
    (case (for/or ((i 4)) (and (eq? y (list-ref locs (modulo (+ ori i) 4))) i))
      ((0) #f) ((1) 'right) ((2) 'around) ((3) 'left))))

;; a path of adj locations whose first element=start and last element=end
(define (find-path start end)
  (let loop ((seen '()) (nodes (list (list start))))
    (define (nexts x) (for/only ((y (adjs x))) (and y (not (memq y seen)))))
    (define (next-paths x path) (for/list ((y (nexts x))) (cons y path)))
    (if (null? nodes) #f
        (let ((path (car nodes)) (x (caar nodes)))
          (if (eq? x end) (reverse path)
              (loop (cons x seen) (append (cdr nodes) (next-paths x path))))))))
```

is necessary). Finally, the function `find-path` returns a shortest list of adjacent locations between a start and end location.

8.3.2 The delivery BAT

The BAT for the delivery robot is shown in Figure 8.10. It uses the fluents `location` and `orientation` as described above, the fluent `in-transit?` for when the robot is between locations, the fluent `loading?` for when the robot is waiting for a customer to add

Figure 8.10: Program file `Projects/LEGO/delivery-bat.scm`

```
;;;  This the BAT for a LEGO delivery robot

(include "delivery-map.scm")

(define-fluents
  location    1      ; where the robot was last located on the map
  orientation 1      ; 0,1,2,3 according to the last orientation of the robot
  in-transit? #f     ; is the robot between locations?
  loading?    #f     ; is the robot loading/unloading packages?
  onboard     '()    ; a list of (pkg from to) items that are with the robot
  pending     '()    ; a list of (pkg from to) items that are awaiting pickup
  done        '())   ; a list of (pkg from to) items that have been delivered

;; the endogenous actions
(define-action (leave-location! r)
  #:prereq    (and (not in-transit?) (next-location location orientation))
  in-transit? #t)

(define-action (turn! r dir)
  #:prereq    (not in-transit?)
  orientation (shift-orientation orientation dir))

(define-action (req-customer-action! r)
  #:prereq    (not in-transit?)
  loading?    #t)

;; the exogenous actions
(define-action (request-delivery! u obj from to)
  pending     (append pending (list (list obj from to))))

(define-action (arrive-at-location! r)
  location    (next-location location orientation)
  orientation (next-orientation location orientation)
  in-transit? #f)

(define (pkgs-on) (for/only ((x pending)) (= (cadr x) location)))
(define (pkgs-off) (for/only ((x onboard)) (= (caddr x) location)))

(define-action (customer-action-done! r)
  loading?    #f
  done        (append (pkgs-off) done)
  onboard     (append (remove* (pkgs-off) onboard) (pkgs-on))
  pending     (remove* (pkgs-on) pending))
```

or remove packages, and the fluents `pending`, `onboard`, and `done` as a list of items that are either to be delivered, are being delivered, or have been delivered respectively. Note that `leave-location!` changes `in-transit?` only; `location` and `orientation` are changed only after the robot arrives at its next location via the exogenous `arrive-at-location!`. (The actual movement of the robot will be initiated and terminated by the robot manager.) However, the prerequisite of the `leave-location!` action tests the next location to ensure that there is a location to go to. As for the packages, note that items are put on at the ends of the onboard and pending lists, making those lists behave like queues.

Figure 8.11: Program file `Projects/LEGO/delivery-main.scm`

```
;;; This the main program for a LEGO delivery robot
;;; To start, first get EV3 server running at IP address <addr>
;;; Then do:  racket -l ergo -f <this file>  -m <addr>

(include "tag-bridge.scm")
(include "delivery-bat.scm")

(define home-location location)
(define robo 'my-EV3)

(define (at-package-loc?) (not (and (null? (pkgs-on)) (null? (pkgs-off)))))

(define (load/unload)
  (:begin (:act (req-customer-action! robo)) (:while loading? (:wait))))

(define (work-to-do?) (not (and (null? onboard) (null? pending))))

(define (next-work-loc)
  (let ((loc1 (if (null? onboard) #f (caddr (car onboard))))
        (loc2 (if (null? pending) #f (cadr (car pending)))))
    (if (not loc1) loc2
        (if (not loc2) loc1
            (let ((p1 (find-path location loc1)) (p2 (find-path location loc2)))
              (if (> (length p1) (length p2)) loc1 loc2))))))

(define (head-for loc goal)
  (:let ((dir (get-direction loc orientation (cadr (find-path loc goal)))))
    (:when dir (:act (turn! robo dir)))
    (:act (leave-location! robo))
    (:while in-transit? (:wait))))

(define (delivery)
  (:monitor
    (:when (at-package-loc?) (load/unload))
    (:while (work-to-do?) (head-for location (next-work-loc)))
    (:until (eq? location home-location) (head-for location home-location))
    (:while #t (:wait))))

(define (main . args)
  (and (null? args) (error "Must supply an IP address for EV3"))
  (tag-stdio-setup)                          ; for request-delivery! action
  (tag-tcp-setup robo 8123 (car args))       ; port 8123 assumed
  (ergo-do #:mode 'online (delivery)))
```

8.3.3 The delivery program

The main delivery program is shown in Figure 8.11. All the work happens in the `delivery` procedure which is a monitor with programs at four priorities. At the lowest level, the monitor does nothing, waiting for something to happen, presumably a `request-delivery!` action from the user. At a slightly higher level, the robot has a homing behaviour, and takes one step towards its home base. At the next higher level, the robot checks to see if it has delivery work to do, and if so, makes a decision about what location to head to, and takes one step in that direction. At the highest level, the robot checks if there are packages to

load or unload at the current location, and if so, requests customer action, and then waits for a signal to proceed. Note that the robot is opportunistic: if on the way to some location, it happens to find itself somewhere where there are packages to be picked up or dropped off, it will stop there before continuing.

The `head-for` procedure is what is used for taking one step towards a goal location. It finds the shortest path towards the goal, does a turn as necessary to face the first destination in the path, and then heads out towards that location, waiting until it arrives. The way the program is structured, the robot only does one step at a time before reconsidering where it is heading. This allows it to react to changing circumstances, possibly reconsidering the rest of the projected moves.

The `next-work-loc` procedure decides where to go to deliver or pickup packages. If either nothing is onboard or nothing is pending, the case is clear. But if there are packages on board and packages waiting to be picked up, the current routine looks at the destination of the first onboard package and the source of the first pending package and goes to the location that is the furthest away. (Other scheduling options are certainly possible.)

8.3.4 A robot manager in EV3 Python

Turning now to the EV3 side, the job of the robot manager on the EV3 is to perform the endogenous actions `leave-location!`, `turn!`, and `req-customer-action!` on request, and to signal the two exogenous actions `arrive-at-location!` and `customer-action-done!` when they occur. We will not try to describe here how a LEGO robot that can do this should be assembled, connected to motors and sensors, or programmed in Python, but will only offer some general suggestions.

We expect the robot manager to have the structure of the program shown in Figure 8.12. Before going any further, it is probably worth trying out this version of the entire system. (No motors or sensors are required.) The `delivery-manager0.py` program should be run on an EV3, and then the *ERGO* program `delivery-main.scm` on another computer. At this point, the TCP link should be established, but nothing should happen until an action is typed at the `Act:` prompt from *ERGO*, something like `(request-delivery! pkg1 4 6)`. Then the EV3 should start printing out messages simulating a computer moving, turning, and eventually arriving at location 4 on the map. It will then prompt for input (simulating a customer action), head for location 6, require another input, and finally head home.

Assuming the overall system is working as above, what is needed to complete the project is a new version of `delivery-manager0.py`, but with functions that control the motors and read from the sensors, rather than using `print` and `input`.

Observe that an action like `leave-location!` should return immediately so that other things can happen while the robot is moving towards its destination. The easiest way to do this is to start a thread as shown in Figure 8.12. For `leave-location!`, the thread should keep the robot moving until it reaches the next location, at which point the thread should stop and signal its arrival with the exogenous action `arrive-at-location!`. The function for the `req-customer-action!` action is similar except that the operation ends with the exogenous action `customer-action-done!`. (The `turn!` action, on the other hand, may be quick enough that a thread is not needed to handle its completion.)

To handle a `turn!` action, the robot can do a turn of 90° or −90° or 180° by dead reckoning, and then adjust its position as necessary so that it remains at the same designated

Figure 8.12: Program file Servers/EV3/delivery_manager0.py

```
import sys, time, threading #, ev3dev.ev3 as ev3 (uncomment when needed)
from ergo_communication import signal_exogenous, ergo_tcp_session

def do_leave_location():
    print('*** heading straight')
    threading.Thread(target=going_to_location, daemon=True).start()

def going_to_location():
    for i in range(4):
        print('      moving forward ...')
        time.sleep(2.0)
    signal_exogenous('arrive-at-location!',[])

def do_turn(dir):
    if dir=='left': print('*** turning left')
    elif dir=='right': print('*** turning right')
    else: print('*** turning around')
    time.sleep(3.0)

def do_req_customer_action():
    print('*** request customer_action')
    threading.Thread(target=getting_customer_action, daemon=True).start()

def getting_customer_action():
    input('Type ENTER when customer_action is complete: ')
    signal_exogenous('customer-action-done!',[])

#########################################################################
###   The two procedures needed for ergo_tcp_session

def initialize():
    print('Starting Delivery Manager')

def do_endogenous(actName,args):
    if actName=='leave-location!': do_leave_location()
    elif actName=='turn!': do_turn(args[0])
    elif actName=='req-customer-action!': do_req_customer_action()
    else: print('No action')

ergo_tcp_session(8123,initialize,do_endogenous)
```

location, ready to move forward. This may require some experimentation and tuning depending how well the robot can rotate on its own axis.

There are different ways of handling the `req-customer-action!` action. First the robot should announce its presence. It can beep, say something using the onboard speech system, flash its LEDs, or even use a small motor to raise a flag. At this point, the thread should begin. Maybe the simplest thing to use is a touch sensor aimed upwards. The thread can repeatedly read the value of that sensor until the touch sensor has been pressed by the customer, in which case the thread would signal completion and terminate.

The `leave-location!` action should start the robot moving forward and then monitor its progress in a loop within a thread. The idea is to check the light sensor aimed downwards repeatedly and decide what to do based on whether the sensing value indicates the

presence of a road, a designated location, or just ordinary terrain. When the robot is on a road, it should continue moving; when the robot is at a designated location, it should stop moving and signal arrival and terminate the thread; when the robot is on ordinary terrain, it should adjust its motion, to the left or to the right, to get back onto the road. How much to turn and for how long to get a smooth forward motion along a road (especially a curving one) is something that will definitely require experimentation and tuning. This might be the most difficult robot behaviour to get right. (There are some online examples of programs that get a LEGO robot to follow a line in this way.)

8.4 Extensions and variants

The LEGO robot as presented can be refined in a number of ways. For one thing, very little error checking is done and there is no option to print information for debugging. It might be useful to have a new action that allows users to request a status report on the delivery of a package: has it been picked up? has it been delivered? where is it currently? Related to this, it would be useful for a user to be able to cancel a scheduled pickup.

Another limitation of the program concerns how jobs are scheduled by `next-job-loc`. By using a time stamp, it should be possible to do better at ensuring packages do not wait too long before being processed. (As is, requests to pick up packages could "starve" the delivery of the packages on board.) Similarly, instead of determining distance on the map in terms of the number of locations to cross, the map could contain distance information, with a change needed to the `find-path` function to get the shortest path.

The idea of `head-for`, that is, getting to a location by going to an adjacent location and then reconsidering, makes it easy to allow other things to happen. For example, it might be useful to allow users to request the delivery of "priority" packages that could interrupt normal deliveries or pickups taking place.

A slightly more complex issue involves the use of time. Calculations on the map are done just before the `leave-location!` action, which is fine when the map is small. However, it is after the `leave-location!` action, when the robot is in transit, that *ERGO* will have time to spare. One possibility is to make the `head-for` procedure use a fluent `next-step` whose value is the location the robot should next be heading for. This fluent can then be reset by the `leave-location!` action or by some other concurrent program.

A more global issue concerns what to do with a robot that does not arrive at its intended destination in a reasonable amount of time. As suggested in Section 6.3.2, the robot can signal the end of the transit using an action other than `arrive-at-location!`. What to do next is the hard part. A simple solution is to ensure that the world has some hard-to-miss landmarks that the robot can use to reorient itself. For example, there may be an outside boundary to the world that the robot can detect and follow (perhaps using its touch sensors) until it arrives at some feature with a known location on the map.

An even more interesting prospect, however, is to have multiple LEGO robots on a large surface controlled by one *ERGO* program. The primitive actions are tagged and so are ready to deal with multiple robots. Some of the fluents will have to be tables of values (indexed by tags) rather than single values. But the hard part will be deciding how to coordinate the robots to use them effectively. Two robots should never be heading to pickup the same package, for example. It will also be necessary to deal with robots trying to get past each other on the roads and avoid collisions.

Chapter 9

A Real-Time Video Game

Real-time video games offer one of the more compelling examples of computer-generated artificial worlds. In these games, a user is shown a graphically-rendered environment, and then, using a keyboard or other hardware controller, the user must control one or more agents living in that environment to achieve some goal. In some cases, the goal might be no more than to explore the environment; in other cases, the goal might be to build a city or to stay alive. It is typical of these games that other agents also inhabit the same environment. These might be other users connected to the game somehow (for example, over a network) or computer-generated agents (often called *non-player characters* or NPCs).

In this chapter, we consider an *ERGO* project that involves the construction of a very simple game called CarChase using the freely-available Unity 3d game development system. In this game, there are two primitive cars called Patrol and JoyRide that drive around an equally primitive terrain. The Patrol car is under user control, while the JoyRide car will be an NPC controlled externally by *ERGO*. The object of the game is for the Patrol car to catch the JoyRide car and tap it on the rear bumper. A snapshot of a game in progress is shown in Figure 9.1. This shows the Patrol car from behind with its headlights on approaching the JoyRide car ahead and to its right. (The other triangular-shaped features on the terrain are supposed to be rocks.)

9.1 Unity 3D

Unity 3d is a system for constructing real-time video games. The games can then be played on computers, over the web, or on smart phones. What the games actually look like (or sound like) is completely up to the game designer: Unity 3d provides basic shapes and textures, but it is up to the programmer to assemble them into reasonable-looking worlds. When a game is being played, the world is displayed (according to what the main camera has been told to look at), one video frame at a time. What the objects in the game actually do from frame to frame is again up to the programmer: Unity 3d provides basic physics for inactive physical objects, but the programmer must provide scripts in the C# programming language for objects that are intended to behave on their own.

A full account of Unity 3d and C# are beyond the scope of this book, but there is considerable material online. See `https://unity3d.com/` for information about Unity 3d, including how to install it on various platforms, and an extensive manual and help system.

Figure 9.1: A CarChase game in progress

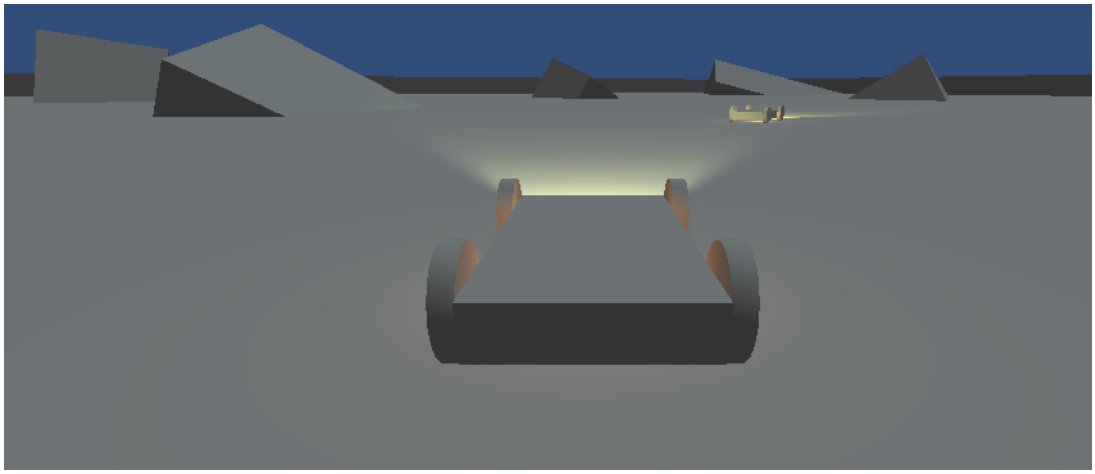


Figure 9.2: Program file `Servers/Unity3D/CarChase/Assets/PatrolCar.cs`

```
using UnityEngine;

// This is the script to be attached to the car "Patrol" in the scene.
// It reads arrow keys from the terminal and moves the car accordingly.

public class PatrolCar : MonoBehaviour {
    private float forwardSpeed = 80f;
    private float turnSpeed = 0.4f;
    private Rigidbody myCarBody;
    private Transform myCar;

    void Start () {
         myCarBody = GetComponent<Rigidbody>();
         myCar = GetComponent<Transform>();
    }

    void Update () {
         float forwardAmount = Input.GetAxis("Vertical")*forwardSpeed;
         float turnAmount = Input.GetAxis("Horizontal")*turnSpeed;
         myCar.Rotate(0,turnAmount,0);
         myCarBody.AddRelativeForce(0,0,forwardAmount);
    }
}
```

For C# itself, see `https://docs.microsoft.com/en-us/dotnet/csharp/` for the language reference and tutorials.

The main idea in Unity 3d scripting can be seen in a small example. Figure 9.2 shows the full script for the Patrol car in the CarChase game. There are two fixed parameters, `forwardSpeed` and `turnSpeed`, which are used to control how quickly the car will go forward or turn. After the variable declarations, two special methods are defined: `Start`,

called by Unity 3d at the start of a game; and `Update`, called by Unity 3d before a video frame is rendered during a game. In this case, the `Update` method first determines which arrow keys are currently being pressed by the user. For the vertical, `Input.GetAxis` returns 1 for up-arrow, -1 for down-arrow, and 0, otherwise; for the horizontal, it returns 1 for left-arrow, -1 for right-arrow, and 0, otherwise. The `Update` method then calls the car's `Rotate` and `AddRelativeForce` methods to change the orientation and position of the Patrol car in the current frame. (The car has been constructed as a rigid body subject to physics, and thus force is required to move it forward.) So for example, if no arrow keys are pressed, the `turnAmount` and `forwardAmount` will both be set to zero, and the car will not move in current frame, unless it is coasting as a result of its physics.

In addition to the `Start` and `Update` methods, there are many other special methods that are called automatically by Unity 3d. The ones used here in the CarChase project for the other car (the JoyRide car controlled by *ERGO*) are the following:

- `OnApplicationQuit`, called for an object at the end of the game only;
- `OnTriggerEnter`, called for an object declared to be a trigger when another object first touches it;
- `OnTriggerExit`, called for an object declared to be a trigger when another object stops touching it;

This only scratches the surface of the many behaviours that can be controlled by Unity 3d, which includes GUI events and facilities for multi-player games.

9.2 Interacting with *ERGO*

To interact properly with *ERGO*, a robot manager for Unity 3d needs to be able to perform the endogenous actions generated by the *ERGO* program as well as to signal the *ERGO* program when exogenous events occur. In our case, communication between *ERGO* and Unity 3d will take place over TCP managed by two programs, `ErgoServer.cs` on the Unity 3d side, and `u3d-bridge.scm` on the *ERGO* side.

9.2.1 The interaction programs

The `ErgoServer.cs` file defines a class `ErgoServer` with the following attributes:

- `Start`: starts a TCP server for communication with *ERGO*;
- `Stop`: terminates the TCP server communication with *ERGO*;
- `Update`: obtains an endogenous action from *ERGO*, when one is available;
- `hasData`: true when an endogenous action was received by `Update`;
- `getEndogenous`: returns the endogenous action if `hasData` was true
- `signalExogenous`: sends an exogenous action to *ERGO*;

Figure 9.3: The form for a Unity 3d robot manager

```
using UnityEngine;

