Chapter 11

Prolog and logic programming

Contents

11.1 Logic as a programming language 525
11.2 The language 527
11.2.1 Concrete syntax 527
11.2.2 Abstract syntax (and no values) 529
11.2.3 Semantics 530
11.2.4 Primitive predicates 543
11.3 Small examples 544
11.3.1 Lists 544
11.3.2 Arithmetic 547
11.3.3 Sorting 548
11.3.4 Difference lists 550
11.4 Implementation 551
11.4.1 The database of clauses 551
11.4.2 Substitution and unification 552
11.4.3 Backtracking search 554
11.4.4 Processing clauses and queries 555
11.4.5 The read-eval-print loop 557
11.4.6 Primitives 557
11.4.7 Putting the pieces together 559
11.5 Larger example—The blocks world 559
11.6 Prolog as it really is 564
11.6.1 Syntax 564
11.6.2 Semantics 564
11.7 Prolog and mathematical logic 566
11.7.1 Propositional logic 567
11.7.2 First-order predicate logic 571
11.8 Summary 577
11.8.1 Glossary 577
11.8.2 Further Reading 579
11.9 Exercises 579

Logic is the study of valid arguments. Logic became important in the 19th century, when mathematicians discovered that certain seemingly ordinary arguments led to clearly erroneous results. Mathematicians undertook a more rigorous and formal study of the reasoning used in mathematical proofs, creating a new branch of mathematics called *mathematical logic.* Computer science has benefited in many ways, including influence on programming languages. Not only do we use mechanisms of formal logic (inference rules, derivations, and metatheory) for operational semantics and type systems, but we also have languages that draw on logical ideas. Functional languages were influenced by the λ calculus, which was an attempt by logicians to capture aspects of mathematical reasoning related to the definition and use of names. Clu was designed in part for program verification, which requires logic in which to state and reason about programs' specifications. And in this chapter, we see Prolog, a language designed to express logical rules directly.

There are many ways to formalize mathematical argument, but the most popular is *first-order predicate logic.* It is powerful enough to allow many arguments to be expressed, and it is weak enough to be mathematically tractable. In particular, first-order predicate logic is *decidable*; a computer can be programmed to carry out all reasoning used in this logic, and if a theorem is provable, the program would find a proof, at least in principle. The possibility of proving theorems in practice, and thereby automating much of mathematical thinking, has driven the computer-science specialty known as *mechanical theorem proving*, or *automated deduction.*

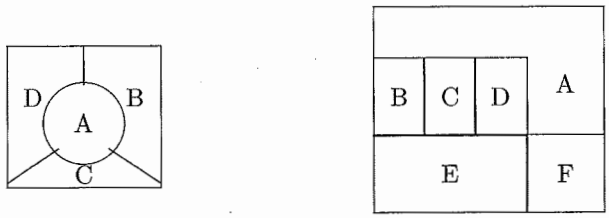
While working on automated deduction in the early 1970's, several researchers—especially Alain Colmerauer and Robert Kowalski—had the insight that led to the development of Prolog. They observed that when a mechanical theorem prover succeeded in proving a theorem, the proof normally had computational content. In particular, when a theorem has the form "for every x there is a y such that $P(x, y)$," a proof of that theorem might provide a method of *computing* y from a given x. For example, a theorem prover might be presented with (a formalized version of) the following theorem:

Theorem 11.1 For any list of integers L, there exists a list M such that M contains exactly the same elements as L, and the elements of M are arranged in non-descending order.

This theorem asserts that any list can be sorted. The proof of this theorem, discovered by a theorem-proving program, might give a method for *constructing* M from L. In short, a proof of this theorem might lead to a sorting algorithm.

Thus was conceived *logic programming.* Among logic-programming languages, the best known and most popular, especially for applications in artificial intelligence, is Prolog. The name itself is an abbreviation for "*pro*gramming in *log*ic." Other logic-programming languages exist, but nearly all are confined to research labs. Nonetheless, it is best to distinguish the idea of logic programming from its particular realization in Prolog; we show below how Prolog, despite its virtues, is not everything that you might hope for in a logic-programming language.

We introduce Prolog in our usual way, with first a general discussion and two simple examples, then formal presentations of its syntax and semantics, a longer list of examples, a presentation of our interpreter, and finally a larger example. We also present a special section devoted to logic, theorem proving, and their relation to Prolog.