// This is a script to be attached to an ERGO-controlled object called myRobot

public class myRobot : MonoBehaviour {
  private ErgoServer server = new ErgoServer();
  // any other declarations

  void Start () {
    server.Start();
  }
  void OnApplicationQuit() {
    server.Stop();
  }
  void Update () {
    server.Update();
    if (server.hasData) {
      string act = server.getEndogenous();
      // what to do with the endogenous action
    }
  }
  // any other methods
}
```

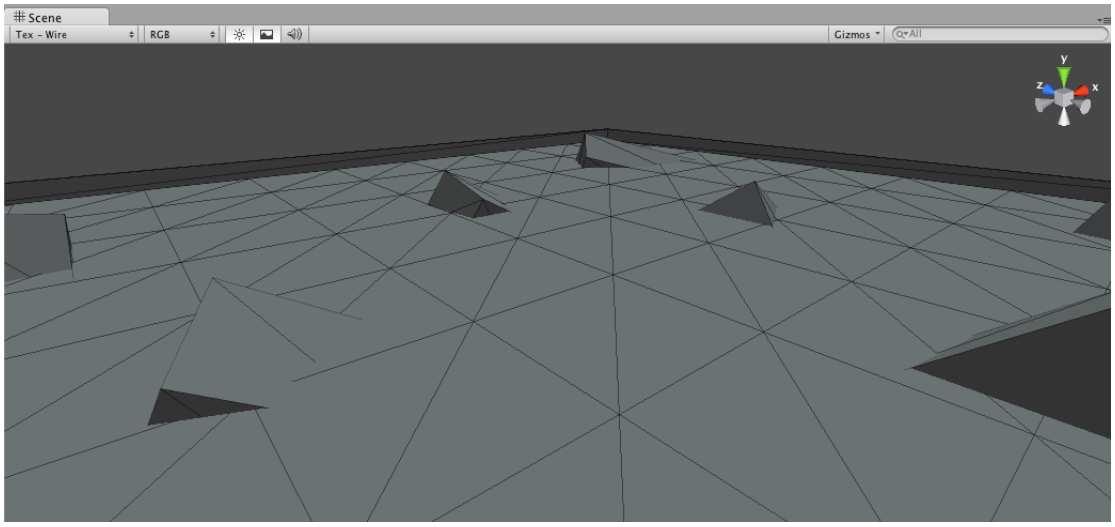
These are all methods to be called except for `hasData` which is a read-only variable. None of the methods take arguments except for `signalExogenous`, which takes a string argument. The methods return no values except for `getEndogenous`, which returns a string.

On the *ERGO* side, the file `u3d-bridge.scm` defines `(u3d-start-comm` *port trace*?`)`, which makes a connection to a Unity 3d server over TCP, and defines the interfaces required by online *ERGO* programs (See Section 6.2.) If the *trace*? argument is true, the actions sent and received are also displayed on the terminal.

9.2.2 Programming a robot manager

No matter how the Unity 3d robot to be controlled by *ERGO* behaves, its C# script will have a form like the one shown in Figure 9.3. The script creates an instance of the `ErgoServer` class and then calls the `Start`, `Stop`, and `Update` methods of this class at the beginning of its own corresponding methods. Within its `Update` method, after it calls the server `Update` method, if the `hasData` variable has been set to true, then this indicates that Unity 3d has received an endogenous action from *ERGO*. So the robot manager should obtain this action from the server using the `getEndogenous` method and perform whatever robot behaviour is appropriate for that action. (The `Update` method must complete very quickly as it is called every video frame. Typically the action involves nothing more than setting some global variables for later use by the robot.) Regarding exogenous actions, the `Update` method may look for certain conditions and signal an exogenous action, or this may happen within other game objects affiliated with the robot, like in the script for the LookAhead component at the front of the JoyRide car, seen in Figure 9.8 below.

Figure 9.4: The terrain with no cars



9.3 The CarChase project

This section covers all the steps needed to build a working CarChase game from scratch. Almost all the effort in this example is on the Unity 3d side; the *ERGO* program in this project is extremely simple. It is assumed in what follows that the reader already knows the Unity 3d editor interface and how to write basic C# programs.

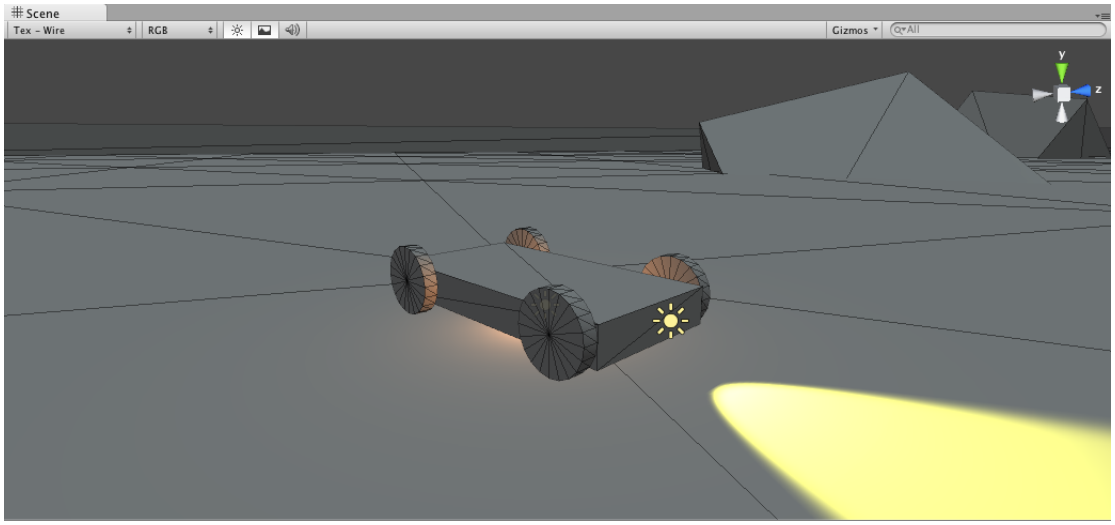
9.3.1 The terrain and Patrol car

The first thing to do in a new Unity 3d project is to build some terrain. The CarChase terrain consists of a `Plane` of scale `(50,1,50)`; four small boundary edges, each of which is a `Cube` of scale `(500,10,1)` positioned at an edge of the plane; and a number of rocky features, each of which is a `Cube` rotated and embedded within the plane. An overhead `Directional Light` is added at position `(0,100,0)` and rotation `(70,280,280)`. Its intensity can be set to `.5` and colour to `(60,67,69)`. The light should be renamed `OverheadLight` for later use. The scene that results should look something like what is seen in Figure 9.4.

Next the cars are built. The main body of a car is a flattened `Cube` of scale `(4,1,8)`. Four wheels are added. Each wheel is a flattened `Cylinder` of scale `(2,.05,.25)` positioned beside the main body. For a headlight, a `Spotlight` is used, of rotation `(14,0,0)` and scale `(.8,1.3,.17)` and positioned at the front of the car. Its range is set to `120`, angle to `63`, colour to `(219,218,129)` and intensity to `2`. A `Point Light` is then positioned just under the body of the car to light it from below. It has range `12`, colour `(242,135,65)`, and intensity `.8`. At this point, the car should look something like the one in Figure 9.5.

The last thing to do with the car is to add a `Rigidbody` component to give it some physical properties. The mass is set to `1`, the drag to `.75`, the angular drag to `.5` and the `Use Gravity` box is checked. Under `Constraints`, the `Freeze Rotation` boxes for `X` and `Z`

Figure 9.5: A closeup of a car



are checked, so that the car will only turn on its Y-axis. This completes a basic car. Rename the car `Patrol`, and make a copy of it called `JoyRide` for later use below.

To complete the Patrol car, move the Main Camera to within the Patrol object. This causes the camera to follow the Patrol car, providing a first-person view of the scene during the game. Position the camera so that it is just behind the Patrol car looking down, as in Figure 9.1. Finally, the Patrol car needs a controlling script: attach the one from Figure 9.2.

The Patrol car is now ready for a test drive. Confirm that the system works properly by starting the game, pressing the arrow keys and seeing the Patrol car move appropriately. (Note that vertical and horizontal keys can be pressed simultaneously.) It should be possible to drive around the entire terrain, avoiding rocks, using the arrow keys.

9.3.2 The JoyRide car

Assuming a Patrol car is working properly under user control, we can now turn to the JoyRide car to be controlled by *ERGO*. This car will be like the Patrol car, but it will not get the camera and will have its own script, shown in Figure 9.6. For this car, the amount to rotate and the force to apply depend on the variables `turnDir` and `forwardDir` which are set according to endogenous actions received from *ERGO*. This script should be attached to the JoyRide car. It uses the `ErgoServer` class, which means that the `ErgoServer.cs` script must also be included as part of the project by copying that file to the `Project` window.

To ensure that the TCP communication is now working properly, start the game and enter the following in another window:

```
> telnet localhost 8123
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Figure 9.6: Program file Servers/Unity3D/CarChase/Assets/JoyRideCar.cs

```
using UnityEngine;

// This is the script to be attached to the "JoyRide" car in the scene.
// It accepts actions via an ErgoServer and moves the car accordingly.

public class JoyRideCar : MonoBehaviour {

    private float forwardSpeed = 35f;
    private float turnSpeed = 0.6f;
    private Rigidbody myCarBody;
    private Transform myCar;

    private int turnDir = 0;
    private int forwardDir = 0;
    public ErgoServer server = new ErgoServer();

    void Start () {
         myCarBody = GetComponent<Rigidbody>();
         myCar = GetComponent<Transform>();
         server.Start();
    }

    void Update () {
         server.Update();
         if (server.hasData) {
            string act = server.getEndogenousAct();
            switch(act) {
              case "right-turn!": turnDir = 1; break;
              case "left-turn!":  turnDir = -1; break;
              case "straight!":   turnDir = 0; break;
              case "stop!":       forwardDir = 0; break;
              case "go!":         forwardDir = 1; break;
              case "reverse!":    forwardDir = -1; break;
            }
         }
         myCar.Rotate(0,turnDir*turnSpeed,0);
         myCarBody.AddRelativeForce(0,0,forwardDir*forwardSpeed);
    }
}
```

Nothing happens in Unity 3d unless it has the focus, so bring it to the front. It should now display a message saying that a connection from *ERGO* has been received. Returning to the telnet window, typing a `go!` action should cause the JoyRide car to move when Unity 3d regains the focus. If all is well, the JoyRide car will proceed forward and eventually crash.

The Joyride car needs two more components to give it additional behaviour. First, create a rear bumper as a `Cube` of scale `(1,1,.16)` positioned at the rear of the car. The `MeshRenderer` component of this cube should be removed so that the cube will be invisible. The `IsTrigger` box of its `BoxCollider` should be checked so that it will be triggered by collisions. Then the script shown in Figure 9.7 should be attached. During the game, if anything touches the rear bumper (namely, the Patrol car), the `OnTriggerEnter` method

Figure 9.7: Program file Servers/Unity3D/CarChase/Assets/BumperScript.cs

```
using UnityEngine;

// This script is attached to the RearBumper component of the JoyRide car.
// It detects contact with the Patrol car and ends the game.

public class BumperScript : MonoBehaviour {
    Light overhead;

    void Start() {
        overhead = GameObject.Find("OverheadLight").GetComponent<Light>();
    }
    void OnTriggerEnter(Collider col) {
        overhead.intensity = 0.0f;
        Application.Quit();
    }
}
```

Figure 9.8: Program file Servers/Unity3D/CarChase/Assets/LookAheadScript.cs

```
using UnityEngine;

// This script is attached to the LookAhead component of the JoyRide car.
// It detects collisions and signals them with an exogenous action.

public class LookAheadScript : MonoBehaviour {
    ErgoServer server;
    Transform myTrans;

    void Start() {
        myTrans = transform;
        server = transform.parent.GetComponent<JoyRideCar>().server;
    }
    void OnTriggerEnter(Collider col) {
        Vector3 myPos = myTrans.position;
        Vector3 itsPos = col.ClosestPointOnBounds(myPos);
        float dist = Vector3.Distance(myPos,itsPos);
        server.signalExogenous("(object-detect! "+dist+")");
    }
    void OnTriggerExit(Collider col) {
        server.signalExogenous("(object-detect! 0)");
    }
}
```

will be called to turn off the overhead light and quit the game.

Finally, the JoyRide car needs a way of seeing objects ahead of it. One way of doing this is to create a long Cube of scale (15,1,2.7) and placing it at the front of the car. Like the rear bumper, the cube should be made invisible and set so that it will be triggered by collisions. The idea is that an object that is directly in front of the car will touch this cube, and this event can be reported to *ERGO* so that it can decide what to do. In effect, the invisible cube behaves like a forward-looking sensor for the car.

Figure 9.9: Program file `Projects/Unity3D/u3d-car.scm`

```
;;;  This is the ERGO code for a robot car doing a JoyRide
;;;  It communicates with a running Unity 3d engine.

(include "u3d-bridge.scm")                    ; bridge to the Unity 3d Engine

(define-fluents
  distance    0                               ; how far to an object (0 = clear)
  turning?    #f                              ; am I turning?
  advancing?  #f)                             ; am I going forward?

;; four normal actions
(define-action right-turn! turning? #t)
(define-action straight! turning? #f)
(define-action go! advancing? #t)
(define-action stop! advancing? #f)

;; one exogenous action, reporting on objects ahead
(define-action (object-detect! d) distance d)

;; the car controller
(define (control)
  (:monitor
     (:when (and (> distance 0) (< distance 20) advancing?)
         (:act stop!) (:unless turning? (:act right-turn!)))
     (:when (and (> distance 0) (< distance 60) (not turning?))
         (:act right-turn!))
     (:when (= distance 0)
         (:when (not advancing?) (:act go!))
         (:when turning? (:act straight!)))
     (:while #t (:wait))))

(define (main . args)
   (u3d-start-comm 8123 (not (null? args)))  ; set up online interfaces
   (ergo-do #:mode 'online (control)))       ; run the ERGO program
```

The script to be attached to the cube for this purpose appears in Figure 9.8. What it does is to first locate the `ErgoServer` instance used by the JoyRide car itself. Then, if anything touches the cube, the distance between the cube and the colliding object is calculated and sent to *ERGO* using `signalExogenous`. When the object stops touching the cube, *ERGO* is sent another exogenous event with a distance value of zero.

This completes the Unity 3d part of the CarChase project, which now can be saved as an application, allowing the game to run without the Unity 3d development environment.

9.3.3 The *ERGO* program

The *ERGO* program for the JoyRide car is shown in Figure 9.9. The program has three fluents: the fluents `turning?` and `advancing?` are set and reset by the four endogenous actions; the `distance` fluent is set exogenously by the `object-detect!` action.

What the manager does is as follows: under normal conditions it sends the car forward

without turning, waiting for something to happen; if it detects an object ahead in the distance, it continues forward but initiates a turn to the right; if the object ahead gets too close, it stops the forward force and initiates or continues the turn to the right; when it no longer detects an object ahead, it resumes driving straight normally.

9.3.4 Playing the game

To play the CarChase game, restart the game, and then start the JoyRide car with:

```
> racket -l ergo -f u3d-car.scm -m [ trace ]
```

Nothing will happen until the game regains the focus, but at that point, the JoyRide car should start moving under *ERGO* control. The Patrol car can then be put in pursuit using the arrow keys, and the chase is on.

9.4 Extensions and variants

The CarChase is a fairly primitive game and a number of extensions are possible.

On a superficial level, the appearance of the game needs work. The terrain, including the "rocks," could use texture and refinement. The basic surface need not be a flat plane and the rocks need not have smooth surfaces or right angles. There can be other objects as obstacles on the terrain, including some moving objects. The cars could also be more than basic geometric shapes and could have interesting textures and features of their own. The Patrol car could have a rotating red light on the roof, for example. The headlights on a car could turn on and off as the car starts and stops. Unity 3d also allows particles to be emitted from an object, so that the cars could leave a trail of smoke behind them (making it easier for the Patrol car to spot the JoyRide car). There could also be objects like a hay stack that a car could drive through, and even objects like a stack of tires that could stop a car but in a more physically complex way.

At the next level, both cars could have a more sophisticated turning mechanism under the control of physics. For example, the JoyRide car would be able to go much faster if it applied negative force (rather than zero force) on sharp turns.

Dividing the terrain into roads and off-roads is somewhat more problematic. The JoyRide car would certainly need a more complex script to keep it on a road. Perhaps the simplest way to handle roads is to reinterpret the `go!` endogenous action to mean "continue along the current road" and the `right-turn!` action to mean "take the next turn to the right." Then the C# script for a forward-moving car would need to take note as the car drifts to one side or another of the road and compensate appropriately in the orientation and speed (especially for a sharp turn). The C# script for a turning car would need to take note as a branch in the road appears on the right and again adjust the speed and the orientation appropriately. A more complex alternative is to leave the `go!` and `right-turn!` actions as is (perhaps adding a `left-turn!` action), but arrange to send sensing information to *ERGO* about the position of the car on the road. In this case, the micro-adjustments required to drive the car in its lane on the road would be generated by *ERGO*.

Additional sensors and controls on the cars are other possibilities. Both the Patrol car and the Joyride car might allow variable speed and turn controls, rather than the current

all or nothing. For the Patrol car this would mean using other keyboard keys as controls; for the Joyride car this would require new endogenous actions. The simplest way perhaps is to have two endogenous actions: one that sets a forward force factor to a given amount (from -1 to 1, say) and one that sets a turning factor. A more complex solution would involve separate gas and brake controls and a more sophisticated physics for coasting.

The JoyRide car could also make use of the ability to sense a nearby Patrol car. This might be programmed in the form of a "radar detector" that sends an exogenous event whenever the Patrol car is close (as part of the car `Update` method), or more simply, a rear-looking detector that reports only when the Patrol car is not too far behind.

From a cognitive robotics point of view, perhaps the biggest limitation of the CarChase project is the purely reactive nature of the *ERGO* control program. It could have been written just as easily in C#. In the end, if all a cognitive robot needs to do is to move forward without bumping into things, there is not much for it to think about! Where the *ERGO* language pays off is when the robot has some purpose other than just driving, and needs to deliberate about what to do. (In other chapters, we have seen cases where a robot needs to do considerable offline planning before it can decide what to do next.) One way to consider making the CarChase world richer in this sense is for there to be locations in the world that the JoyRide car needs to visit in some order to achieve some goal.

It is worth remembering, however, that any agent in Unity 3d that can be controlled using arrow keys can also be controlled by endogenous actions in a similar way. The main limitation is that whereas a user at a keyboard gets to see the entire screen (and perhaps hear things too), the cognitive robot only gets sensing reports via exogenous actions. As a first step towards levelling the playing field, it will be necessary to ensure that an agent controlled by *ERGO* gets as much information as it needs from its sensors.

Advanced Topics

* Chapter 10

The Logical Story

This book has emphasized cognitive robots from the programming perspective. The idea of cognitive robotics, however, first arose as a problem not in programming, but in *knowledge representation and reasoning*. The idea is to imagine a system that can reason in an automated way from a collection of facts called its *knowledge base*.

In the case of cognitive robotics, the facts in the knowledge base concern the state of the world as well as its dynamics, that is, how the state of the world changes as the result of actions. In this chapter we reconsider the planning and program execution of a cognitive robot from the perspective of a reasoning agent that uses the language of first-order logic to represent what it knows, and that reasons using logical inference.

The main reason for moving from programming to logical reasoning is to consider an agent capable of making effective use of a wider range of knowledge than the BATs seen so far. For example, an agent might have to deal with incomplete knowledge about the world it is dealing with, where the values of fluents are not known. (We will see more of this in Chapters 11 and 12.) More generally, an agent may need to deal with knowledge that does not fit the current BAT pattern at all. For example, in deciding what to do, an agent may need to reason about the passage of time, or about the properties of rigid physical objects, or about how people typically interact. All of these suggest a knowledge base drawn from a declarative language that is more expressive than the *ERGO* language of BATs.

10.1 First-order logic

This is a very brief review of the syntax, semantics, and pragmatics of the language of first-order logic. (Standard textbooks should be consulted for further details.)

10.1.1 Syntax

Expressions in the language of first-order logic are made up of logical and non-logical symbols. The logical symbols are the punctuation marks (parentheses, comma, and period), the logical connectives ($\wedge$, $\neg$, $\forall$, and $=$ are sufficient, but others are typically introduced as abbreviations), and an infinite supply of individual variables. The non-logical symbols vary from application to application and are made up of predicate symbols and function symbols. Each non-logical symbol has an "arity" which is a non-negative integer specify-

ing how many arguments the predicate or function takes. Function symbols of 0-arity are called constants.

The expressions in the language come in two forms: terms and well-formed formulas (or wffs, for short). The terms are defined as the least set satisfying the following:

1. Every variable is a term.
2. If f is a function symbol of arity k and $t_1, \ldots t_k$ are terms, then $f(t_1, \ldots, t_k)$ is a term.

For constants, c is usually written instead of $c()$. The wffs are defined inductively as follows:

1. If t and t' are terms, then $(t = t')$ is a wff.
2. If P is a predicate symbol of arity k and $t_1, \ldots t_k$ are terms, then $P(t_1, \ldots, t_k)$ is a wff.
3. If α and β are wffs and x is a variable, then $\neg\alpha$, $(\alpha \wedge \beta)$ and $\forall x.\alpha$ are wffs.

Usually certain liberties are taken with the syntax: parentheses are added or omitted for clarity; square brackets and curly braces are used instead of parentheses, and abbreviations are used: $(\alpha \vee \beta)$ for $\neg(\neg\alpha \wedge \neg\beta)$, $(\alpha \supset \beta)$ for $(\neg\alpha \vee \beta)$, $(\alpha \equiv \beta)$ for $((\alpha \supset \beta) \wedge (\beta \supset \alpha))$, and $\exists x.\alpha$ for $\neg\forall x.\neg\alpha$.

Variables within a wff are considered to have a scope determined by the quantifier $\forall$. An appearance of a variable x within a formula is said to be bound if it appears within a subformula $\forall x.\alpha$; otherwise the appearance is said to be free. The notation α_t^x is used to name the formula that results from replacing every free occurrence of the variable x in formula α by the term t. A wff without free variables is called a sentence.

10.1.2 Semantics

Terms and formulas of first-order logic are interpreted according to an interpretation $\mathcal{M}$ made up of two parts: a non-empty set D called the domain of interpretation, and a mapping from the predicate and function symbols of the language to relations and functions over D. More precisely, for each predicate symbol P of arity k, the interpretation of P according to $\mathcal{M}$, written $P^{\mathcal{M}}$ is a k-ary relation over D:

$$P^{\mathcal{M}} \subseteq [D \times D ... \times D] \quad (k \text{ times}).$$

Similarly, for each function symbol f of arity k, the interpretation of f according to $\mathcal{M}$, written $f^{\mathcal{M}}$ is a total k-ary function from D to D:

$$f^{\mathcal{M}} \in [D \times D ... \times D \to D] \quad (k \text{ times}).$$

The idea is that an interpretation will determine which sentences of the language are true and which are false. To do so, it specifies which formulas are satisfied for which values of its free variables. Let μ be a function from the variables to D. Then, the denotation of a term t with respect to $\mathcal{M}$ and μ, written $|t|_{\mathcal{M},\mu}$, is defined inductively by:

1. $|x|_{\mathcal{M},\mu} = \mu(x)$;

2. $|f(t_1,\ldots,t_k)|_{\mathcal{M},\mu} = f^{\mathcal{M}}(|t_1|_{\mathcal{M},\mu},\ldots,|t_k|_{\mathcal{M},\mu})$.

The satisfaction of a formula α with respect to $\mathcal{M}$ and μ, written $\mathcal{M},\mu \models \alpha$ is defined inductively by:

1. $\mathcal{M},\mu \models (t = t')$ iff $|t|_{\mathcal{M},\mu}$ is identical to $|t'|_{\mathcal{M},\mu}$.
2. $\mathcal{M},\mu \models P(t_1,\ldots,t_k)$ iff $\langle |t_1|_{\mathcal{M},\mu},\ldots,|t_k|_{\mathcal{M},\mu}\rangle \in P^{\mathcal{M}}$.
3. $\mathcal{M},\mu \models \neg\alpha$ iff $\mathcal{M},\mu \not\models \alpha$.
4. $\mathcal{M},\mu \models (\alpha \wedge \beta)$ iff $\mathcal{M},\mu \models \alpha$ and $\mathcal{M},\mu \models \beta$.
5. $\mathcal{M},\mu \models \forall x.\alpha$ iff $\mathcal{M},\mu^* \models \alpha$ for every μ^* that is identical to μ except on variable x.

For sentences, α is said to be true wrt $\mathcal{M}$ iff $\mathcal{M},\mu \models \alpha$ for any μ, and false otherwise.

10.1.3 Pragmatics

The main use of first-order logic in our context is for entailment: we say that a set of sentences Σ *logically entails* a sentence α, written $\Sigma \models \alpha$ iff there is no interpretation that makes all the sentences in $\Sigma \cup \{\neg\alpha\}$ true. In other words, every interpretation where the Σ sentences are true is one where α is also true.

Logical inference is the process of determining the entailments of a given Σ. The most basic argument that a sentence α is entailed involves using interpretations as above. But an argument can also be formulated in terms of a collection of rules of inference that allow the entailment question to be broken down into simpler pieces. Here are some typical rules:

- If $\alpha \in \Sigma$, then $\Sigma \models \alpha$.
- If $\Sigma \models \alpha$, then $\Sigma \models (\alpha \vee \beta)$ and $\Sigma \models (\beta \vee \alpha)$.
- $\Sigma \models \forall x(x = x)$.
- If $\Sigma \models \alpha^x_c$ and c does not appear in Σ or α, then $\Sigma \models \forall x.\alpha$.

A collection of rules like this is said to be sound if it only allows correct entailments to be derived, and complete if all the correct entailments can be derived using just the rules. (The four rules above are sound but not complete.)

10.2 The situation calculus

The situation calculus is a dialect of first-order logic whose domain of interpretation is made up of three sorts of entities: *actions* which, as usual, are events that change properties of the world, *situations* which are histories of actions, and *objects* for everything else. The language itself has two distinguished symbols for situations: a constant S_0 intended to denote the situation where no action has yet taken place (the empty history); and a binary function symbol do, where $do(a,s)$ is intended to denote the situation that results from performing the action denoted by a in the situation denoted by s. The predicate and function symbols of the situation calculus have their normal logical interpretation and use,

except that some of them are considered to be changeable as the result of actions; these are called *fluents* and they take a situation term as their final argument.

So, for example, $Broken(x, s)$ might be the predicate that says whether the object denoted by x is broken in the situation denoted by s. The following sentence might be true:

$$\neg Broken(obj_1, S_0) \wedge Broken(obj_1, do(drop(rob, obj1), S_0))$$

Informally, this says that the object obj_1 is not broken in the initial situation, but is broken in the situation that results from doing the action $drop(rob, obj_1)$ in the initial situation. In other words, the object is not broken initially, but is broken right after the robot *rob* drops it. Note that there is no distinguished "current" situation. A single sentence in the language can talk about many different situations, past, present, and future.

There is a final distinguished symbol in the language, a predicate symbol *Poss*, where $Poss(a, s)$ is intended to be true if the action denoted by a is possible in the situation denoted by s.

10.3 Logical basic action theories

A knowledge base formulated in the language of the situation calculus will contain facts not only about what is true in the initial state of the world but also about the world dynamics, that is, how actions can change what is true.

Typically actions have *preconditions* (or prerequisites), that is, conditions that need to be true for the action to take place. These can be formulated using the *Poss* predicate. An example is below. (These examples are written using free variables, but for convenience only. They should be understood as sentences with variables universally quantified from the outside.)

$$Poss(pickup(r, x), s) \equiv \forall z \neg Holding(r, z, s) \wedge \neg Heavy(x) \wedge NextTo(r, x, s)$$

Informally, this says that a robot r can pick up an object x provided it is not holding anything, the object x is not too heavy, and the robot is located next to the object. (Note that *Heavy* here is not a fluent. This means that it is considered to be unaffected by any action.) A sentence of this form is called a <u>*precondition axiom*</u> for the action.

Actions typically also have *effects*, that is conditions that are made true by performing the action. For example, there might be this fact:

$$Fragile(x) \supset Broken(x, do(drop(r, x), s))$$

This says that one of the effects of dropping a fragile object is to break it. Similarly,

$$\neg Broken(x, do(repair(r, x), s))$$

says that one of the effects of repairing an object is to make it unbroken. Of course, not every action affects every fluent, and so it is convenient to write a single sentence for each fluent called a <u>*successor state axiom*</u> for the fluent that lists all the actions that affect it one way or another. The one for *Broken* might go like this:

$$\begin{aligned} &Broken(x, do(a, s)) \equiv \\ &\quad \exists r(a = drop(r, x) \wedge Fragile(x)) \vee \\ &\quad \exists b(Bomb(b) \wedge a = explode(b) \wedge Near(b, x, s)) \vee \\ &\quad Broken(x, s) \wedge \neg \exists r\, a = repair(r, x). \end{aligned}$$

This says that an object x will be broken after doing an action a iff a is a dropping action and x is fragile, or a is a bomb exploding where x is near the bomb, or x was already broken and a is not the action of repairing it. In other words, *Broken* is made true by dropping and exploding actions (under the right circumstances), made false by appropriate repairing actions, and left unchanged by all other actions.

A situation calculus basic action theory consists of these parts:

1. initial state axioms: arbitrary sentences whose only situation term is S_0;
2. precondition axioms: for each action A, a sentence of the form

$$Poss(A(x_1, \ldots, x_n), s) \equiv \phi(x_1, \ldots, x_n, s)$$

   where ϕ does not mention *Poss* and the only situation term it uses is the variable s;
3. successor state axioms: for each predicate fluent P, a sentence of the form

$$P(x_1, \ldots, x_n, do(a, s)) \equiv \gamma(x_1, \ldots, x_n, a, s)$$

   where the only situation term in γ is the variable s. (Similar axioms are needed for the functional fluents;
4. other domain-independent axioms whose details need not concerns us.

10.4 An example

An example basic action theory of this form is shown in Figure 10.1. This is the situation calculus version of the fox, hen, grain problem of Section 3.5.1. In this example, there are two actions, *crossAlone* and $crossWith(x)$ to move the farmer and possibly one other passenger from one bank of the river to the other, and a single fluent $OnLeft(x, s)$ that is intended to hold when object x is located on the left bank of the river in situation s. As can be seen in the specification of the initial state, there are four objects in the world (not counting the situations and actions) and they all start out on the left bank of the river.

The precondition axiom states that the farmer can cross the river alone provided it would be safe to do so: either the hen or both the fox and the grain are on the other side of the river. In addition, the farmer can cross with passenger x if x is on the same side as the farmer and it would be safe to do so: either x is the hen, or it would be safe to cross alone.

The successor state axiom characterizes how the crossing actions change the locations of objects: for any object x and action a, x will be on the left bank after a is performed iff a moves x and x was not on the left bank before or a does not move x and x was on the left bank before. (A crossing action moves x when x is the farmer or the passenger.)

10.5 Planning as reasoning

Planning can now be formulated as a logical reasoning problem. First some notation: A situation-suppressed formula is one with a single free situation variable that has been omitted. If ϕ is a situation-suppressed formula, then $\phi[t]$ is the wff that results from

Figure 10.1: The fox, hen, grain problem in logic

The function symbols: $farmer$, fox, $hen\ grain$, $crossAlone$, $crossWith(x)$
The predicate symbols: $OnLeft(x,s)$
The basic action theory:

Let $OppSide(x,y,s)$ abbreviate $\neg(OnLeft(x,s) \equiv OnLeft(y,s))$

Let $Moves(a,x)$ abbreviate
$[(x = farmer \wedge (a = crossAlone \vee \exists y\, a = crossWith(y))) \vee a = crossWith(x)]$

Initial State:
$\forall x(x = farmer \vee x = fox \vee x = hen \vee x = grain)$
$\forall x\, OnLeft(x, S_0)$

Preconditions:
$Poss(crossAlone, s) \equiv$
$[OppSide(farmer, hen, s) \vee (OppSide(fox, hen, s) \wedge OppSide(hen, grain, s))]$
$Poss(crossWith(x), s) \equiv \neg OppSide(farmer, x, s) \wedge [x = hen \vee$
$OppSide(farmer, hen, s) \vee (OppSide(fox, hen, s) \wedge OppSide(hen, grain, s))]$

Successor States:
$OnLeft(x, do(a,s)) \equiv \neg(OnLeft(x,s) \equiv Moves(a,x))$

reinstating the situation variable and then replacing it by t. Given a basic action theory Σ as above, and a situation-suppressed goal formula G, the planning problem is to find a sequence of action terms without variables $a_1, a_2, \ldots, a_n$ such that the following hold:

1. $\Sigma \models G[do(a_n, \ldots, do(a_2, do(a_1, S_0)) \ldots)]$.
2. For each $1 \leq i \leq n$, $\Sigma \models Poss(a_i, do(a_{i-1}, \ldots, do(a_2, do(a_1, S_0)) \ldots))$

In other words, the action sequence must be such that (1) it follows logically from Σ that the goal formula G holds in the final situation resulting from performing all the actions in sequence, and (2) at each step along the way, it follows logically from Σ that the next action in the sequence can be performed.

For the example problem of Figure 10.1, the goal formula G would be $\forall x \neg OnLeft(x)$ which is to say that all four objects must end up on the right bank in the final situation. It is possible (with some effort) to show that the planning problem as formulated above is indeed solved by the following sequence of actions:

$$\langle\, crossWith(hen),\ crossAlone,\ crossWith(fox),\ crossWith(hen),\ crossWith(grain),\ crossAlone,\ crossWith(hen)\,\rangle$$

10.6 Golog

Just as planning can be formulated as a logical reasoning problem, so can the execution of programs. To do so, for each program δ under consideration, a formula of the situation

calculus $Do(\delta, s_1, s_2)$ is defined. This formula is intended to hold when the program δ started in situation s_1 can terminate in situation s_2. (We say "can terminate" instead of "will terminate" since the programs may be nondeterministic.)

The programming language Golog is the ancestor of $\mathcal{ERGO}$ and has constructs similar to those of Chapter 4. The *Do* formula for these is defined as follows:

- primitive actions: $Do(a, s_1, s_2) \doteq Poss(a, s_1) \wedge s_2 = do(a, s_1)$.
- sequence: $Do(\delta_1 ; \delta_2, s_1, s_2) \doteq \exists s. Do(\delta_1, s_1, s) \wedge Do(\delta_2, s, s_2)$.
- test action: $Do(\phi?, s_1, s_2) \doteq \phi[s_1] \wedge s_2 = s_1$. (where ϕ is situation-suppressed)
- nondeterministic choice: $Do(\delta_1 \mid \delta_2, s_1, s_2) \doteq Do(\delta_1, s_1, s_2) \vee Do(\delta_2, s_1, s_2)$.
- nondeterministic pick: $Do(\pi x. \delta, s_1, s_2) \doteq \exists x. Do(\delta, s_1, s_2)$.
- nondeterministic iteration: $Do(\delta^*, s_1, s_2) \doteq \forall P(\ldots \supset P(s_1, s_2))$
  where the ellipsis stands for: $\forall s P(s, s) \wedge \forall s, s', s''(P(s, s') \wedge Do(\delta, s', s'') \supset P(s, s''))$.

(The Golog language includes other constructs not discussed here.) Note that the *Do* formula for the nondeterministic iteration is in fact a formula of *second-order logic* whose details need not concern us.

An example of *Do* with the basic action theory of Figure 10.1 is the following:

$$\begin{aligned}
&Do(\pi x.[\neg OppSide(farmer, x)? ; crossWith(x) ; crossAlone], s_1, s_2) \\
&\quad = \exists x.\, Do([\neg OppSide(farmer, x)? ; crossWith(x) ; crossAlone], s_1, s_2) \\
&\quad = \exists x, s, s'.\, Do(\neg OppSide(farmer, x)?, s_1, s) \wedge Do(crossWith(x), s, s') \wedge \\
&\qquad\qquad Do(crossAlone, s', s_2) \\
&\quad = \exists x.\, \neg OppSide(farmer, x, s_1) \wedge Poss(crossWith(x), s_1) \wedge \\
&\qquad\qquad Poss(crossAlone, do(crossWith(x), s_1)) \wedge \\
&\qquad\qquad s_2 = do(crossAlone, do(crossWith(x), s_1))
\end{aligned}$$

With the *Do* formula defined as above, the execution of programs in general can now be defined. Given a basic action theory Σ and a Golog program δ, the task of offline execution is to find a sequence of action terms $a_1, a_2, \ldots, a_n$ such that the following holds:

$$\Sigma \models Do(\delta, S_0, do(a_n, \ldots, do(a_2, do(a_1, S_0)) \ldots)).$$

In other words, offline execution consists in finding a sequence of actions that constitute a legal execution of the program starting in S_0 according to the *Do* formula. For the program above, the offline execution is solved by the sequence $\langle crossWith(hen), crossAlone \rangle$ since the sentence $\neg OppSide(farmer, hen, S_0)$ is entailed, as are the two required *Poss* formulas. As in Section 4.4, planning is once again seen as a special case of offline execution in that any offline execution of the program $[(\pi a.\, a)^*; G?]$ is a plan that will achieve the goal G.

10.7 From Golog to *ERGO* and back

In this section, the relationship between Golog (as presented above) and *ERGO* (as seen in the rest of the book) is considered. Roughly speaking, the idea is to come up with a translation between *ERGO* and Golog such that

$$\texttt{(ergo-do}\ \delta\texttt{)}\ \text{returns}\ \vec{a} \quad \text{iff} \quad \Sigma \models Do(\delta, S_0, do(\vec{a}, S_0)).$$

As will become clear, however, the match between the two formalisms is not quite exact.

10.7.1 Programs

Perhaps the easiest correspondence between *ERGO* and Golog concerns the programs themselves. It is clear that there is a direct correspondence between the programming constructs seen in this chapter and those in Chapter 3. For example, sequence is handled with ; in Golog and with `:begin` in *ERGO*, but the effect is the same. Similarly, the π operator in Golog corresponds to the `:for-some` of *ERGO*, the main difference being that *ERGO* requires a list of elements for the quantification, whereas Golog effectively iterates over the entire domain. There are primitives for `:if` and `:while` in *ERGO*, whereas these would need to be expressed in Golog as abbreviations using the given constructs:

$$\begin{aligned} \textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2 &\doteq [(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)] \\ \textbf{while}\ \phi\ \textbf{do}\ \delta &\doteq [(\phi?;\delta)^*;\neg\phi?] \end{aligned}$$

It is also possible to provide a logical account of the concurrency and online aspects of *ERGO*, but this would take us too far afield.

10.7.2 Action preconditions and effects

Turning now to the preconditions of actions, the correspondence between the two formalisms is again clear. What is written in *ERGO* as

$$\texttt{(define-action}\ A\ \ldots\ \texttt{\#:prereq}\ e\ \ldots\texttt{)}$$

would be translated in the situation calculus into something like

$$\forall s.\ Poss(A, s) \equiv e$$

and vice-versa. To deal with the effects of actions, it is necessary for each fluent f to identify all the actions in an *ERGO* basic action theory that can change the value of that fluent. So, for example, given actions like

$$\begin{array}{l} \texttt{(define-action}\ A_1\ \ldots\ f\ e_1\ \ldots\texttt{)} \\ \qquad\vdots \\ \texttt{(define-action}\ A_k\ \ldots\ f\ e_k\ \ldots\texttt{)} \end{array}$$

the corresponding successor state axiom for a functional fluent f is written as

$$\begin{array}{l} f(do(a,s)) = v \equiv \\ \quad (a = A_1 \wedge v = e_1) \vee \\ \quad\ \vdots \\ \quad (a = A_k \wedge v = e_k) \vee \\ \quad (v = f(s) \wedge a \neq A_1 \wedge \ldots \wedge a \neq A_k) \end{array}$$

(The case where the fluent f has a Scheme function as its value would be handled just like the initial state axioms, with universally quantified arguments in the axiom. The case where an action A_i has arguments would be handled by existential quantifiers in the successor state axiom.) Going from a situation calculus successor state axiom to *ERGO* is possible only when the axiom can be massaged into a form similar to the one above.

10.7.3 Initial states

The actions taken by an *ERGO* program during execution depend, of course, on what is known about the initial state, and Golog is no different. What is quite different, however, is how this initial state is characterized in the two formulations. In *ERGO*, the initial state is characterized by providing the initial values of the fluents (using the `define-fluents` primitive); in the situation calculus, it is done using the initial state axioms. In the case where the values of the fluents are Booleans or symbols, an expression like

$$\texttt{(define-fluents } f_1\ e_1\ \ldots\ f_k\ e_k\texttt{)}$$

in *ERGO* would be translated to an initial state axiom like

$$f_1(S_0) = e_1 \wedge \cdots \wedge f_k(S_0) = e_k.$$

When the values of fluents are Scheme functions, then

$$\texttt{(define-fluents } f\ \texttt{(lambda (}x_1\ \ldots\ x_n\texttt{)}\ e\texttt{))}$$

could be translated to something like

$$\forall x_1 \cdots \forall x_n.\ f(x_1, \ldots, x_n, S_0) = e.$$

When the values of fluents involve other datatypes of Scheme, such as numbers, lists, tables, and so on, the initial state axioms would need to contain axioms defining the operations on objects of those types. (It is typical, however, to use a variant of first-order logic where the properties of *numbers* are built in.)

Going in the other direction on the other hand, that is, from the situation calculus to *ERGO*, is possible only when the initial state axioms uniquely determine the values of the fluents. For example,

$$f(S_0) = 3$$

can be translated into *ERGO* in the obvious way, but the initial state axiom

$$f(S_0) = 3 \vee f(S_0) = 4 \vee f(S_0) = 5$$

cannot be translated into *ERGO* at all. This is due to the fact (stated at the start of Chapter 3) that the BAT representation of *ERGO* assumes *complete knowledge* about the initial values of the fluents. The case of an *ERGO* with incomplete knowledge is considered only in Chapters 11 and 12. To handle initial state axioms more generally in *ERGO*, something like the `#:known` keyword introduced in Section 11.1 would be needed.

10.8 Bibliographic notes

This chapter deals with the logical foundations of cognitive robotics. Logic itself has a long history that in Western culture goes back to the ancient Greeks. The modern form of symbolic logic as used in this chapter is due to Gottlob Frege, with a specific notation due to Giuseppe Peano, in the early 1900s. Two excellent mathematical textbooks on modern symbolic logic are [14] and [34].

The original motivation behind the development of symbolic logic was to put all of mathematical reasoning on a sound footing. But it was John McCarthy, one of the founders of the field of artificial intelligence (AI) [41], who first proposed using symbolic logic not for mathematics, but to represent the commonsense knowledge of an agent [33]. A textbook on this topic is [6], and a more advanced handbook is [20].

McCarthy's original paper introduced the situation calculus and the idea of planning as a form of logical reasoning. His formulation suffered from what was called the frame problem [42]: to work properly, a logical axiomatization had to specify not just the effects of actions, but all their myriad non-effects as well. It was Ray Reiter who, building on some earlier work, proposed a solution to this frame problem in [39] in terms of what he called basic action theories, as presented in this chapter. Reiter went on to write a much more comprehensive study of the situation calculus in [40]. See [32] for a survey of this and subsequent work on the situation calculus.