(a) (b)

Figure 11.1: Maps

11.1 Logic as a programming language

Consider the problem of *3-coloring of maps*. We are given three colors—red, yellow, blue—with which to color the countries on a map. We must choose colors so that no two adjacent countries have the same color. The shapes of the countries are arbitrary, but a country must consist of a single piece. Countries are adjacent if they share a border; countries that meet only at a corner are not adjacent.

Is 3-coloring possible? Not always, as the map in Figure 11.1(a) shows. On the other hand, some maps, such as that in Figure 11.1(b), can be 3-colored, and it is not necessarily obvious when it is or isn't possible. Our program should give us a 3-coloring when one exists, or tell us if none does.

From the viewpoint of logic, it is easy to set down the parameters of the problem, that is, to assert what properties a solution, if it exists, must have. Take the map of Figure 11.1(b). Our solution should be a list of colors, A, B, ..., F, such that:

- A is different from B, C, D, and F.
- B is different from C and E.
- C is different from D and E.
- D is different from E.
- E is different from F.

To render this problem into Prolog, we first assert that the colors red, yellow, and blue are different. These six facts do the job.

525 ⟨*transcript* 525⟩≡ 526a▷

```
-> [fact].  /* makes the interpreter ready to receive facts */
-> different(yellow, red).
-> different(red, yellow).
-> different(yellow, blue).
-> different(blue, yellow).
-> different(blue, red).
-> different(red, blue).
```

We must state both that yellow is different from red *and* that red is different from yellow; Prolog can't tell on its own that we intend `different` to be a symmetric relation.

Prolog works by building up a *database* of facts about the world. Of course these facts needn't be true of the real world; they are claims made by the Prolog programmer. Given such a database, we can issue *queries* to find out what other facts Prolog can prove from the facts we have given. The language of facts has its own terminology: in each fact above, `different` is the *predicate*; `red`, `yellow`, and `blue` are *arguments*.

Prolog allows us to assert not only facts but also *rules*. Here we write a rule that describes what we mean by a coloring of map 11.1(b). In effect, we assert the shape of the map.

526a ⟨*transcript* 525⟩+≡ ◁525 526b▷

```
-> mapb_coloring(A, B, C, D, E, F) :-
     different(A, B), different(A, C), different(A, D), different(A, F),
     different(B, C), different(B, E), different(C, E), different(C, D),
     different(D, E), different(E, F).
```

This rule should be read as saying

> The colors `A`, `B`, ..., `F` constitute a coloring of `mapb` if `A` is different from `B`, `A` is different from `C`, `A` is different from `D`, and so on.

The language of rules also has its own terminology: in the rule above, the phrase `mapb_coloring(A, B, C, D, E, F)` is the *head* of the rule, sometimes also called the *conclusion*. The list of phrases following `:-` is the *body*; the individual elements may be called the *premises* or the *subgoals*. The symbols `A` through `F` are *logical variables*, to be filled in in such a way as to satisfy all the predicates.

The Prolog database mixes facts and rules freely; collectively they are called *clauses*. To use the database, we make a *query*, asking the interpreter to attempt to prove a *goal* from the given clauses.

526b ⟨*transcript* 525⟩+≡ ◁526a 527a▷

```
-> [query].  /* makes the interpreter ready to answer queries */
?- mapb_coloring(A, B, C, D, E, F).
A = yellow
B = red
C = blue
D = red
E = yellow
F = red
yes
```

The interpreter has succeeded in assigning colors to the variables so as to constitute a coloring. It not only reports back that the query can be satisfied; it also provides a satisfying assignment to the logical variables.

This example shows an unusual property of our μProlog interpreter: it has two *modes*. In *rule mode*, the prompt is `->`, and the interpreter silently accepts facts or rules. In *query mode*, the prompt is `?-`, and the interpreter answers queries based on the facts known to it. This odd style of interaction is necessary because Prolog uses the *same* concrete syntax for both queries and facts; other implementations of Prolog also use modes. We could get rid of the modes by using nonstandard syntax, but then you wouldn't be able to use our code with other Prolog interpreters.