Reiter's formulation of the situation calculus was very influential and became the main foundation of what was to become cognitive robotics. (Reiter coined the term in 1993.) Among many other things, Reiter and colleagues developed the Golog language discussed in this chapter [31]. Subsequent work incorporated concurrency [12] and made the distinction between offline and online execution [13], as seen throughout this book. In a very real sense, this book is an attempt to extract the programming ideas from the on-going research in cognitive robotics that began with Ray Reiter. For a survey of cognitive robotics from this knowledge representation point of view, see [30].

* Chapter 11

Numerical Uncertainty

In previous chapters, a cognitive robot was assumed to have complete knowledge of the initial state of the world. A robot might not know how the world was changing beyond its own actions, and we considered how active sensing using exogenous actions could deal with this in Chapters 5 and 6. But in all cases, we assumed that the *initial values* of the fluents were known, so there was no need for sensing there. In more realistic settings, however, even the initial knowledge of a cognitive robot can be incomplete. Unlike in Chapter 3, the robot may not know which of two rooms contains a certain box, for example, and it may need to go to those rooms to find out. Similarly, the robot may only have a rough idea of how far it is from a nearby wall and need to use its onboard sensors to get a more accurate picture. In this chapter and the next, we consider how a cognitive robot can decide what to do in the presence of knowledge that is incomplete in this way.

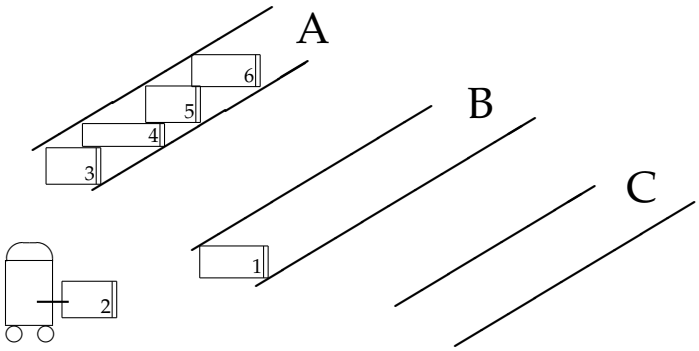
The easiest way to think about the sort of incomplete knowledge we have in mind is in terms of the representation of knowledge it uses. In previous chapters, what the robot knew about the world could be represented by what we called a *state*, that is, a mapping from fluents to their values. In this chapter and the next, a robot will not be able to use a state as its representation, since it may know certain things about some fluents without knowing their values. Instead, the robot will have to make do with a *set of possible states*, any one of which might be the correct representation of the world. For example, if all the robot knows about a fluent f is that it must have value x or y, then there will be some possible states where the value of f is x and others where its value is y. We say that x and y are *possible values* for the fluent f. The only conditions the robot knows for sure are those conditions that come out true in all these possible states.

In practice, there might be very little that a cognitive robot actually knows for sure. But as we will see in this chapter, it may be able to use some numerical information it has to make an informed guess about the most likely cases. (In Chapter 12, we will see how a robot might still be able to plan courses of action without this numerical information.)

11.1 Incomplete knowledge

Before looking at an example BAT with incomplete knowledge, let us consider a simple case of complete knowledge. Imagine a robot located in a work area where it can pick up and put down objects (such as bricks or parcels) in three separate stacks named A, B, and

Figure 11.1: A world with three stacks of objects



C. At any point, the robot may be holding one of these objects. See Figure 11.1. Let us suppose that each object in a stack can be coloured red or blue. We could represent this information using a hash-table that maps objects to their colour. However, if colour is the only property of an object that the robot really cares about, we can simply list the objects in a stack according to their colour.

A basic action theory for this world is shown in Figure 11.2. There are no objects initially in stacks B and C, and stack A contains four objects identified only by their colours: red, red, blue, and red. The goal of the robot (the `goal?` function) is to make two towers of uniform colour, that is, to get the all red blocks onto stack B and all the blue objects onto stack C. A plan for this goal can be found using the basic planner from Chapter 3:

```
> (ergo-simplan goal? (append (map pick! '(A B C)) (map put! '(A B C))))
'((pick! A) (put! B) (pick! A) (put! B)
  (pick! A) (put! C) (pick! A) (put! B))
```

The generated plan obviously depends on the initial list of blocks in stack A. In fact, for any list of blocks in stack A, the planner can find a plan that works for that initial state.

11.1.1 The `define-states`, `possible-values`, and `known?` functions

Let us now complicate the picture and suppose that the robot has *incomplete knowledge* of the initial state of this world. In particular, assume that the robot does not know at the outset the colour of the second block on stack A, but that everything else is known as before. In other words, as far as the robot is concerned, the initial state of the world is one of two possibilities: either as given in Figure 11.2 with stack A starting as `(red red blue red)`, or exactly the same state except that stack A starts as `(red blue blue red)`.

To represent this incomplete knowledge, we can use the BAT of Figure 11.2 except that instead of `define-fluents`, the *ERGO* function `define-states` is used:

```
(define-states ((x '(red blue)))
   hand   'empty
   stacks (hasheq 'A `(red ,x blue red) 'B '() 'C '()))
```

Figure 11.2: A basic action theory for the three stacks world

```
;;;  A simple robot world involving objects on three stacks, A, B, and C.
;;;  There are two fluents
;;;      hand    the object held in the robot's hand or 'empty'
;;;      stacks  the contents of each stack as a list of objects
;;;  There are two actions:
;;;      pick!   pick up the first object from a stack
;;;      put!    push an object onto a stack

(define-fluents
   hand    'empty
   stacks  (hasheq 'A '(red red blue red) 'B '() 'C '()))

;; abbreviation to get the content of a stack A, B, or C
(define (stack st) (hash-ref stacks st))

;; action to pop an object from the top of a stack
(define-action (pick! st)
   #:prereq  (and (eq? hand 'empty) (not (null? (stack st))))
   hand      (car (stack st))
   stacks    (hash-set stacks st (cdr (stack st)))  )

;; action to push what is being held to the top of a stack
(define-action (put! st)
   #:prereq  (not (eq? hand 'empty))
   hand      'empty
   stacks    (hash-set stacks st (cons hand (stack st)))  )

(define (goal?)
   (and (eq? hand 'empty) (null? (stack 'A))
        (for/and ((o (stack 'B))) (eq? o 'red))
        (for/and ((o (stack 'C))) (eq? o 'blue)) ))
```

This function is very much like `define-fluents` except that it constructs a *list* of states rather than a single one. The general form is this:

`(define-states ((`v_1 d_1`) ... (`v_k d_k`))` f_1 e_1 ... f_n e_n`)`

where the f_i are fluents and the e_i are their values, as before with `define-fluents`. The difference here is v_j and the d_j. The v_j are variables that may be used in any of the e_i and the d_j are expressions that evaluate to lists. The idea is that we will get initial states for all possible values of the variables v_j taken from the lists d_j. (A d_j fexpr can also evaluate to a positive integer m in which case the v_j has a value taken from $0, 1, \ldots, m-1$.) So, for example, if we had $k = 3$ where the list d_1 had three elements, d_2 had two elements, and d_3 had four elements, `define-states` would produce a list of $3 \times 2 \times 4 = 24$ initial states.

In the example above, a list with two initial states is defined. The two states differ only in the colour x of the second block in stack A. There are two possible values for this block, which can be confirmed with the *ERGO* function `possible-values`:

```
> (possible-values (cadr (stack 'A)))
'(blue red)
```

```
> (possible-values (car (stack 'A)))
'(red)
```

As mentioned above, an fexpr is considered *known to be true* by the robot if it comes out true in all of these states. We can use the *ERGO* function known? as follows:

```
> (known? (eq? (caddr (stack 'A)) 'blue))          ; the third block is blue
#t
```

The value is true since the eq? expression is true in both initial states. However, we have

```
> (known? (eq? (cadr (stack 'A)) 'blue))          ; the second block is blue
#f
> (known? (eq? (cadr (stack 'A)) 'red))           ; the second block is red
#f
```

since in each case there is a state in the list where the expression comes out false. More generally, we say that an fexpr is known to have value x if it has the same value x in all the states. (In other words, x is its only possible value.) Note that something can be known about an fexpr even if its value is not known:

```
> (known? (memq (cadr (stack 'A)) '(red blue)))  ; the second block is red
#t                                               ; or blue
```

Although the colour of the second object is not known, it is known to be red or blue.

With this view of incomplete knowledge, the larger the list of states, the less complete is the knowledge of the robot. For example, if the robot only knows the colour of the top two blocks on stack A, we would use a list with four states to capture the range of possibilities for the third and fourth blocks:

```
(define-states ((x '(red blue)) (y '(red blue)))
   hand   'empty
   stacks (hasheq 'A `(red blue ,x ,y) 'B '() 'C '()))
```

For this list of states, we have the following:

```
> (known? (not (eq? (car (stack 'A)) (cadr (stack 'A)))))
#t
```

In other words, the first object in the stack is known to be different in colour from the second. However, we have the following:

```
> (known? (eq? (car (stack 'A)) (caddr (stack 'A))))
#f
> (known? (not (eq? (car (stack 'A)) (caddr (stack 'A)))))
#f
```

In other words, the first object in the stack is not known to have the same colour or to have a different colour from the third object. Observe that the value of

```
(length (stack 'A))
```

would be known here, since this fexpr has the same value in all the states. Nothing stops us, however, from considering initial states where even that value is unknown:

```
(define-states ((u '(() (red) (blue) (red blue) (red red blue))))
   hand   'empty
   stacks (hasheq 'A u 'B '() 'C '()))
```

In this case, the number of objects in stack A is unknown, although we do have this:

```
> (known? (< (length (stack 'A)) 10))
#t
```

(With any finite list of states, there will always be a known upper bound on the length of any stack. There is no way in this representation to represent knowing absolutely *nothing* about the number of objects in stack A.)

As a final convenience in specifying the initial state of knowledge, the `define-states` function can take an optional keyword `#:known` followed by a Boolean fexpr e_0 so that

`(define-states ((`v_1 d_1`) ... (`v_k d_k`)) #:known` e_o f_1 e_1 ... f_n e_n`)`

behaves just like before except that only states where e_0 is true are considered. An example is the following:

```
> (define tf '(#t #f))
> (define-states ((x1 tf) (x2 tf) (x3 tf) (x4 tf))
>    #:known (or p q)
>    p x1  q x2  r x3  s x4)
```

In this case, instead of getting $2 \times 2 \times 2 \times 2 = 16$ initial states, we get only 12 of them, just those where either `p` or `q` comes out true. In general, if we want to represent a state of knowledge where some Boolean formula ϕ is what is known initially (as seen in the initial state axioms of Section 10.7.3), we can let the fluents range over all their possible values, and then constrain the set of initial states using `#:known` with ϕ as the e_0.

11.2 Degrees of belief

The starting point for dealing with numerical uncertainty in *ERGO* is the representation of incomplete knowledge described in the previous section. But for fluents denoting *numeric values* like distance, depth, weight, temperature, voltage, sound intensity, and the like, it might not make sense to try to consider all the possible values the fluents could have.

Consider, for example, a robot that wants to be located five metres away from a wall. A motor may be able to move the robot a given distance towards the wall, but not with total accuracy. A sonar sensor may be able to report the current distance to the wall, but only to within a certain tolerance. (A laser rangefinder may do the same job but more accurately.) In cases such as these, the robot may never know its true distance to the wall, but will have to consider what that distance could possibly be. Typically, there will be an (uncountably)

infinite number of possible values to contend with, but some of these values may be much more likely than others.

For some applications, of course, an assumption of completely accurate sensors and effectors is quite justified. We might have a primitive action of moving ahead one metre, for instance, and leave it to the robot manager to perform that action as best as it can with its onboard sensors and effectors. This is how we think of an elevator moving to a given floor in a building, for example. But in other cases, small errors can accumulate. We might want a robot to use what it knows about the world and the goals it is working on to decide how worthwhile it might be to obtain a more accurate assessment of what it is doing. That is to say, we might want to consider *ERGO* programming that deals explicitly with this uncertainty.

11.2.1 The `weight` fluent and the `belief` function

Let us return to the example world of Figure 11.1. Suppose, for example, that we do not know the colour of the second block in stack A, but that we have good reason to suspect that it is red. We might want to allow for the possibility that it is blue while considering this to be much less likely. To model such a state of knowledge, we want to give some of the states in the list of all possible states a higher ranking than others. A *weight* in this case, is a number $w \geq 0$ that indicates in a relative way how certain we are that a given state is a correct representation of the world. If we have n states in the list, $s_1, \ldots, s_n$, each one is considered to have a weight, $w_1, \ldots, w_n$, so that when $w_i > w_j$ we consider state s_i to be more likely than state s_j to be the correct representation of the world.

By default, each of the states created by `define-states` is given an equal weight of 1.0. To change this initial weight, a special fluent (or meta-fluent, perhaps) `weight` can be used explicitly in the `define-states` expression. For example, in the following definition,

```
(define-states ((x '((4 red red red) (3 red red blue)
                     (2 red blue red) (1 red blue blue))))
   hand   'empty
   stacks (hasheq 'A (cdr x) 'B '() 'C '())
   weight (car x))
```

we have four initial states, where the possible values for stack A are: follows:

s_1 =`(red red red)`
s_2 =`(red red blue)`
s_3 =`(red blue red)`
s_4 =`(red blue blue)`.

The corresponding weights for the four states are: $w_1 = 4$, $w_2 = 3$, $w_3 = 2$, $w_4 = 1$. We can see that s_1, where the three blocks are red, is the most likely possible value, and s_4, where only the the first block is red, is the least likely. Of course, we would have said the same thing if we had $w_1 = 44$. The difference is that in that case we would have thought s_1 to be *much* more likely than its alternatives. The actual absolute values for the w_i do not mean anything, but we do care about the *relative values* of the weights, the fact that w_1 is four times or forty four times as high as w_4.

To make this idea precise, we define the notion of the *degree of belief* (sometimes called the *subjective probability*) of a Boolean fexpr e as a real number from 0 to 1 defined as the sum of the weights of the states where e is true divided by the sum of the weights of all the states. In mathematical notation, it's this:

$$\frac{\sum_{i=1}^{n} \begin{cases} w_i & \text{if } e \text{ is true} \\ 0 & \text{otherwise} \end{cases}}{\sum_{i=1}^{n} w_i}.$$

So, for example, for the four initial states above, we have the following degrees of belief:

Proposition about stack A	Degree of belief
Block 1 is red	1.0
Block 2 is red	0.7
Block 3 is red	0.6
Blocks 2 and 3 are the same colour	0.5
Blocks 1, 2 and 3 are all the same colour	0.4
There are more blue blocks than red ones	0.1

Note that these degrees of belief would have been exactly the same if all the weights had been multiplied by the same factor, like $w_1 = 12, w_2 = 9, w_3 = 6, w_4 = 3$. This is because we end up "normalizing" the weights, that is, dividing each weight by the sum of all of them. Note also that the degrees of belief would have been identical had there been additional states with a weight of 0.0. Finally, observe that in this scheme, formulas that are known to be true will get a degree of belief of 1.0, and formulas known to be false will get a degree of belief of 0.0. (If there are no states with a weight of 0.0, the converse also holds: formulas with a degree of belief of 1.0 are known to be true, and formulas with a degree of belief of 0.0 are known to be false.)

All degrees of belief other than 1.0 and 0.0 are for formulas that are not known to be true or to be false, where the degree measures how far we are from those two possibilities. So, for example, if we had $w_1 = 44$ instead of $w_1 = 4$ (and the other three weights the same), the degree of belief that there are more blue blocks than red ones would have fallen from 0.1 to 0.02. In that case, the formula is not quite known to be false, but very close. As we will see, there will be times when a practical robot may need to treat formulas where the degree of belief is this low as false.

It is also useful to talk about the *conditional belief* in e_1 given e_2. This is defined as the degree of belief in the conjunction of e_1 and e_2 divided by the degree of belief in e_2. (The conditional belief is undefined when the degree of belief in e_2 is 0.0.) For the block example above, the degree of belief that block 3 is blue is .4, but the degree of belief that block 3 is blue given that block 2 is blue is about .33. This is a way of saying that our confidence that block 3 is blue is somewhat lower if we limit ourselves to states where block 2 is blue.

To test the degree of belief in a condition, we can use the *ERGO* function `belief` which is like the `known?` function except that it returns a number between 0.0 and 1.0:

```
> (belief (eq? 'red (car (stack 'A))))
1.0
```

```
> (belief (eq? (cadr (stack 'A)) (caddr (stack 'A))))
0.5
> (define (count-colour c) (for/sum ((x (stack 'A))) (if (eq? x c) 1 0)))
> (belief (> (count-colour 'blue) (count-colour 'red)))
0.1
```

We can also provide a second argument to the `belief` function for conditional belief:

```
> (belief (eq? 'blue (caddr (stack 'A))) (eq? 'blue (cadr (stack 'A))))
0.33333
```

11.2.2 Numeric fluents: `sample-mean` and `sample-variance`

When we are dealing with fluents with numeric values and there is uncertainty about those values, it is often useful to deal with the standard statistical notions of mean and variance. The *mean* of a numeric expression (or its *expected value*) is its weighted average across all states. More precicely, the mean of f is the normalized sum over all states of the product of the value of f in that state and the weight of that state. In mathematical notation, the mean μ is defined as follows:

$$\mu = \frac{\sum_{i=1}^{n} w_i \cdot f_i}{\sum_{i=1}^{n} w_i}.$$

So, for example, suppose we have the four states with the four weights above and the value of some fexpr f in each state is as follows: $f_1 = 5, f_2 = 3, f_3 = 5, f_4 = 6$. Then the mean value of f for these four states would be

$$\mu = \frac{(5 \cdot 4 + 3 \cdot 3 + 5 \cdot 2 + 6 \cdot 1)}{10} = 4.5$$

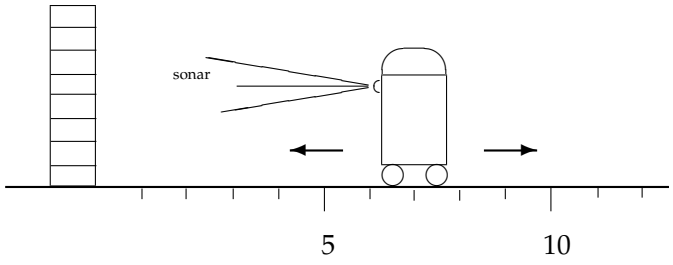
When all states have equal weight, the mean is the same as the ordinary average value.

Note that there is a close connection between means and degrees of belief: the degree of belief in e can be defined as the mean value of the expression `(if` e `then 1 else 0)`.

Faced with inaccuracies and uncertainty, it will sometimes be necessary to use the mean as a guess for an unknown numeric value. The *ERGO* function `(sample-mean` e`)` can be used for this purpose. There are other possible guesses we could consider, but the mean has the advantage of being neither too low nor too high, in the sense that the total error we might encounter from being too low is balanced by the total error from being too high. For example, we might have considered using 5 as a guess for the value of f (which is the value f takes in two of the four states, after all), but there's a chance this could be as much as 2 units too high (in case state s_2 turns out to be the correct value). We might also consider using the unweighted average of 4.75 as the guess, but this seems to give undue prominence to the possible value of 6 (for the f in s_4), which has a low weight relative to the others. The mean value of 4.5 strikes the right balance.

One thing that might be done before using the mean as a guess is to look at the amount of uncertainty we have about the value. Suppose we have a numeric expression e whose

Figure 11.3: A one-dimensional robot world



mean value is μ. The *variance* of e is defined as the mean value of $(e - \mu)^2$. For the four states above, the variance of f would be

$$\frac{(.25 \cdot 4 + 2.25 \cdot 3 + .25 \cdot 2 + 2.25 \cdot 1)}{10} = 1.05$$

If the four values had all been 4.5, the mean would have been the same, but the variance would be 0, indicating that the value of f is known. Had the four values of f been 7, 2, 4 and 3 with the same weights as before, then the mean would again have been 4.5, but this time the variance would be higher, about 4.65, indicating a higher degree of uncertainty about the value of f. The *ERGO* function `(sample-variance e)` can be used to obtain the variance of expression e.

While the variance is often used as a measure of the spread of the values of f, there are again other expressions we might have considered. We cannot use the mean value of $(f - \mu)$ (without the square) since its value is always 0 (the negative values cancelling out the positive ones). But we could have used the mean value of $|(f - \mu)|$, or the mean value of $(f - \mu)^4$, or perhaps the degree of belief in some formula like $(|(f - \mu)| < |.9\mu|)$ as a measure of uncertainty. The variance is simply a very convenient measure, and one that is easy to calculate. We sometimes also use the *standard deviation*, defined as the square root of the variance, as the measure of uncertainty, as it has the advantage of undoing the squaring, and so deals with units closer to those of the mean.

11.3 Dealing with noise and inaccuracy

As a concrete example of dealing with the uncertainty that arises from the inaccuracy of sensors and effectors, consider the situation of a mobile robot that happens to be located 7 metres away from a wall and that wishes to be located 5 metres away, as depicted in Figure 11.3. If the robot knows where it is located and there is a primitive action to move the robot forward 2 metres, we could handle this as before. But suppose the robot does not know where it is located, somewhere between 2 and 12 metres away from the wall. Without a sensor, the robot would be stuck. But suppose the robot happens to have a sonar sensor. If the sonar gives a reading that is guaranteed to be accurate, such as 7.0 metres, then again it is clear what the robot should do. If the sonar gives a reading like 7.2 that is not guaranteed to be exact, then the robot can take multiple independent readings to get a

clearer picture of where it is located, and then at some threshold, stop and make a move. But finally let us suppose that the motors themselves are also inaccurate. The robot might request a move of 2.2 metres but end up moving 2.217 metres. The best the robot could do at that stage would be to use the sonar again to try to find out how close it is to the desired 5 metres, and either settle for that, or consider additional smaller moves. As long as the motors are not so inaccurate as to move the robot further and further away from the goal of 5 metres, eventually the robot will be able to get close enough.

What we have sketched in the previous paragraph is in effect a program that uses inaccurate sensors and effectors to achieve a goal to within a certain tolerance. In this example, we have a single fluent representing the current distance to the wall, and two somewhat inaccurate operations, one that moves the robot towards the wall, and one that senses the distance to the wall. We now develop an *ERGO* program along these lines, starting with the BAT.

11.3.1 Random numbers and sampling

In the BAT for this domain, we will have a single fluent `h` representing (horizontal) distance to the wall, and whose value is not known. Let us suppose, however, that the value is known to lie somewhere between 2 and 12 metres. We cannot represent this incomplete knowledge with a state for each possible value of `h` since there are uncountably many such values. Instead, we work with a *sample* of these values, as follows:

```
(define-states ((i 1000000))
   h (UNIFORM-GEN 2.0 12.0))                          ; distance to wall
```

We are using one million initial states, not because there are exactly that many possible values for `h`, but to have a large number of randomly-selected sample values that have an appropriate mean and variance.

The *ERGO* functions `UNIFORM-GEN`, `GAUSSIAN-GEN`, `DISCRETE-GEN`, and `BINARY-GEN` can be used to generate random numbers. The expression `(UNIFORM-GEN` x y`)` has as its value a floating point number selected at random according to a uniform distribution between x and y. Similarly, `(GAUSSIAN-GEN` x y`)` has as its value a floating point number selected at random according to a Gaussian (or normal) distribution with mean x and standard deviation y. In other words, it generates random numbers that cluster around a mean value of x with a variance of y^2. (In practice, about 68% of the generated numbers will lie between $(x - y)$ and $(x + y)$ and about 95% of them will lie between $(x - 2y)$ and $(x + 2y)$.) To generate random values chosen from a fixed list of choices, the expression `(DISCRETE-GEN` v_1 p_1 ... v_n p_n`)` where the p_i add up to 1, generates values from the v_i according to the probability p_i. Finally, the expression `(BINARY-GEN` p`)` is an abbreviation for `(DISCRETE-GEN #t` p `#f (- 1.0` p`))`.

(A side note on random numbers: The random numbers produced by *ERGO* are of course only *pseudo* random numbers generated by the underlying Scheme implementation. Scheme provides a way of "seeding" its pseudo-random-number generator, that is, initializing it so that the same sequence of numbers will be produced in distinct runs of the program. This can be useful for debugging, but is beyond the scope of this book.)

To return to the BAT, what we are doing with the `define-states` above is generating a list of one million initial states where the `h` fluent takes values from 2.0 to 12.0 in a

uniformly random way. So for example, the degree of belief that h is between 3 and 4 (say), will be the same as the degree of belief that it is between 6.5 and 7.5. (This is how a uniform distribution is supposed to work in theory. By using sampling, we forgo these precise theoretical results and make do with approximations. How close we get to the theoretical values depends on how many sample states we generate.) If we wanted instead to say that the unknown value of h was somewhere around 7 metres, with the range between 3 and 4 much less likely than the range between 6.5 and 7.5, we could have used something like this:

```
(define-states ((i 1000000))
   h (GAUSSIAN-GEN 7.0 1.5))
```

Among the million samples generated, the range of h values will be centered around 7.0, with about 95% of the values lying between 4.0 and 10.0.

Another option in the sampling is to choose values along exact intervals between 2.0 and 12.0 and to weigh those samples explicitly:

```
(define-states ((i 1000000))
   h      (+ 2.0 (* 10.0 (/ i 1000000)))
   weight (GAUSSIAN h 7.0 1.5))
```

The *ERGO* functions UNIFORM, GAUSSIAN, DISCRETE, and BINARY are random number testing functions analogous to the random number generating functions described above. In all cases, the functions are given an additional first argument z and return a number between 0 and 1 measuring the relative likelihood of the z value. So the (GAUSSIAN h 7.0 1.5) in the definition here says the weight of a sample should be the likelihood of getting its value of h, assuming that those values were distributed according to a Gaussian with mean 7.0 and standard deviation 1.5. (This method of sampling has the advantage of avoiding any randomness in the sampling process, but has the disadvantage of having many samples with weights very close to 0. The other method is usually more accurate.)

Note that we can have fluents with uncertain values mixed with fluents with certain values by using something like this:

```
(define-states ((i 1000000))
  h    (UNIFORM-GEN 2.0 12.0)
  q    7
  temp (DISCRETE-GEN 'hot .2 'cold .6 'tepid .2))
```

In this case, we would get the following. The h fluent will be as above. The temp fluent will get the value hot in about .2 of the states, cold in about .6 of them, and tepid in the rest. The fluent q will get the value 7 in all states. In other words, the mean value of q will be 7.0 and its variance will be 0.0. Since the temp value is not numeric, there is no mean or variance, but the degree of belief that the value of temp is either cold or tepid should be about .8. (See Section 11.4 for dealing with fluents whose values are not independent.)

11.3.2 Noisy actions

The next thing we need in the BAT is the action that moves the robot towards the wall, reducing the value of h:

```
(define-action (move! x)                                    ; advance x units
   h (max 0 (- h (* x (GAUSSIAN-GEN 1.0 .2)))))
```

Because of inaccuracies in the motor or for other reasons, the value of `h` after the action `(move! 1)` will not be exactly `(- h 1)`, but will deviate from this value like a random number chosen from a Gaussian distribution with standard deviation .2. Even if we knew the value of `h` precisely before the move, after the move, we would no longer know its value, although we expect it to be close to `(- h 1)`. How close? This depends on the properties of the effector. For a very accurate motor, the standard deviation should be small; for an inaccurate motor, it might be large. In other examples, the mean might not even be `(- h 1)`; there may be a bias in the motor that makes it more likely to overshoot than to undershoot. Similarly, we can use an expression like

```
(define-action (move! x)                                    ; advance x units
   h (max 0 (- h (* x (GAUSSIAN-GEN 1.0 (if slippery .5 .1))))))
```

to say that the variance in the moving can depend on other considerations, here whether or not the `slippery` fluent is true at the time of the move.

11.3.3 Noisy sensing

The next two actions in the BAT are for sensing:

```
(define-action sonar-request!)                              ; sonar endogenous
(define-action (sonar-response! z)                          ; sonar exogenous
   weight (* weight (GAUSSIAN z h .4)))
```

As in Section 6.3, the sensing involves a request for sensing information followed by an exogenous report. The first action, `sonar-request!`, will be used as an ordinary action (of no arguments) in the *ERGO* program. The second, `sonar-response!`, will be used as an exogenous action (of one argument) that happens implicitly. When `(sonar-response!` z`)` happens for some value z, the z is then understood to be the value returned by the sonar sensor asked to measure the current distance to the wall.

How the z returned by `sonar-response` is then used is quite different from what we did before. We do not want to try to update the `h` fluent according to this value as we did in Chapter 6. It's not as if we learned that our previous value of `h` is out of date and that the distance to the wall has changed to z. Rather, what we want to do is to update our *beliefs* about the distance to the wall in a way that accommodates both our previous beliefs and this new bit of sensing information.

The way we do this is to change the weight of each sample according to the value z returned. Typically, we want the new weight to be the product of two numbers: the previous weight and the number between 0 and 1 returned by the `GAUSSIAN` testing function. The effect of the first number is to give higher weights to those sample states that were considered more likely before; the effect of the second number is to give higher weights to those sample states whose `h` values are closer to the z value returned by the sonar.

In other words, although we do not want to assume that the sonar gives us the true value of `h`, we do want to assume that it will give a value z that is close. Hence, we want to rate more highly those states whose `h` values are close to this z. How seriously should

we take this z value? This depends on the sensor. So the expression `(GAUSSIAN z h .4)` in the definition in effect says that the sensing results z from the sonar will deviate from the true value of `h` like a random number chosen from a Gaussian distribution with standard deviation .4. Other sensors might be more or less accurate and so have smaller or larger standard deviations. As with ordinary actions, we can also use expressions like

```
(define-action (sonar-response! z)
  weight (* weight (GAUSSIAN z h (if raining .9 .4))))
```

to say that inaccuracies in the sensing may depend on other considerations, in this case, whether or not the `raining` fluent is true at the time the sonar is used.

11.3.4 How uncertainty increases and decreases

Suppose the mean value of the `h` fluent happens to be 7.0 at some point. If an action like `(sonar-response! 7.001)` occurs, the mean value of `h` would stay about the same, but the variance (that is, the degree of uncertainty about the value) would decrease. We become more confident that `h` is close to 7.0. If `(sonar-response! 6.7)` occurs, then the variance would still decrease, but now the mean would also shift to a lower value. How close we get to 6.7 as the new mean depends on how strongly we believed it to be close to 7.0 before (that is, the variance on `h` before the sensing) against how strongly we believed in the accuracy of the sensor (that is, the variance specified in the sensing expression).

Just as sensing reports like `sonar-response!` decrease our uncertainty about the values of fluents, ordinary actions like `move!` increase our uncertainty. For example, if the mean value of `h` happens to be 7.0 at some point and we perform the two actions, `(move! -1)` followed by `(move! 1)`, the mean value of `h` would remain close to 7.0, but the variance would now increase. Even if we were quite confident that the value of `h` was close to 7.0 before, we would would be less confident of its value after the two actions. If we think of a Gaussian distribution of values of `h` as a bell-shaped curve with the mean at its peak, the effect of sensing actions like `sonar-response!` is to sharpen the curve, making the peak relatively higher, while the effect of ordinary actions like `move!` is to flatten the curve, making the peak relatively lower.

11.3.5 An online knowledge-based program

Having looked at the BAT, let us now examine the *ERGO* program for the robot in its entirety, which appears in Figure 11.4. The `get-close` procedure looks like a standard *ERGO* procedure with all the usual programming constructs, but with one major difference: the program never uses the fluents themselves as values except as part of a `sample-mean`, `sample-variance`, or `belief` expressions. This reflects the fact that the fluent values are not known, and so an *ERGO* program must rely on calculations over possible values of the fluents and their weights. For this reason, we call it a *knowledge-based program*.

The `get-close` procedure says that to move the robot close to x units away from the wall, we do the following in a loop: first use `sonar-request!` repeatedly to reduce the uncertainty about the current position `h`; then get the estimated distance d from `h` to x (that is, get the difference between the current estimate of `h` and x); and finally, either quit when d is small enough or use `move!` to move approximately d units.

Figure 11.4: Program file `Examples/sonar-robot.scm`

```
;;;  This is program for a robot with incomplete knowledge that starts out
;;;  somewhere between 2 and 12 units away from the wall, and uses a noisy
;;;  sensor and noise effector to move to close to 5 units away from the wall

(define-states ((i 1000000))
  h (UNIFORM-GEN 2.0 12.0))                              ; distance to wall

(define-action (move! x)                                 ; advance x units
  h (max 0 (- h (* x (GAUSSIAN-GEN 1.0 .2)))))

(define-action sonar-request!)                           ; sonar endogenous
(define-action (sonar-response! z)                       ; sonar exogenous
  weight (* weight (GAUSSIAN z h .4)))

(define (get-close x)
  (:let loop ()
    (:while (> (sample-variance h) .02)                ; too much uncertainty?
       (:begin (:act sonar-request!) (:wait)))         ; get sonar data
    (:let ((d (- (sample-mean h) x)))                  ; get distance to x
       (:when (> (abs d) .05)                          ; |d| is still large?
          (:act (move! d))                             ; move the robot
          (loop)))))                                   ; repeat

;; program interacts with an external robot manager on port 8123
(define (main)
  (let ((ports (open-tcp-client 8123)))
    (define-interface 'in (lambda () (read (car ports))))
    (define-interface 'out (lambda (act) (displayln act (cadr ports)))))
  (ergo-do #:mode 'online (get-close 5)))
```

A procedure like this would normally be run online (as discussed in Chapter 6) interacting with the manager of a physical robot. So *ERGO* will need an external interface that can receive endogenous actions (`move!` and `sonar-request!`) and return exogenous actions (`sonar-response!`). The program here assumes that there is a robot manager of some sort already running and listening on port 8123. (A simulation of such a manager appears in the file `Servers/sonar-robot-server.scm`.)

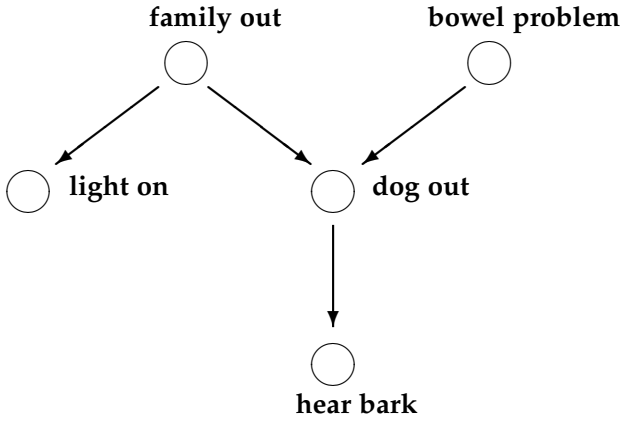
11.4 Bayesian networks

To conclude this chapter, we consider the notion of independent and dependent fluents. The fluents in a BAT with incomplete knowledge are defined using the function `define-states` in an expression like

`(define-states ((`v_1 d_1`) ... (`v_k d_k`))` f_1 e_1 ... f_n e_n`)`

where the e_i are fexprs which may or may not involve random elements. In many cases, these fexprs can be evaluated in any order because they are independent of each other. For example, in the example used above

Figure 11.5: A Bayesian network example




```
(define-states ((i 1000000))
  h    (UNIFORM-GEN 2.0 12.0)
  q    7
  temp (DISCRETE-GEN 'hot .2 'cold .6 'tepid .2))
```

the value of the `temp` fluent was independent of the value of the `h` fluent. What this means more precisely is that if we generate enough sample states, we expect half of them to have `(< h 7)` true, say, and among the states where `(eq? temp 'cold)` is true, we expect half of those to have `(< h 7)` true as well. This is what we mean when we say the two fluent values are independent.

But in some cases, there will be dependencies among the fluents, where the proportion of states where one fluent has a value is not preserved when we consider just those states where other fluents have certain values. For this reason, when we have a `define-states` expression, *ERGO* evaluates the e_i in *sequential* order so that the value given to f_i can depend on the values given to earlier f_j. (We already used this property in the `define-states` expression of Section 11.3.1 that had the explicit `weight` value.) These dependencies define what is called a *Bayesian network* (or a *belief network*) of fluents: the nodes are the fluents, and there is an edge from fluent f_j to f_i if f_j appears in e_i, in other words, if the value of f_i depends on the value of f_j.

Figure 11.5 shows an example (from the literature) of a Bayesian network with dependent and independent fluents (all of which happen to have binary values): a family may or may not be in the house, the family dog may or may not have bowel problems, the external light on the house may or may not be on, the dog may or may not be placed outdoors, and we may or may not hear barking. So, for example, the proportion of states where the dog is out depends on whether or not the family is out, but the proportion of states where the dog has a bowel problem does not depend on whether the family is out. Similarly, the proportion of states where barking can be heard depend on whether or not the dog is out, and so it depends indirectly on whether or not the family is out.

Here are the numbers for this Bayesian network example, showing the dependencies:

```
(define-state ((i 1000000))
  family-out    (BINARY-GEN .15)
  bowel-problem (BINARY-GEN .01)
  light-on      (if family-out (BINARY-GEN .6) (BINARY-GEN .05))
  dog-out       (if family-out
                  (if bowel-problem (BINARY-GEN .99) (BINARY-GEN .9))
                  (if bowel-problem (BINARY-GEN .97) (BINARY-GEN .3)))
  hear-bark     (if dog-out (BINARY-GEN .7) (BINARY-GEN .01)))
```

In about 15% of the states, the fluent `family-out` will be true, and in about 1% of the states, the fluent `bowel-problem` will be true. These two fluents are independent. But for `light-on`, the fluent will be true in about 60% of those states where `family-out` is true, but only in about 5% of those states where `family-out` is false. So `light-on` depends on `family-out`. Similarly `dog-out` depends on both `family-out` and on `bowel-problem`, and `hear-bark` depends on `dog-out`.

The *ERGO* `belief` function can be used to estimate the degree of belief of any formula involving the fluents of a Bayesian network. (The accuracy of the estimate depends on the number of samples used.) For the above network, the degree of belief that the family is out given that the light is on but we don't hear barking can be estimated as follows:

```
> (belief family-out (and light-on (not hear-bark)))
0.5011908149967795
```

(The theoretically correct value here is .5).

In more general terms, it is worth noting that when we have a collection of fluents f_i that are independent, the degree of belief that $f_1 = x_1$ and $f_2 = x_2$ and ... and $f_n = x_n$ will always be the product of the degrees of belief that $f_i = x_i$. So, for example:

```
> (belief (and (not family-out) bowel-problem))
.008394
```

(This is about $.85 \times .01$.) But with dependencies, such as in a Bayesian network, this is no longer true. The degree of belief in the conjunction will end up being the product of the degrees of *conditional* belief that $f_i = x_i$ given that its dependent fluents f_j have their values x_j. This can be seen in this example:

```
> (belief light-on (not (family-out)))
0.050516094862590476
> (belief light-on)
0.133473
> (belief (and (not family-out) light-on))
.04291
```

(This is about $.85 \times .05$ and not $.85 \times .13$.) In this case, the degree of belief in the conjunction is not simply the product of the degrees of belief in the two conjuncts.

11.4.1 The dynamic case

Turning now to actions, if we have a sensing action that tells us something about the value of some fluent(s), we have to adjust the weight of all states as before. But this

adjustment can depend on the values of other fluents. For example, as noted above, a sonar value might depend on the value of the `raining` fluent. In this case, once the weights are adjusted, it need not be the case that two independent fluents remain independent. (Roughly, the mean and variance of `h` across all states may no longer agree with the mean and variance of `h` across those states where `raining` is true.)

With ordinary actions, the situation is more complex. We may have had a dependency between `light-on` and `family-out` precisely because we thought there was some sort of *causal* relation between the two fluents: actions that causes the `family-out` fluent to change also cause the `light-on` fluent to change as well. For actions that are completely accurate, we can model this causal connection by arranging that the actions that change one fluent also change the other. But for actions that (for one reason or another) may or may not change the first fluent, we cannot use `define-action` as before.

For example, imagine an action `maybe-go-out!` that changes the fluent `family-out` to true, but only 80% of the time (for one reason or another). We might want to say that if the value is changed, then there is a 60% chance that the light will be turned on after the action. If we were to use

```
(define-action maybe-go-out!
  family-out (BINARY .8)
  light-on   (if family-out (BINARY-GEN .6) light-on))
```

then we would be saying that the new value of `light-on` depends on the *previous* value of `family-out` (which is the way `define-action` has been used until now). To say that the value of `light-on` depends on the *new* value of `family-out`, *ERGO* allows an optional `#:sequential?` keyword argument in a `define-action` expression:

```
(define-action maybe-go-out! #:sequential? #t
  family-out (BINARY .8)
  light-on   (if family-out (BINARY-GEN .6) light-on))
```

This ensures that the e_i are evaluated in *sequential* order, so that the new value given to an f_i can depend on the new values given to earlier f_j, preserving the dependence between the two fluents.

11.5 Bibliographic notes

This chapter presents an attempt to apply ideas from probability theory to the programming of cognitive robotics. (See the Chapter 12 notes for dealing with incomplete knowledge without numeric information.) Probability theory itself is a well studied topics with many textbooks, such as [8, 16, 25]. It can be interpreted as being about the outcome of random events (such as rolling a dice or buying a lottery ticket) or about uncertainty more generally (such as whether a door is open or where a box is located), as considered here. For the more philosophical foundations of probability, see [11] and [27]. For an analysis of uncertainty that includes probability as well as other notions, see [19].

One of the main obstacles in using probability theory in a computational setting involves the very large number of possible states that would need to be considered in any realistic application. For n binary fluents, there would need to be 2^n states, each of which

might have a different weight. (In the case of real-valued fluents, the situation is much worse: there would be uncountably many possible values for the fluents, so the summation of weights must be reformulated as the integration of densities.) Judea Pearl was perhaps the first to tackle this issue in [35]. He is credited with the proposal for Bayesian networks [10, 26], which is one important way of making the specification of all the possible states and their weights more manageable. However, reasoning with a Bayesian network directly appears to be difficult [9], which is why an approximate method based on sampling (what Pearl called "stochastic simulation") is often used, as it was here. The example Bayesian network used in this chapter is taken from [7].