Entering "`[query].`" puts the μProlog interpreter into query mode. Entering "`[rule].`" or "`[fact].`"[1] puts it into rule mode.

[1] Or "`[user].`" or "`[clause].`" Don't ask.

Here is a more familiar example, checking membership of a value in a list. Lists are built into μProlog using the familiar `cons` and `nil`.[2] Like ML, Prolog uses special syntax for lists. We write "`[]`" for the empty list, `[`x`|`l`]` for `cons(`x`, `l`)`, and `[`a`,`b`,...,`z`]` for `cons(`a`, cons(`b`, cons(..., cons(`z`, [])))`. There is also a more rarely used form; `[`a`,`b`,...,`y`|`z`]` stands for `cons(`a`, cons(`b`, cons(..., cons(`y`, `z`))))`.

We would like to define a predicate `member` such that `member(`x`, `l`)` is true if x occurs in the list l. The following assertions about `member` are certainly valid:

- `member(`x`, `l`)` is true if l has the form `[`x`|`m`]`, for any list m.
- `member(`x`, `l`)` is also true if l has the form `[`y`|`m`]`, for any y and m, provided that `member(`x`, `m`)` is true.
- `member(`x`, `l`)` is false, otherwise.

We can state these assertions in Prolog as follows:

527a ⟨*transcript* 525⟩+≡ ◁526b 527b▷

```
?- [rule].
-> member(X, [X|M]).
-> member(X, [Y|M]) :- member(X, M).
```

These clauses state only the cases when the goal is true; when no clause applies, Prolog always considers the goal to be false. (As in other forms of logic, Prolog doesn't really deal in truth or falsehood; it deals in *provability*. Rules say only when a judgment is provable.)

As above, we can use these clauses by making a query involving `member`:

527b ⟨*transcript* 525⟩+≡ ◁527a 531a▷

```
-> [query].
?- member(3, [2, 3]).
yes
?- member(3, [2, 4]).
no
```

This, then, is the idea behind Prolog: the programmer describes the logical properties of the result of a computation, and the interpreter searches for a result having those properties.

11.2 The language

11.2.1 Concrete syntax

Our examples show most of Prolog. Data structures are defined implicitly by introducing constructors like `cons` and constants like `nil`, which are called *functors* in Prolog terminology. There are integers and built-in predicates as well. Figure 11.2 gives the full concrete syntax of μProlog.

The most salient fact about the syntax is that unlike other languages in this book, Prolog doesn't have a syntactic category definitions. At top level, an input to a Prolog interpreter is either a clause or a query. Neither a clause nor a query defines anything, and we see below that Prolog does not use a top-level environment—the only state maintained at top level is the database of clauses.

[2] Lists are built into other versions of Prolog as well, but these versions often use the constructor "." (a single dot) instead of "cons." The distinction is not usually important, because it is typical to use syntactic sugar for lists.

clause-or-query ::= *clause* | *query* | *mode-change* | *use*

clause ::= *goal* [`:-` *goals*] `.`

query ::= *goals* `.`

goals ::= *goal* {`,` *goal*}

goal ::= *term* `is` *term*
| *term binary-predicate term*
| *predicate* [`(`*term* {`,` *term*}`)`]

term ::=
| *term binary-functor term*
| *functor* [`(`*term* {`,` *term*}`)`]
| `[`*term*{`,` *term*} [`|`*term*]`]`
| `[]`
| *integer*
| *variable*

mode-change ::= `[query].` | `[rule].` | `[fact].` | `[clause].` | `[user].`

use ::= `[`*filename*`].`

predicate ::= ! | name beginning with lower-case letter

binary-predicate ::= name formed from symbols `|%^&*-+:=~<>/?'$\`

functor ::= name beginning with lower-case letter

binary-functor ::= name formed from symbols `|%^&*-+:=~<>/?'$\`

variable ::= name beginning with upper-case letter

Figure 11.2: Concrete syntax of μProlog

Clauses and queries are formed from goals, which are themselves formed from terms. Terms are roughly analogous with expressions in other languages. Here are some examples:

```
[14,7]
mktree(1, nil, nil)
ratnum(17, 5)
on(a, table)
```

These structures are called "terms" rather than "expressions" because Prolog doesn't "evaluate" them. In Prolog, terms do duty as *both* abstract syntax and values. *Functors* such as `cons`, `mktree`, and `ratnum` don't actually *do* anything; they are just a way of representing structured values. Terms can also contain variables, which are identifiable as such because a variable starts with a capital letter, as in `[X|L]` or `on(B, table)`.