The application of probability to various facets of (non-cognitive) robotics has a long history, especially for sensing, localization, path planning, and learning. An excellent textbook on the subject is [45]. The application of probability to the programming of cognitive robots, where an agent would still need to reason about the prequisites and effects of actions, for instance, is somewhat newer. One proposal based on the Golog language (discussed in Chapter 10) is [5], but other variants exist. The probabilistic version of *ERGO* presented in this chapter is indeed one such variant, and draws heavily from [1] and [3], with the sonar example used in this chapter taken from [3]. For other ways of looking at probabilistic programming in general, see [18] and [38].

* Chapter 12

Generalized Planning

In Chapter 11, we explored the idea of a cognitive robot with incomplete knowledge, with an emphasis on real-valued numerical fluents and on the uncertainty that arises due to noise and inaccuracies in a robotic system. In this chapter, we continue the exploration of incomplete knowledge but without this focus on numbers.

The starting point of this chapter will be the representation of incomplete knowledge presented in Section 11.1. To recap very briefly, instead of using `define-fluents` to create a single state mapping fluents to their known values, the function `define-states` is used to create a list of possible states, each of which maps fluents to values, and each of which might be a correct representation of the world.

In this chapter, we will deal with fluents that only have a small number of possible values (so sampling will not be necessary) and where numerical inaccuracies in the sensors and effectors are not the issue. In contrast to the online programming of Chapter 11, we will consider how a cognitive robot can use offline planning in such a context.

12.1 Conformant planning

Perhaps the simplest sort of reasoning about what to do in the case of incomplete knowledge is what is called conformant planning. Like the basic planning seen in Chapter 3, conformant planning involves finding a sequence of actions that achieves a goal of interest. The difference is that the robot needs to *know* that the sequence is a correct plan: each action can be performed legally in turn, and the goal is satisfied in the state that results from performing them. In other words, in conformant planning, we seek a single sequence of actions that is a solution to the basic planning problem for every possible initial state.

Let us return to the BAT of Figure 11.2. Suppose that it is known that there are four objects in stack A, but that their colours are unknown. This can be represented as follows:

```
(define cols '(red blue))
(define-states ((p cols) (q cols) (r cols) (s cols))
  hand   'empty
  stacks (hasheq 'A `(,p ,q ,r ,s) 'B '() 'C '()))
```

(So there are a total of sixteen states in the list of initial states.) Consider the goal defined by the following function:

```
(define (all-on-C?)
  (and (eq? hand 'empty) (null? (stack 'A)) (null?  (stack 'B))))
```

A conformant plan that solves this goal is the following:

```
((pick! A) (put! C) (pick! A) (put! C)
 (pick! A) (put! C) (pick! A) (put! C))
```

In each of the sixteen states, the actions in the sequence can be legally performed and in each case, the goal of having all the objects in stack C will be satisfied in the final state.

To generate a conformant plan like this, the `ergo-simplan` function can be used exactly as before. What the implementation does is to search for a plan that works for one of the initial states, but before accepting it, the procedure confirms that the plan also works for the remaining states. (Other implementation strategies are certainly possible.)

12.2 Non-sequential planning

Let us now turn to goals that are outside the reach of conformant planning. Consider the goal defined in Figure 11.2 of making two towers of blocks, a red one on stack B and a blue one on stack C. Clearly no fixed sequence of actions will achieve this goal for all the sixteen initial states above. After picking up a block, the `hand` fluent will have the value `red` in some states and `blue` in others. No matter what action is specified next in the sequence, it will be wrong for some initial states. So conformant planning cannot succeed here.

This is where sensing enters the picture. In a world of red and blue objects like this, it would seem reasonable for a robot to have a sensor of some sort that would allow it to determine the colour of the object it is holding. With a sensor like this, a plan to make the two stacks of blocks is now clear. The robot should do the following four times:

- it should pick up a block from stack A;
- it should then sense the colour of the block it is holding;
  - if the sensing result happens to be `red`, it should place the block in stack B;
  - if the sensing result happens to be `blue`, it should place the block in stack C.

In a plan like this, we are obviously going beyond a mere sequence of actions. (There is a form of branching here, and later we will see that there can be loops.) Nonetheless, this non-sequential plan has two desirable properties: first, it does the job properly in all sixteen initial states, that is, in each initial state, the actions recommended by the plan can be legally executed and will result in final states where the goal condition is true; second, like a simple sequence of actions, the plan can be executed by a robot manager independently of *ERGO*, that is, without any information about the current BAT, the state of the world, or the goals under consideration.

This idea will be made more precise below, and we will show how such plans can be generated automatically in *ERGO*. But first, we must examine in more detail the sort of sensing being considered to support this form of planning.

12.3 Using sensing information offline

In earlier chapters, sensing information was used to update the values of certain changing fluents (like the temperature in the elevator, in Chapter 6). In the case of incomplete knowledge, however, sensing information is not used to track a changing world, but to revise the state of knowledge of the robot. (In Chapter 11, it was the state of *belief* that was revised.) The robot begins by not knowing the colour of the first block in stack A, but then picks it up, does the sensing, and then comes to know what that colour is. We can use this idea in an offline setting: we can consider the range of sensing results we might obtain, calculate how each of them would change the state of knowledge, and then plan appropriately for each contingency.

12.3.1 The `#:sensing` keyword

To specify the range of possible sensing results of an action, a special keyword `#:sensing`, analogous to `#:prereq`, is used in the definition:

`(define-action` *action-name*
  `#:prereq` *fexpr*
  `#:sensing` *fexpr*
  fluent$_1$ *fexpr*$_1$
  ...
  fluent$_n$ *fexpr*$_n$`)`

The idea is that whenever this action is performed, the available sensors will eventually report a value for the sensing result. The range of results to be expected are the possible values of the given `#:sensing` fexpr. For example, a sensing action `sense-colour!` that tells the robot the colour of the object in its hand can be defined as follows:

```
(define-action sense-colour!
   #:prereq   (not (eq? hand 'empty))
   #:sensing  hand)
```

After performing this `sense-colour!` action, no fluents change, but the sensors will eventually tell the robot some value (`red` or `blue`), which is to be understood as the value of the `hand` fluent, which can then be used to update the state of knowledge.

A sensing action is not limited to making known the value of a given fluent only. For example, imagine a weaker colour sensor that can only detect if what is being held is red or not. This can be formalized using an action like `sense-red!` defined as follows:

```
(define-action sense-red!
   #:prereq   (not (eq? hand 'empty))
   #:sensing  (eq? hand 'red) )
```

In this case, the sensor is assumed to provide a Boolean value, `#t` or `#f`. The robot can then eliminate those states for which the fexpr `(eq? hand 'red)` has a different value, and thus come to know whether or not the object it is holding is red. If there are only red and blue objects in the world, this is the same as `sense-colour!`. But in a world of red, blue, and

green objects, the robot would not be able to tell the difference between a green and a blue object in its hand since the sensing result would be `#f` in both cases.

A slightly different way of handling the colour information in this world is to imagine that the sensing information is relayed any time an object is picked up:

```
;; action to pick up an object, while sensing its colour
(define-action (pick! st)
   #:prereq   (and (eq? hand 'empty) (not (null? (stack st))))
   #:sensing  (car (stack st))
   hand       (car (stack st))
   stacks     (hash-set stacks st (cdr (stack st)))  )
```

In this case, the `pick!` action is assumed to change the world as before (moving an object from a stack to the robot's hand), and simultaneously, to provide `red` or `blue` as the value for the sensing result. It is important to note that after doing this `pick!` action, the robot comes to know what the first element of the stack *was* (just prior to the `pick!` action) and not what the first element of the stack *will be* (after the `pick!` action). The elimination of states (or acquisition of knowledge) is the same as for the `sense-colour!` action.

So far, this colour sensing information will not allow a robot to deal with the different numbers of objects it may need to deal with on the stacks. For this, one might suppose that the robot has a second sensor that will tell it if there are any objects left on a stack:

```
(define-action (sense-empty! st)
   #:prereq   (eq? hand 'empty)
   #:sensing  (if (null? (stack st)) 'empty 'non-empty))
```

This sensing action is assumed to tell the robot `empty` or `non-empty` according to whether or not there are objects on the given stack, which would again allow it to eliminate world states from its list of possible states appropriately.

12.3.2 Actions as attempts

There is, however, a more compact and convenient way of formulating all this sensing information. The action `pick!` can be reinterpreted as an *attempt* to pick up a block from a stack. Intuitively, an attempt is an action that is somehow less demanding to perform than the corresponding successful action. For example, a person can always perform the action of *attempting to lift* a big weight; that same person may or may not be able to perform the action of *actually lifting* the weight. For the `pick!` action, the robot can perform an attempt to pick up an object from a stack without actually getting one. So instead of requiring as a prerequisite that the stack be non-empty, we can allow the action to be legally applied to an empty stack, but give it no effects in this case, and have it return a special sensing result telling the robot that the stack was empty. A version of `pick!` defined in this way is shown in the BAT of Figure 12.1. (It may be conceptually clearer to give this action a name like `try-to-pick!`.) The sensing result returned by `pick!` will be `red`, `blue`, or, in the case of an an empty stack, the symbol `fail`. Note that this version of `pick!` still requires the hand to be empty.

Figure 12.1: Program file `Examples/PlanningExamples/red-blue-bat.scm`

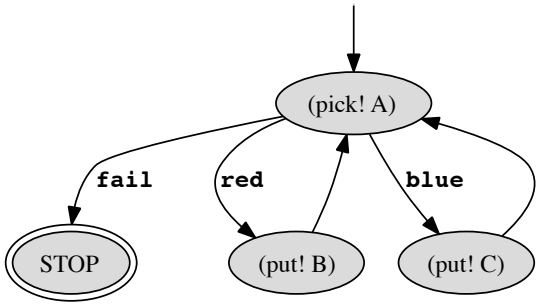
```
;;;  A simple robot world involving objects on three stacks, A, B, and C.
;;;  There are two fluents
;;;      hand    the object held in the robot's hand or 'empty'
;;;      stacks  the contents of each stack as a list of objects
;;;  There are two actions:
;;;      pick!   attempt to pick up the first object from a stack
;;;      put!    push an object onto a stack

;; use stackA-values defined elsewhere to get the values for stack A
(define-states ((ini stackA-values))
    hand   'empty
    stacks (hasheq 'A ini 'B '() 'C '()))

;; abbreviation to get the content of a stack A, B, or C
(define (stack st) (hash-ref stacks st))

;; action to pick up an object and return either its colour or 'fail'
(define-action (pick! st)
   #:prereq   (eq? hand 'empty)
   #:sensing  (if (null? (stack st)) 'fail (car (stack st)))
   hand       (if (null? (stack st)) 'empty (car (stack st)))
   stacks     (if (null? (stack st)) stacks
                  (hash-set* stacks st (cdr (stack st)))) )

;; action to push what is being held to the top of a stack
(define-action (put! st)
   #:prereq   (not (eq? hand 'empty))
   hand       'empty
   stacks     (hash-set* stacks st (cons hand (stack st)))  )
```

12.4 FSA plans

With this notion of sensing, and the `pick!` action defined as in the BAT of Figure 12.1, we can now be more precise about how to solve the two tower problem, that is, the kind of plan we have in mind that achieves the goal of getting all the red blocks of stack A onto stack B and all the blue ones onto stack C. The plan we are looking for is displayed as a graph in Figure 12.2. A plan of this sort is called an *FSA plan* (or finite-state-automata plan). It is made up of *plan states* (drawn as nodes in the graph) with *transitions* between them (drawn as possibly labelled edges). The *starting state* is a special state indicated by a single edge with no source. There are also *final states*, which have the label `STOP` and no outgoing edges. Each non-final state is labelled with a primitive action from the BAT, and must have either a single outgoing edge or outgoing edges labelled with the possible sensing results for that action. In the BAT above, only the `pick!` action has sensing results, and the three possible values are `fail`, `red`, and `blue`.

Although an FSA plan like this is obviously more than a sequence of actions, a robot manager (of the sort discussed in Chapter 6) that can perform the actions of the BAT should still be able to execute such a plan, without needing any further information or interaction with *ERGO*. Here is what a robot manager needs to do:

Figure 12.2: A plan to make two coloured towers



1. First, let the current plan state be the start state.
2. If the current state is a final state of the plan, then terminate the execution.
3. Otherwise get the robot to perform the action associated with the current state, and obtain any sensing report from the sensors.
4. Make a transition from the current state to the next state of the plan according to the sensing report obtained at step 3, and then go to step 2.

This is a completely offline use of *ERGO*. That is, the robot manager is not being asked to send the sensing results of the actions to *ERGO* for consideration. Instead the manager uses those sensing results itself to traverse the plan it was given.

Note that any fixed sequence of actions (as produced by `ergo-simplan`, and the offline mode of `ergo-do`) can be trivially encoded as an FSA plan: put an edge to the first action in the sequence, put unlabelled edges from each action in the sequence to the next one, and finally, put an unlabelled edge from the last action in the sequence to a final stop state.

12.4.1 When is an FSA plan correct?

To explain what it means for an FSA plan to be a solution to a planning problem, the notion of a trace is needed. A *trace* of an FSA plan with respect to a single initial state of the world is the sequence of actions that result from executing the plan in that state. In other words, a trace is a complete path through the FSA plan, from start to end, where the transitions are as determined by the plan and the sensing results obtained.

For example, consider the FSA plan of Figure 12.2. In a world state where stack A is empty, the trace is the single action `(pick! A)`. In a world state where the stack A is `(red red blue)`, the trace is the following sequence:

```
(pick! A) (put! B) (pick! A) (put! B) (pick! A) (put! C) (pick! A).
```

A trace is considered undefined when there is an action that cannot be performed legally or when there is a sensing result for which there is no edge in the plan. An FSA plan is then considered to solve a planning problem if for each initial state of the world given by the BAT, the trace of the plan solves the planning problem in that state.

This definition of plan correctness reduces it to the correctness of a sequence of actions in a single initial state (as in Chapter 3). But it is sometimes also useful to take a different perspective and consider planning not in terms of individual world states, but in terms of the states of knowledge. For an FSA plan to solve a planning problem, the following conditions must hold:

- at the final state of the plan, the given goal condition must be known to be satisfied, according to the current state of knowledge.
- at a non-final state, the prerequisite of the action in the plan must be known to be satisfied, according to the current state of knowledge;
- at a non-final state, each possible sensing results of the action in the plan, according to the current state of knowledge, must be represented by a transition in the plan;

Implicit in this new specification is the idea that the planner will be in various states of knowledge at various states of the plan. For example, at the start, the state of knowledge is as given by the BAT. (This implies that the prerequisite of the first action in the plan must be known to be true initially.) Then, after performing an action and obtaining a sensing result, the state of knowledge evolves: world states are eliminated if they conflict with the sensing result, as described in Section 12.3.1. At the end, the final goal condition must be known to be true.

12.4.2 Generating FSA plans

It is possible to generate FSA plans automatically using the `ergo-genplan` function:

`(ergo-genplan` *goal actions* `[#:prune` *prune*`] [#:loop?` *flag*`] ...)`

The arguments here are similar to those for conformant planning (and simple planning), but with a few additional optional arguments described later. The value returned by `ergo-genplan` will be an FSA plan (or `#f`, if it cannot find a plan) that contains the given actions and that solves the problem in the above sense.

A *ERGO* program that uses `ergo-genplan` to produce the FSA plan of Figure 12.2 is shown in Figure 12.3. When run, it will return the following value:

```
'((3 1 (put! C) #t)   (1 3 (pick! A) blue)   (2 1 (put! B) #t)
  (1 2 (pick! A) red) (1 0 (pick! A) fail))
```

This an FSA plan in the form of a list of transitions $(q\ q'\ a\ r)$ where q and q' are plan states, a is an action, and r is either a sensing result or `#t` indicating none. Plan states here are numbered, with 0 always indicating the final plan state, and 1 always indicating the start plan state. If `ergo-genplan` is called with the `#:draw` optional argument set to true (its default value), it will also produce a file called `tmp.dot` which can be used with the freely available `dot` utility to draw an FSA plan as a graph like the one shown in Figure 12.2.

It is not too hard to see that the generated plan is correct not only for the given four initial states of the BAT, but for other initial states as well. In particular, the plan generated by `ergo-genplan` works no matter how many blocks there are initially in stack A, and no matter how many of them have unknown colours. In other words, this plan happens to achieve the desired goal for *any* initial state where there are red and blue objects on stack A. (We return to this issue in Section 12.4.6.)

Figure 12.3: Program file `Examples/PlanningExamples/two-towers.scm`

```
;;;  This is a program for the Two Towers problem

;; possible initial values for stack A
(define stackA-values '(() (red) (blue) (red red blue red)))

;; the BAT
(include "red-blue-bat.scm")

;; make a tower of reds on stack B and a tower of blues on stack C
(define (goal?)
   (and (eq? hand 'empty) (null? (stack 'A))
        (for/and ((o (stack 'B))) (eq? o 'red))
        (for/and ((o (stack 'C))) (eq? o 'blue))))

(define (main)
  (ergo-genplan goal? (append (map pick! '(A B C)) (map put! '(A B C)))))
```

12.4.3 The odd bar problem

Let us now consider a very different planning problem:

> There are twelve gold bars that have the same weight, except for one of them that is either heavier or lighter than the others. There is a balance scale that can compare the weight of any two collections of bars: it will indicate whether the collection on the left is heavier, or the one on the right is heavier, or whether the two collections have the same weight. The goal is to determine which bar is the odd one, whether it is heavier or lighter than the others, and to do so using the balance scale only three times.

With twelve bars and three weighing actions, this is a very challenging problem! (What makes the problem so hard is that there are an extremely large number of possible weighing actions to consider, and the smallest solution to the problem is a plan with thirty seven states in it. We will see how to deal this in Section 12.5.) But a variant of the problem where there are only three gold bars and where the balance scale can only be used twice is much easier. A representation of this three-bar variant is shown in Figures 12.4 and 12.5.

In this version, the fluents `odd-bar` and `odd-weight` represent the (unknown) gold bar and its (unknown) weight, heavy or light. (So for three bars, there will be six initial states.) The fluent `tries` represents how many weighing actions remain: it starts at `allowed-weighs`, and is decremented each time a `weigh!` action is performed. The other action in this BAT is the `say!` action, which simply announces the odd bar and its weight, making the `announced` fluent true, which is the goal to be achieved.

The important part of the `weigh!` action is its sensing result: it returns `left`, `right`, or `even`, according to whether the scale goes down on the left, down on the right, or not all for the two lists of bars it is given as arguments. (For simplicity, the `weigh!` action is restricted to lists of bars of the same length.) If the sensing result is `left`, for example, this indicates that either the odd bar is heavy and appears in the first argument of the action, or the odd bar is light and appears in the second argument.

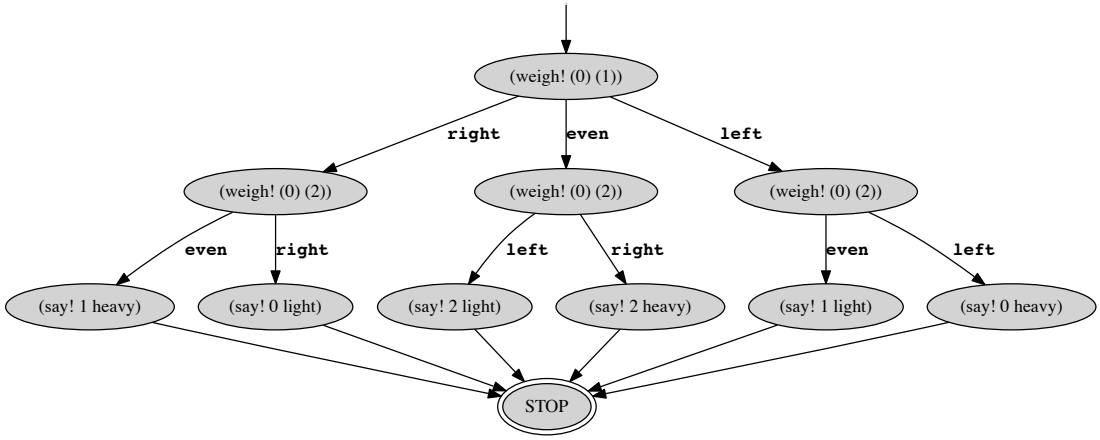
Figure 12.4: Program file Examples/PlanningExamples/bars-bat.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   This is the odd bar problem, where the number of bars and the number
;;;      of allowed weigh actions are defined elsewhere

;;;  Fluents:
;;;    odd-bar:  the bar that is odd
;;;    odd-weight: heavy or light for the odd bar
;;;    announced?:  true only after odd bar has been announced
;;;    tries:  numbers of weighing actions remaining
;;;  Actions:
;;;    (say! b w): announce that bar b is the odd one and of weight w
;;;    (weigh! l1 l2): compare weight of the bars in list l1 vs list l2

(define weights '(heavy light))

(define-states ((b bars) (w weights))      ; bars defined elsewhere
  odd-bar    b
  odd-weight w
  announced? #f
  tries      allowed-weighs)               ; allowed-weighs defined elsewhere

(define-action (say! b w)
  #:prereq   (and (eq? b odd-bar) (eq? w odd-weight))
  announced? #t)

(define-action (weigh! l1 l2)
  #:prereq   (and (> tries 0) (= (length l1) (length l2)))
  #:sensing  (cond
               ((memq odd-bar l1) (if (eq? odd-weight 'heavy) 'left 'right))
               ((memq odd-bar l2) (if (eq? odd-weight 'heavy) 'right 'left))
               (else 'even))
  tries      (- tries 1))

(define all-say-acts
  (for/append ((b bars)) (for/list ((w weights)) (say! b w))))
```

Figure 12.5: Program file Examples/PlanningExamples/3bars.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   This is the odd bar problem, with 3 bars and 2 weighings allowed.

(define bars 3)
(define allowed-weighs 2)

(include "bars-bat.scm")

(define (main)
  (ergo-genplan (lambda () announced?)
    (append all-say-acts
            '((weigh! (0) (1)) (weigh! (0) (2)) (weigh! (1) (2))) )))
```

Figure 12.6: A plan for the 3bars problem



Since the goal is to make the `announced` fluent true, and since this is what a `say!` action does, why is it then not sufficient to have a plan consisting of a single `say!` action? The answer is that this action can only be performed if its prerequisite is known to be true, which only happens when the `odd-bar` and `odd-weight` fluents have known values. To get to such a state, it will be necessary to first perform some `weigh!` actions.

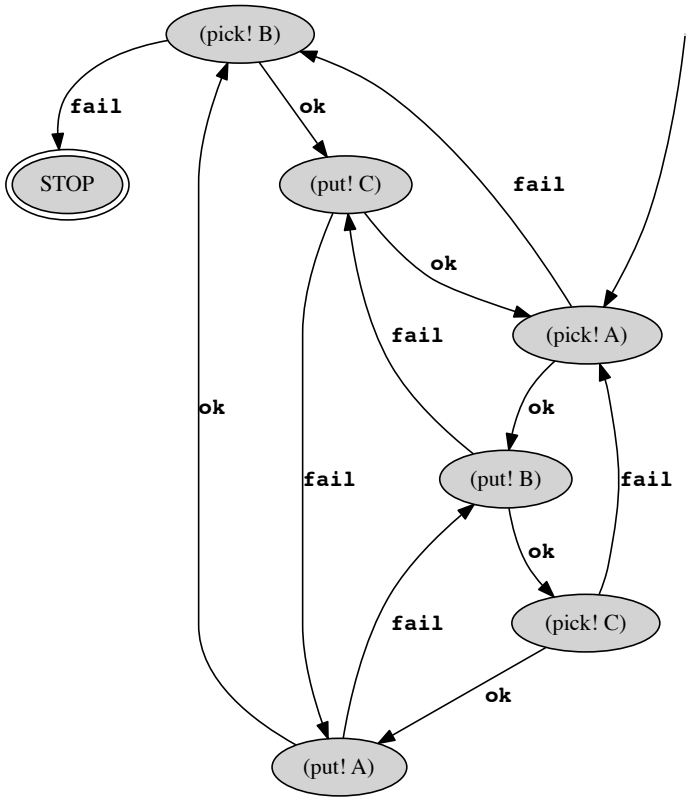
The FSA plan that the program finds is shown in Figure 12.6. It begins by weighing bars 0 and 1, an action whose prerequisite is indeed known to be true at the outset. There are then three possible sensing outcomes. Consider the one labeled `even`, and how the state of knowledge changes. Since an `even` outcome only happens in states where bar 2 is the odd bar, any initial state where this is not the case is eliminated, leaving only two of the original six states. Next, the plan is to weigh bars 0 and 2. In this case, the `even` outcome is not possible according to what is known. (It would require bar 2 to have the same weight as bar 0, which is now known to be false.) Consider the `left` outcome. Since bar 0 is not heavy, this outcome only happens when bar 2 is light. At this point, the prerequisite of a `say!` action is satisfied, and so the plan is to next announce that bar 2 is light, and then stop. All the other paths through the plan are analogous. There are six traces through the FSA plan, each involving two `weigh!` actions and the appropriate `say!` action.

12.4.4 The towers of Hanoi problem

Let us now go on to a more complex problem, a robotic version of the well-known towers of Hanoi. This is similar to the problem with red and blue blocks: there are three stacks A, B, C, each containing objects, now interpreted as disks of varying sizes. The robot can attempt to pick up a disk and will be told either `fail` as before or simply `ok` when the attempt is successful. The robot can also attempt to put the disk it is holding onto a stack, but this is also an attempt and it too can fail when the disk being held is larger than the top one on the stack. Again, the robot will be told `ok` or `fail`. The goal, as usual, is to move all the disks from stack A to stack C.

Note that unlike the way the problem is usually posed in computer science courses, in

Figure 12.7: A plan for the towers of Hanoi



this robotic version, the robot has no way of knowing initially how many disks there are, or even if the number of disks is even or odd. Nonetheless, there is an FSA plan that works for an initial state with any number of disks on stack A, shown in Figure 12.7.

Unlike the case with the red and blue blocks, the fact that this FSA plan does the job is far from obvious. Because the number of disks is unknown at the outset, the goal is solved in an unusual way. For instance, as can be seen from the figure, the first disk from stack A is always placed on stack B. This is the right thing to do when there are an even number of disks on stack A. But when there are an odd number of disks, what happens is that all the disks end up being moved to stack B first, and then they are all moved to stack C. Another unusual aspect of the problem is that the robot has no way of knowing which disk is the largest (at the bottom of stack A). Consequently, it may end up picking up that largest disk and putting it right back when there happen to be disks on the two other stacks.

A complete program for the towers of Hanoi problem appears in Figure 12.8. The BAT is a variant of the one in Figure 12.1, but where both the `pick!` and `put!` actions now have sensing results. Let us consider some of the additional optional arguments of `ergo-genplan`. The procedure works by iterative deepening, looking for plans with ever more plan states, up to some maximum specified by the optional `#:states` argument. (If the `#:deep?` argument is set to false, only plans of the size given by `#:states` are considered.) Because even a small plan can loop forever, a separate `#:steps` argument can

Figure 12.8: Program file `Examples/PlanningExamples/hanoi.scm`

```
;;;  This is a program for the robotic version of the Towers of Hanoi.

(define-states ((ini '((1) (1 2) (1 2 3) (1 2 3 4))))
   hand   'empty                              ; start holding nothing
   stacks (hasheq 'A ini 'B '() 'C '()))      ; all stacks empty

;; abbreviation to get the content of a stack A, B, or C
(define (stack st) (hash-ref stacks st))

;; the contents of hand cannot be placed on stack st
(define (cannot-put? st)
  (and (not (null? (stack st))) (> hand (car (stack st)))))

;; action to pick up an object and return either 'ok' or 'fail'
(define-action (pick! st)
   #:prereq   (eq? hand 'empty)
   #:sensing  (if (null? (stack st)) 'fail 'ok)
   hand       (if (null? (stack st)) 'empty (car (stack st)))
   stacks     (if (null? (stack st)) stacks
                  (hash-set stacks st (cdr (stack st)))) )

;; action to put what is being held on a stack and return 'ok' or 'fail'
(define-action (put! st)
   #:prereq   (not (eq? hand 'empty))
   #:sensing  (if (cannot-put? st) 'fail 'ok)
   hand       (if (cannot-put? st) hand 'empty)
   stacks     (if (cannot-put? st) stacks
                  (hash-set stacks st (cons hand (stack st))) ))

;; get all the disks onto stack C
(define (goal?) (and (eq? hand 'empty) (null? (stack 'A)) (null? (stack 'B))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  The main program

(define (main)
  (ergo-genplan goal? (append (map pick! '(A B C)) (map put! '(A B C)))
                #:steps 60))
```

be set to an upper bound of how many actions should be allowed in attempting to reach a goal state. (In the case of the towers of Hanoi, the default value of 30 steps is insufficient, and a higher value of 60 is specified to deal with the initial state having four disks.) The final optional argument to `ergo-genplan`, `#:loose?`, is considered in the next problem.

12.4.5 The striped tower problem

A final even more challenging problem is the striped tower problem shown in Figure 12.9. This problem is similar in its setup to the two towers problem with red and blue blocks. However, this time, an equal number of red and blue blocks are assumed to be on stack A in some order at the outset (generated using `even-blocks`), and the goal is to get them all onto stack C in a striped order (using `striped?`): blue at the top, then red, then blue, and

Figure 12.9: Program file `Examples/PlanningExamples/striped.scm`

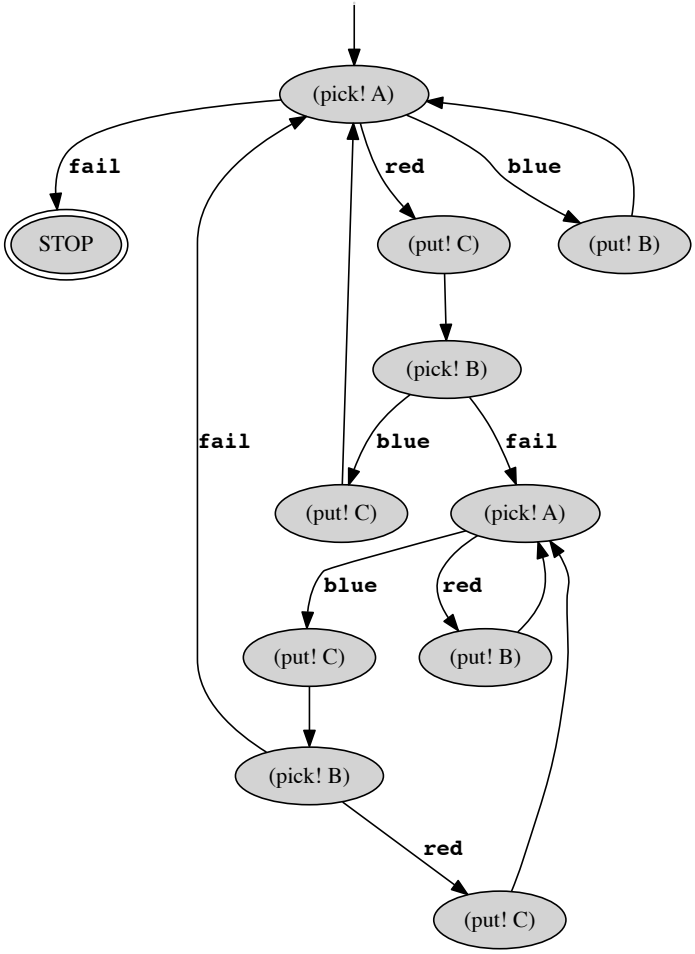
```
;;;  A robot world just like the Two Towers problem but with a harder goal.

;; all lists of length n made up of only red and blue elements
(define (all-blocks n)
  (if (= n 0) '(())
      (let ((x (all-blocks (- n 1))))
        (append (for/list ((e x)) (cons 'blue e))
                (for/list ((e x)) (cons 'red e))))))

;; all lists of length n with an even number of blue and red elements
(define (even-blocks n)
  (for/only ((x (all-blocks n)))
    (= (for/sum ((b x)) (if (eq? b 'blue) 1 0)) (/ n 2))))

;; stack A: up to 6 blocks with equal number of red and blue ones
(define stackA-values (for/append ((n 4)) (even-blocks (* n 2))))

(include "red-blue-bat.scm")    ; the BAT from the Two Towers problem

;; the goal to be achieved: a striped tower on stack C
(define (goal?)
  (and (eq? hand 'empty) (null? (stack 'A)) (null? (stack 'B))
       (striped? (stack 'C)) ))

;; the recursive definition of a striped tower with blue at the top
(define (striped? x)
  (or (null? x)
      (and (eq? (car x) 'blue) (eq? (cadr x) 'red) (striped? (cddr x)))))

;; find a plan that works for all states with up to 6 blocks
(define (main)
  (ergo-genplan #:states 10  #:loop? #t  #:loose? #f
          goal? (append (map pick! '(A B)) (map put! '(B C)))))
```

so on to a red one at the bottom of the stack. Finally, it is only permissible to pick objects up from stacks A and B, and to put objects down on stacks B and C.

What makes this problem challenging from a search point of view is that the smallest FSA plan that achieves the goal (for the given world states) has ten states: see Figure 12.10. The problem can be solved by `ergo-genplan` with iterative deepening, but even using the `#:loop?` argument to avoid visiting the same state twice, it takes much too long.

One of the reasons `ergo-genplan` can take a long time is that it begins by looking at the sensing outcome where the resulting knowledge is the *least* constrained. This is fine for quickly eliminating actions that in some cases fail to provide enough information. But for domains with many legal actions, it can be much better to look first at the sensing outcomes where the resulting knowledge is the *most* constrained (to help build up a plan with the specific knowledge). To do so, an optional `#:loose?` argument is used with `ergo-genplan`. The default value here is `#t`, but if `#f` is given, as it is in Figure 12.9, the most constrained sensing outcomes are considered first. With this, the program works much better, although it will still take minutes (but not hours) to find the desired plan.

Figure 12.10: A plan for the striped tower problem



12.4.6 How general are the plans?

In general, `ergo-genplan` looks for an FSA plan that works for a given list of initial states. For example, in the Hanoi program in Figure 12.8, it looks for a plan that achieves the goal when there are one, two, three, and four disks on stack A. It is not asked to find a plan that also works when there are five, ten, or twenty disks. So it is a mere *coincidence* that the FSA plans displayed in this chapter work more generally. If `ergo-genplan` were asked to find a plan for the towers of Hanoi that worked for one and two disks only, it would again come up with a plan, but likely one that would not work at all for larger numbers of disks.

Unfortunately, it is not at all obvious how to find a set of initial states that would allow `ergo-genplan` to produce a plan that works more generally. One strategy is to use a certain number of "small" initial states and a few "large" ones to force the plan to scale appropriately. Ideally, one would like to be able to *prove* (for certain BATs) that a plan that is found by `ergo-genplan` to work for a given collection of initial states will continue to work for all initial states. This remains a thorny research problem.

Figure 12.11: Program file `Examples/PlanningExamples/prog-two-towers.scm`

```
;;;  This is an offline program for the Two Towers problem

(include "two-towers.scm")

(define (control)
  (:begin (:act (pick! 'A))
      (:until (known? (eq? hand 'empty))
          (:if (known? (eq? hand 'red)) (:act (put! 'B)) (:act (put! 'C)))
          (:act (pick! 'A)))))

;; find a plan that works for all the initial states
(define (main) (ergo-gendo (control)))
```

12.5 Offline knowledge-based programs

Offline planning with incomplete knowledge as seen in `ergo-genplan` works well enough for problems with small search spaces. Unfortunately, it does not take much for a problem to have more than a million nodes to search (referring to the guideline of Section 2.5). Even the striped tower problem was near the limit of this form of planning. In Chapter 4, we argued that high-level *ERGO* programming was a convenient alternative to basic sequential planning in cases like these. In this section, we will consider how knowledge-based *ERGO* programming can serve as an alternative to generalized planning.

The idea is perhaps best illustrated by considering a simple program for the two towers problem, shown in Figure 12.11. What makes the program `control` knowledge-based is that, because there are multiple initial states, all the fluents must appear within the scope of a `known?` or `possible-values` expression. (In Chapter 11, the fluents would appear within a `sample-mean`, `sample-variance`, or `belief` expression.) The only other difference is that `ergo-gendo` is used instead of `ergo-do` and will return as value an FSA plan or `#f`.

How this `control` procedure works is as follows. It first performs the `pick!` action on stack A (where the prerequisite is known to be true). This action has a sensing result, and so the list of initial states is divided into three groups. (This is because in the plan that is being constructed, the current FSA state can have transitions to three different states.) In the first group where the sensing result was `fail`, the expression `(known? (eq? hand 'empty))` will be true and so the program terminates; in the second group where the sensing result was `red`, the expression `(known? (eq? hand 'empty))` will be false, but then the expression `(known? (eq? hand 'red))` will be true, and so the program will do a `put!` on stack B; in the final group, the program will similarly do a `put!` on stack C. In these last two cases, the program will then perform another `pick!` action, which will cause the list of states for that group to be divided into subgroups, and execution continues.

So the main observation with these knowledge-based programs is that they use actions with sensing results in a very specific way. They appear in the programs before testing what is or is not known, and end up dividing the execution into multiple paths, each of which can consider what would be known if that sensing result were obtained.

With this in mind, let us now consider a program for the striped tower problem, shown in Figure 12.12. The first observation is that this program will generate the FSA plan of Figure 12.10 not in minutes, but in milliseconds! This is because there is almost no

Figure 12.12: Program file `Examples/PlanningExamples/prog-striped.scm`

```
;;;  The striped tower problem as a generalized program

(include "striped.scm")

(define (opp-colour x) (if (eq? x 'red) 'blue 'red))

(define (get-colour col)
  (:begin (:act (pick! 'A))
          (:unless (known? (eq? hand 'empty))
            (:if (known? (not (eq? hand col)))
              (:begin (:act (put! 'B)) (get-colour col))
              (:begin (:act (put! 'C))
                      (:act (pick! 'B))
                      (:if (known? (eq? hand 'empty))
                         (get-colour (opp-colour col))
                         (:begin (:act (put! 'C)) (get-colour col))))))))

;; find a plan that works for all states with up to 6 blocks
(define (main) (ergo-gendo (get-colour 'red)))
```

searching involved in the solution. The `get-colour` procedure says exactly what should be done when trying to get the next block of a certain colour from stack A, including using stack B to store blocks of the wrong colour until they are needed.

Now let us turn to the original odd bar problem, with twelve bars and three possible weighings. As already noted, there were too many possible actions to consider at each step of the plan to blindly search through. For example, consider a first weighing action with three bars on each side of the scale. There will be $(12 \times 11 \times 10)/6$ choices for the bars on the left, and for each of those, $(9 \times 8 \times 7)/6$ choices on the right. But considering all these choices is a waste of time. If any of them work as the first action of a plan, then any of the others would also work as the first action of a plan. This is because, initially, all the bars are "equivalent": nothing special is known about any one bar that is not known about the others. For this reason, as our first weighing action, we only need to consider a single weighing action of k bars on each side, for $1 \leq k \leq 6$. Similarly, after doing a first weighing action, say with bars 0,1,2 on the left and bars 3,4,5 on the right, then for each sensing result, bars 0,1,2 will be equivalent, bars 3,4,5 will be equivalent, and the remaining bars will be equivalent. In general, at any given stage, we can partition the bars into equivalence classes and only consider weighing actions that use representatives from these classes.

A program that does just this is shown in Figure 12.13. All of the effort here is on limiting the weighing actions to be considered. Looking at the last few lines of the program, we can see that it says to do three actions taken from the value of `(weigh-acts)` followed by a `say!` action as before. The function `weigh-acts` gets the possible values for `(list odd-bar odd-weight)` according to what is currently known, partitions the bars into four equivalence classes, then selects k bars from these classes for the left and right sides, where $1 \leq k \leq 6$. Overall, the program runs to completion in a second or so, and produces an FSA plan with thirty seven states (and sixty transitions), too big to display here. (Note that the program discovers that the first weighing action of the three must

Figure 12.13: Program file `Examples/PlanningExamples/prog-12bars.scm`

```
;;;   This is the odd bar problem, with 12 bars and 3 weighings allowed.

(define bars 12)
(define allowed-weighs 3)

(include "bars-bat.scm")

;; partition the list of all bars into 4 groups: both, heavy, light, neither
(define (ppart odds)
  (let loop ((b 0) (both '()) (heavy '()) (light '()) (neither '()))
    (if (= b bars) (list both heavy light neither)
        (if (member (list b 'heavy) odds)
            (if (member (list b 'light) odds)
                (loop (+ b 1) (cons b both) heavy light neither)
                (loop (+ b 1) both (cons b heavy) light neither) )
            (if (member (list b 'light) odds)
                (loop (+ b 1) both heavy (cons b light) neither)
                (loop (+ b 1) both heavy light (cons b neither)) )))))

;; choose k bars for left part of weighing action then call rbars for right
(define (lbars k part arg rem)
  (if (null? (cdr part))
      (if (> k (length (car part))) '()
          (let ((left (append (take (car part) k) arg)))
            (rbars (length left) (cons (drop (car part) k) rem) '() left)))
      (for/append ((i (+ 1 (min k (length (car part))))))
        (lbars (- k i) (cdr part) (append (take (car part) i) arg)
               (cons (drop (car part) i) rem)))))

;; choose k bars for right part of weighing action then build the action
(define (rbars k part arg left)
  (if (null? (cdr part))
      (if (> k (length (car part))) '()
          (list (weigh! left (append (take (car part) k) arg))))
      (for/append ((i (+ 1 (min k (length (car part))))))
        (rbars (- k i) (cdr part) (append (take (car part) i) arg)
               left))))

;; the weighing actions representatives, in terms of possible values
(define (weigh-acts)
  (let ((pp (ppart (possible-values (list odd-bar odd-weight)))))
    (for/append ((k (quotient bars 2))) (lbars (+ k 1) pp '() '()))))

(define (main)
  (ergo-gendo
     (:begin (:for-all i allowed-weighs (:for-some a (weigh-acts) (:act a)))
             (:for-some a all-say-acts (:act a)))))
```

have exactly four bars on each side of the scale. Anything less or more will leave too many possibilities to sort out in the remaining steps.)