μProlog has a number of predicates built in: the comparisons `<`, `>`, `>=`, and `=<`,[3] `print` for printing terms, and `is` for simple computations with numbers. We explain further at the end of the next subsection (page 543).

One last burst of terminology: We sometimes call the clause head the *left-hand side*, as it appears to the left of `:-`. A term or clause with no variables is called *ground*.

Although we use parenthesized-prefix syntax for every other language in this book, there are several reasons we do not use it for μProlog.

- Prolog has little in common with the other languages in this book; it has no functions, no assignment, no control, no methods, and no evaluation. Because there are few opportunities to draw parallels with other languages, there is little reason to use the same syntax as other languages.
- Lots of interesting Prolog programs require extensive search, and our simple interpreter can't compete with Prolog interpreters built by specialists. So we have extra incentive to be sure that μProlog programs will run on real Prolog systems, some of which can be freely downloaded using the Internet.
- The "Edinburgh syntax," which is more or less standard, and more or less what we use for μProlog, is simple enough that it is easy to learn. It is also easy to write a parser for the Edinburgh syntax.

11.2.2 Abstract syntax (and no values)

Of all the languages in this book, Prolog has the simplest structure. Unusually, there is no distinction between "values" and "abstract syntax;" both are represented as terms. A term is a variable, a literal constant, or an application of a *functor* to a list of terms.[4] The only kinds of literal constants in μProlog are integers.

529a ⟨*abstract syntax and values* 529a⟩≡ (559a) 529b▷

```
datatype term = VAR     of string
              | LITERAL of int
              | APPLY   of string * term list
```

Goals and applications have identical structure. A term can be a functor applied to a list of terms; a goal is a predicate applied to a list of terms. A "functor" or a "predicate" is uniquely identified by its name.

529b ⟨*abstract syntax and values* 529a⟩+≡ (559a) ◁529a 529c▷

```
type goal = string * term list
```

A clause is a conclusion and a list of premises, all of which are goals. If the list of premises is empty, the clause is a "fact;" otherwise it is a "rule," but these distinctions are useful only for thinking about and organizing programs—the underlying meanings are the same. Writing our implementation in ML enables us to use the identifier `:-` as a datatype constructor for clauses; we even make it infix.

529c ⟨*abstract syntax and values* 529a⟩+≡ (559a) ◁529b 530▷

```
datatype clause = :- of goal * goal list
infix 3 :-
```

[3] Prolog is intended primarily for symbolic computation, not for numeric computation, so the left-arrow symbol `<=` is considered too valuable to use for "less than or equal," which is written `=<`.

[4] Functors play the same role in Prolog that datatype constructors play in ML, including participation in pattern matching.

At the top level, we can do three things: add a clause to the database, ask a query, or include a file. The switch back and forth between query mode and rule mode is hidden from the code in this chapter; the details are buried in Section J.3.3. In other interpreters, the thing that appears at top level is a "definition;" because Prolog has no definitions, we call the top-level syntactic category `cq`, which is short for *clause-or-query*.

530 ⟨*abstract syntax and values* 529a⟩+≡ (559a) ◁529c 551a▷

```
datatype cq
  = ADD_CLAUSE of clause
  | QUERY      of goal list
  | USE        of string
```

11.2.3 Semantics

A typical Prolog program consists of a list of clauses $C_1, \ldots, C_n$ and a query g. (For simplicity, we assume the query has just one goal.) Each clause has the form G `:-` $H_1, \ldots, H_m$. Prolog semantics can be understood in two different ways, called the *logical interpretation* and the *procedural interpretation*. Of these, the logical interpretation is simpler but occasionally misleading; the procedural interpretation is more accurate.

The logical interpretation

According to the logical interpretation, the function of the Prolog interpreter is to prove goals, and the clauses give the rules of inference to be used in doing proofs. The typical clause given above should be read as follows:

> If we can prove $H_1, \ldots, H_m$, then we can infer G. More generally, if $G, H_1, \ldots, H_m$ contain variables, then for all possible values of those variables, G is true if $H_1, \ldots, H_m$ are true. In other words, the clause can be read as a rule of inference:
>
> $$\frac{H_1 \quad \cdots \quad H_m}{G}.$$
>
> In the special case $m = 0$, the clause "G." means that G is true no matter what values are given to its variables.