As a final point regarding these knowledge-based programs, it is worth noting that writing them is somewhat of a black art. This can be seen in the *ERGO* program that

Figure 12.14: Program file `Examples/PlanningExamples/prog-hanoi.scm`

```
;;;  This is an offline program for the robotic version of the Towers of Hanoi.

(include "hanoi.scm")

;; pick from stack x and then put to stack y.  Stop when Peg B is empty
(define (picker x y z)
  (:begin (:act (pick! x))
      (:if (known? (eq? hand 'empty))         ; the pick action failed
           (:unless (eq? x 'B) (picker y z x))
           (putter y z x))))

;; put to stack x and then pick from stack y
(define (putter x y z)
  (:begin (:act (put! x))
      (:if (known? (eq? hand 'empty))         ; the put action was successful
           (picker y z x)
           (putter y z x))))

(define (main) (ergo-gendo (picker 'A 'B 'C)))
```

solves the Towers of Hanoi problem, shown in Figure 12.14. It is far from obvious how to write such a program. It does not look at all like the FSA plan that it generates, shown in Figure 12.7. The program is recursive, for one thing, even though the FSA plan it produces is clearly not. But the program is not recursive in the way that it would be in the usual formulation of the problem, where the number of disks is known. Analyzing in what sense this program is correct for any number of disks remains an open problem.

12.6 Bibliographic notes

This is a chapter about incomplete knowledge and planning in the presence of incomplete knowledge. (The case where there is supplementary numerical information is considered in Chapter 11.) The main idea, first presented in Section 11.1 but used thoughout this chapter, involves thinking of incomplete knowledge in terms of a set of possible states any one of which might be the correct representation of the world. This draws from the philosophical work of Jaako Hintikka on knowledge in terms of possible worlds [21]. A thorough analysis of this concept of knowledge and many of its variants and extensions can be found in [15]. An application to the sort of logical knowledge bases mentioned in Chapter 10 is [29].

There are many approaches to automated planning in the presence of incomplete knowledge, especially once probabilistic information is included. The terminology is unfortunately not standardized, and one sees reference to planning with nondeterministic actions (since the outcome of an action might not be known) and to planning with partial observability (since only part of a state may be known by observation). The textbook [17] discusses some terminology and approaches.

The direction followed here, with an emphasis on finding a single non-sequential plan that solves an entire set of problem instances, is what has been called generalized planning [2, 43]. Earlier work on non-sequential planning like [37] concentrated on efficient ways

of generating plans with branches, but subsequent work like [4, 23, 44] focusses more on plans with loops. The approach here, with FSA plans and explicit sensing actions, is taken from [22] and [24], from which the examples in this chapter are drawn. The idea of solving a planning problem for a given set of initial states, and then seperately confirming that it also works for a larger perhaps infinite set of states was first suggested in [28]. The move from generalized planning to offline knowledge-based programming in the final section (like the move from planning to programming in Chapter 4) has not appeared anywhere.

End Matter

Final Thoughts

This was a book about programming cognitive robots, and explored a number of ideas on the topic: declarative specifications, planning, high-level programming, search and backtracking, game playing, offline and online execution, reactivity, incomplete knowledge. But all the examples considered in the book were *small*, no more than a hundred lines of code each. From a programming point of view, a good question to ask is this: how will the ideas presented in this book scale up as the cognitive robotic systems grow in size and complexity?

Scaling up

I think there are two answers to this question. In one sense, the systems will scale well. While the examples considered here involved tens of fluents, there should be no major problems dealing with hundreds or even thousands of them. By using arrays and hash-tables, it should be possible to deal with millions of individual changing values. Similarly, the examples in this book used tens of primitive actions, but hundreds or thousands of them should still work quite well. Since actions can have parameters, it should be possible to deal with millions of individual action instances. Regarding the sizes of *ERGO* programs, it should be possible using libraries to build systems made of many thousands of lines of code, although it will be necessary to manage the namespaces more carefully than was done here. (Racket Scheme has a comprehensive module facility, outside the scope of this book.) So from this point of view, the *ERGO* systems should scale quite comfortably.

But there is another sense in which *ERGO* systems will not scale well at all, and that is with respect to *combinatorics*. To take an extreme example, consider the sequential planning seen in Chapter 3. If we had a large BAT (as above) with a million primitive action instances say, then searching for a plan consisting of just two actions might already be too much. We cannot expect even the fastest of computers to be able to cycle comfortably through the 10^{12} possibilities. But this issue is not due to the size of the BAT. As we have seen, the problem is already there in a BAT with just ten actions. In this case, searching blindly for a plan with twelve steps would again mean looking at 10^{12} possibilities. In fact, even with just two actions in a BAT, we already have the problem, in that we cannot expect to search blindly for a plan of forty actions, again about 10^{12} possibilities. The problem here is not the size of the BAT at all. It is the fact that we need to consider the very large number of cases that can arise when trying to reason with the BAT for some purpose.

How to deal with this problem? As we have seen in this book, the solution is to avoid blind search whenever the combinatorics makes the cost too high. (Recall the Million-Billion Rule for *ERGO* searching on page 35.) The word "blind" in this context really

means "without knowledge of how to do any better." The idea is that a cognitive robot should be able to *use what it knows* to avoid working through so many possibilities. This was the primary reason we moved from planning to high-level programming, starting in Chapter 4.

Consider, for instance, the job of finding a path from one location to another, something that came up a few times in the book. In the simple case, with tens of locations say, nothing special is needed to search for a path. But even with hundreds of locations, these simple methods no longer work. What does work, however, is to structure the locations into hierarchic regions. Even though we might be dealing with billions of individual locations, we can structure the hierarchy so that there might only be tens of interconnected regions at each level of the hierarchy. This means that it will remain feasible to find paths from one region to another at the same level. So, for example, to find a path from a street address in Toronto to a street address in Montreal, we can first find a path on the major highways from Toronto to Montreal, ignoring lower street-level details completely.

So in a challenging setting, the idea is not to avoid search completely, but to make sure that the combinatorics are kept manageable. In many cases, there will be *expertise* about the problem domain (such as the idea of structuring a map hierarchically) that can guide a cognitive robot towards a solution. Of course, the *ERGO* system itself does not provide this expertise; it is something that needs to be programmed.

Incomplete knowledge again

So is this the final answer? Is it really just up to the programmer to make a cognitive robot work properly when the combinatorics are challenging? Unfortunately, there is more to it than that. Here's the problem. We want to program a cognitive robot to use what it knows to make good choices about how to behave. That seems clear enough. The problem is what to do when determining what is known is itself a problem.

Consider, for example, how we dealt with incomplete knowledge in Chapters 11 and 12. We represented this knowledge in terms of a list of possible states. In Chapter 11, we allowed that there could be infinitely many such states, but we assumed that there was enough information to weigh the various possibilities, and then used sampling as a way of dealing with that. In Chapter 12, we considered the case where nothing was known about the relative weights of the various possibilities. Either way, we assumed that it would be sufficient to make decisions about what to do by working through a list of possible states.

But is easy to see that this idea does not scale well. Imagine we have forty Boolean fluents where nothing is known about their initial values. We cannot run even the most basic knowledge-based *ERGO* programs in this setting because we would have to go through the 10^{12} initial states just to see if something was known to be true. Even the best knowledge-based program that an *ERGO* programmer can come up with will hit a brick wall if the only way to test for what is known is to cycle through all these possibilities.

In the presence of incomplete knowledge, the responsibility of the *ERGO* programmer is to provide effective knowledge-based programs. But to support this, *ERGO* systems need practical ways of determining what is and is not known. For cognitive robotics to scale well in cases where there is incomplete knowledge to contend with, we cannot rely on testing what is true in overly large numbers of possible states.

Knowledge representation and reasoning

Fortunately, there are some ideas about what to do. Consider again the case where there are forty Boolean fluents with unknown values. Typically, a cognitive robot will still know *something*, even if it does not know the values of any of those fluents. Let us assume that what is known about these fluents can be represented by a formula ϕ (as was suggested with the `#:known` keyword in Chapter 11). To determine if some other formula ψ is known to be true, instead of going through all the states where ϕ is true to see if ψ is also true, we take our job to be that of determining whether or not ϕ *logically entails* ψ. In other words, instead of pouring over a very large number of initial states, we look closely at ϕ itself and see if we can derive ψ from it.

This is of course what is done in the area of *knowledge representation and reasoning*, as noted in Chapter 10. We represent what is known using a formula ϕ (or a finite set of such formulas), that we call the *knowledge base*, and then try to reason directly with this knowledge base to see what is entailed. Of course, one way to do this is would be to construct the list of states that satisfy the ϕ and work from there. But this is precisely what we hope to avoid. Indeed, for forty Boolean fluents it might not even be feasible to construct a list of states in the first place, and for a hundred fluents (still a small BAT), making a list of all the states is totally out of the question.

What are the good options for reasoning in this sense, and will they scale up well enough to serve for cognitive robotics? That is the big question. For certain types of knowledge bases, the answer is definitely *yes*. One well known example is the so-called Horn case, where the knowledge base is a set of Horn clauses, that is, disjunctions of fluents or their negations where at most one fluent appears unnegated in each disjunction. For a knowledge base that happens to be in this form, it is quite feasible to deal not just with forty, but with thousands of fluents. But the techniques that have been proposed so far to deal with *arbitrary* knowledge bases without restriction, appear to be much too weak. There are some ideas about what to do, including using inference methods that are only approximately correct. Whether any of these ideas will lead to practical systems for cognitive robotics remains a question for the future.

Figures and Program Files

Scheme Functions Used

This is a list of the predefined Scheme functions presented in Section 2.4 that are used in the example programs in this book. Most of these are Racket Scheme primitives, but a few are defined in the files `System/misc.scm` and `System/arrays.scm`. (Macros and special forms are also included in this list.)

- Symbols:
  `eq? equal? symbol?`
- Numbers:
  `+ - * / < <= = > >= abs min modulo quotient random`
- Lists:
  `append assq caddr cadr car cddr cdr cons drop length list list-ref member memq null? remove remove* reverse take`
- Strings:
  `display displayln eprintf printf`
- Ports and channels:
  `channel-get channel-put eof-object? open-tcp-client open-tcp-server make-channel read`
- Functions:
  `and-map apply for-each lambda map`
- Boolean values:
  `and not or`
- Hash-tables, vectors, and arrays:
  `array-ref array-set* build-array hash-ref hash-set hash-set* hasheq vector vector-ref vector-set`
- For programming:
  `case define else error for for/and for/append for/list for/only for/or for/sum if include let` '(quote) `(quasiquote) ,(unquote)

Index of Ergo Keywords

ERGO on a Page (v1.5)

The *ERGO* File

An *ERGO* file for an application typically has three parts: a basic action theory, the definitions of some *ERGO* procedures (using `define` as usual), and an optional robotic interface. These three parts are described further below, and may be loaded from other files by using `include`. The *ERGO* file usually ends with a `main` function that calls `ergo-do` or one of the planning functions:

```
(define (main) (ergo-do [#:mode mode] pgm))
```

The *pgm* here is an expression that evaluates to an *ERGO* program. If the execution *mode* is specified, it should be one of `'offline`, `'first` (the default), or `'online`. (The robotic interface part is needed only when the mode is `'online`.)

Using Scheme

Within an *ERGO* file, *ERGO* can be intermingled with Scheme variables and function definitions that appear in the usual way. In what follows, we use "*fexpr*" to mean any Scheme expression where the fluents of the basic action theory (see `define-fluents` below) may appear as global variables.

Running *ERGO*

Once *ERGO* has been properly installed, it is possible to call the `main` function in an *ERGO* file called `my-ergo-app.scm` as follows:

```
> racket -l ergo -f my-ergo-app.scm -m
```

Basic Action Theory

A basic action theory has definitions for the fluents and actions.

Fluents

The fluents of a basic action theory are defined using one or more expressions in the file of the form

```
(define-fluents
    fluent  ini
      ...
    fluent  ini )
```

where each *fluent* is a symbol and each *ini* is a Scheme form that provides the value of the fluent in the initial state. This has the effect of defining the fluents as global variables that can then be used later in fexprs for actions and programs.

Any valid Scheme datum can be used as the value of a fluent, including lists, vectors, hash-tables, and functions.

Actions

Each action is defined by an expression of the following form:

```
(define-action action
    fluent  fexpr
      ...
    fluent  fexpr )
```

The fluents listed are those that are considered to be changed by the action. The value of the *fluent* after the action will be the value of the corresponding *fexpr* before the action. (All changes are considered to be done in parallel.)

The *action* in the definition can be a symbol or a list of symbols (*name var ... var*) for an action with parameters..

In addition, the *fluent* can be the special symbol `#:prereq` or `#:sensing`, in which case, the corresponding *fexpr* defines the prerequisite or sensing result of the action.

The `define-action` expression has the effect of defining the action itself as a global variable whose value is the action symbol (or a list of the action symbol and its arguments).

ERGO Programs

Each of the following expressions evaluates to an *ERGO* program:

`:nil`
: The program that always succeeds.

`:fail`
: The program that always fails.

(`:test` *fexpr*)
: Succeed or fail according to whether or not the current value of *fexpr* is true.

(`:act` *action*)
: Fail if the action has a prerequisite that evaluates to false, but succeed otherwise, and move to a new state where the fluents are updated as per its `define-action` (see above).

(`:begin` *pgm ... pgm*)
: Sequentially perform all of the programs.

(`:choose` *pgm ... pgm*)
: Nondeterministically perform one of the programs.

(`:if` *fexpr* pgm_1 pgm_2)
: Behave like pgm_1 if the current value of *fexpr* is true, but like pgm_2 otherwise.

(`:when` *fexpr pgm ... pgm*)
: Behave like (`:if` *fexpr* (`:begin` *pgm...pgm*) `:nil`)

(`:unless` *fexpr pgm ... pgm*)
: Behave like (`:when` (`not` *fexpr*) *pgm...pgm*).

(`:until` *fexpr pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) repeatedly until the value of *fexpr* becomes true.

(`:while` *fexpr pgm ... pgm*)
: Behave like (`:until` (`not` *fexpr*) *pgm...pgm*).

(`:star` *pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) repeatedly for some nondeterministically chosen number of times.

(`:for-all` *var list pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) repeatedly for all values of the variable *var* taken from the list *list*.

(`:for-some` *var list pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) for some value of *var* from the *list*, chosen nondeterministically.

(`:conc` *pgm ... pgm*)
: Concurrently perform all of the programs, nondeterministically interleaving any single steps.

(`:atomic` *pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) but with no interleaving from concurrent programs.

(`:monitor` pgm_1 pgm_2 ... pgm_n)
: Perform pgm_1 before every step of pgm_2, which is performed before every step of pgm_3, and so on.

(`:>>` *fexpr ... fexpr*)
: Succeed after evaluating the expressions (for effect).

(`:<<` *fexpr ... fexpr*)
: Like `:>>` except that evaluation only happens on failure.

(`:let` ((*var fexpr*) ... (*var fexpr*)) *pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) in an environment where each variable *var* has the value of *fexpr*.

(`:wait`)
: Succeed after the next exogenous action happens.

(`:search` *pgm ... pgm*)
: Perform (`:begin` *pgm...pgm*) online, but using lookahead for any nondeterminism to guard against failure.

Note that obtaining the value of these expressions is not the same as executing them. Execution is what is done by `ergo-do` alone.

Robotic Interface

A robotic interface is defined by a set of expressions of the form:

(`define-interface 'out` *body*)
: The body should evaluate to a function of one argument (like `displayln`) that will send an endogenous action to an outside target, blocking until the target is ready.

(`define-interface 'in` *body*)
: The body should evaluate to a function of no arguments (like `read`) that will return the next exogenous action received from an outside source, blocking when none.

The bodies can perform whatever initialization is needed for the functions they return to work properly (such as opening files, or making TCP connections). More than one in and out interface can be defined. The functions `write-endogenous` and `read-exogenous` can be used as default bodies.

Index of Technical Terms

Bibliography

[1] Vaishak Belle and Hector J. Levesque, Reasoning about continuous uncertainty in the situation calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Beijing, 2013.

[2] Vaishak Belle and Hector J. Levesque, Foundations for generalized planning in unbounded stochastic domains. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* Cape Town, South Africa, 2016.

[3] Vaishak Belle and Hector J. Levesque, ALLEGRO: belief-based programming in stochastic dynamical domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Buenos Aires, 2015.

[4] Blai Bonet, Giuseppe de Giacomo, Hector Geffner, and Sasha Rubin. Generalized planning: non-deterministic abstractions and trajectory constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Melbourne, 2017.

[5] Craig Boutilier, Ray Reiter, Mikhail Soutchanski and Sebastian Thrun. Decision-theoretic, high-level robot programming in the situation calculus. in *Proceedings of the AAAI National Conference on Artificial Intelligence*. Austin, TX, 2000.

[6] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. San Francisco: Morgan Kaufmann, 2004.

[7] Eugene Charniak. Bayesian networks without tears. *AI Magazine*, **12**(4):50–63, 1991.

[8] Kai Lai Chung. *A Course in Probability Theory*. San Diego, CA: Academic Press, 2001.

[9] Gregory F. Cooper. Probabilistic inference using belief networks is NP-hard. *Artificial Intelligence*, **42**:393–405, 1990.

[10] Adnan Darwiche. Bayesian networks. In [20], 467–499.

[11] Bruno de Finetti. *Theory of Probability: A Critical Introductory Treatment*, New York: Wiley, 1974.

[12] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, **121**(1-2):109–169, 2000.

[13] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardiña. On the semantics of deliberation in IndiGolog—from theory to implementation. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. Rome, 2002.

[14] Herbert B. Enderton. *A Mathematical Introduction to Logic*. San Diego, CA: Academic Press, 2001.

[15] Ron Fagin, Joseph Halpern, Yoram Moses and Moshe Vardi. *Reasoning about Knowledge*. Cambridge, MA: MIT Press, 1995.

[16] William Feller. *An Introduction to Probability Theory and Its Applications. Vol. 1* New York: Wiley, 1968.

[17] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. San Francisco: Morgan Kaufmann, 2004.

[18] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Josh Tenenbaum. Church: a language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*. Helsinki, 2008.

[19] Joseph Halpern. *Reasoning about Uncertainty*. Cambridge, MA: MIT Press, 2003.

[20] Frank van Harmelen, Vladimir Lifschitz and Bruce Porter, eds. *Handbook of Knowledge Representation.* Amsterdam: Elsevier, 2008.

[21] Jaako Hintikka. *Knowledge and Belief*. Ithaca, NY: Cornell University Press, 1962.

[22] Yuxiao Hu. *Generation and Verification of Plans with Loops* Ph.D. thesis, Dept. of Computer Science, University of Toronto, 2012.

[23] Yuxiao Hu and Giuseppe de Giacomo. Generalized planning: synthesizing plans that work for multiple environments. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Barcelona, 2011.

[24] Yuxiao Hu and Hector J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* Toronto, 2010.

[25] Edwin Jaynes. *Probability theory: The Logic of Science*. Cambridge, UK: Cambridge University Press, 2003.

[26] Finn Jensen. *An Introduction to Bayesian Networks*. London: University College London Press, 1996.

[27] Richard Jeffrey. *Probability and the Art of Judgment*. Cambridge, UK: Cambridge University Press, 1992.

[28] Hector J. Levesque. Planning with loops. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Edinburgh, 2005.

[29] Hector J. Levesque and Gerhard Lakemeyer. *The logic of knowledge bases*. Cambridge, MA: MIT Press, 2000.

[30] Hector J. Levesque and Gerhard Lakemeyer. Cognitive robotics. In [20], 869–886.

[31] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[32] Fangzhen Lin. Situation calculus. In [20], 649–670.

[33] John McCarthy. Programs with common sense. Reprinted in *Readings in Knowledge Representation*, ed. Ronald J. Brachman and Hector J. Levesque. San Francisco: Morgan Kaufmann, 1986.

[34] Elliott Mendelson. *Introduction to Mathematical Logic*. London: Chapman and Hall/CRC, 2009.

[35] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Revised second printing*. San Francisco: Morgan Kaufmann, 1988.

[36] Judea Pearl. Belief networks revisited. *Artificial Intelligence*, **59**:49–56, 1993.

[37] Ronald Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*. Menlo Park, CA: AAAI Press, 2002.

[38] Avi Pfeffer. *Practical Probabilistic Programming*. Manning Publications, 2016.

[39] Raymond Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Vladimir Lifschitz, ed. New York: Academic Press, 1991.

[40] Ray Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. Cambridge, MA: MIT Press, 2001.

[41] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice Hall, 2009.

[42] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. Cambridge, MA: MIT Press, 1997.

[43] Siddharth Srivastava. Foundations and applications of generalized planning. *AI Communications* **24**:349–351, 2011.

[44] Siddharth Srivastava, Neil Immerman and Shlomo Zilberstein Computing applicability conditions for plans with loops. In *Proceedings of the International. Conference on Automated Planning and Scheduling*. Toronto, 2010.

[45] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. Cambridge, MA: MIT Press, 2005.