Thus, to satisfy the query "g.," we need to find values for the variables occurring in g such that it can be proven using the clauses as inference rules.

For example, the rule

$$\texttt{member}(X, [Y|M])\ \texttt{:-}\ \texttt{member}(X, M)$$

type `clause` 529c
type `goal` 529b

represents this assertion: for any values X, Y, and M, if X occurs in M, then X occurs in $[Y|M]$. Thus, from

$$\texttt{member}(3, [4, 3])$$

we can infer

$$\texttt{member}(3, [7, 4, 3]).$$

Similarly,

$$\texttt{member}(X, [X|L])$$

states that $\texttt{member}(X, [X|L])$ is true no matter what the values of X and L.

These two rules, regarded as assertions, are evidently true, each on its own and independent of the other and of any other assertions. In the logical interpretation, then, their position within the overall list of clauses and the order in which they occur have no effect on their meaning. Similarly, in a rule like:

$$\texttt{mapb\_coloring}(A, B, C, D, E, F) \texttt{ :- different}(A, B), \texttt{different}(A, C), \ldots$$

the order of the subgoals is immaterial;

$$\texttt{mapb\_coloring}(A, B, C, D, E, F) \texttt{ :- different}(A, C), \texttt{different}(A, B), \ldots$$

is precisely the same assertion.

One of the remarkable features of Prolog, implicit in the logical interpretation, is that it permits programs to "run backwards." Although the obvious purpose of `member` is to check whether a given value occurs in a given list, as in the examples above, the same clauses can be used to choose an element from a list:

531a ⟨*transcript* 525⟩+≡ ◁527b 531b▷

```
-> [query].
?- member(X, [4, 3]).
X = 4
yes
```

In this case, the system somehow "reasoned" that, if `X` were to equal 4, then the clause

```
member(X, [X|L]).
```

could be used to infer that the query is true. It could also, by a longer chain of reasoning, have determined that `X` can equal 3. The logical interpretation doesn't tell us which value will be bound to `X`, only that it will be one of them.

The interpreter can even find a list that contains given values:

531b ⟨*transcript* 525⟩+≡ ◁531a 532▷

```
?- member(3, L), member(4, L).
L = [3, 4|_M135]
yes
```

The result list contains an internal variable, `_M135`, indicating that the rest of the list is undetermined. This is natural, since all we stated about `L` was that it contains `3` and `4`, so there is no way to know what else it might contain.

Making the logical interpretation precise

The logical interpretation of Prolog has a simple and elegant formalization as a *nondeterministic* set of logical inference rules. To present it, we need to return to substitution, which we first presented in Chapter 7.

Definition 11.1 A *substitution* θ is a function θ from terms to terms that satisfies the following equations:

$$\begin{aligned} \theta(\text{APPLY}(f, t_1, \ldots, t_n)) &= \text{APPLY}(f, \theta(t_1), \ldots, \theta(t_n)) \\ \theta(\text{LITERAL}(n)) &= \text{LITERAL}(n) \end{aligned}$$

We also require that a substitution be finite, i.e., $\theta(\text{VAR}(X)) = \text{VAR}(X)$ for all but finitely many X.

You might like to confirm these facts about substitutions:

- We can write any substitution as $\theta = \{\mathtt{X}_1 \mapsto \mathtt{t}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{t}_n\}$, where $\theta(\text{VAR}(\mathtt{X}_i)) = \mathtt{t}_i$, and if $\forall i. X \neq \mathtt{X}_i$, $\theta(\text{VAR}(X)) = \text{VAR}(X)$. The variables $\mathtt{X}_i$ are said to be *bound in* this substitution.
- If functions θ_1 and θ_2 are substitutions, the composition $\theta_2 \circ \theta_1$ is also a substitution.

Definition 11.2 Given a substitution θ and a goal g, the *application of θ to g*, written as $\widehat{\theta}(g)$, is the goal obtained by applying θ to each argument of g. Equivalently, if we write $\theta = \{\mathtt{X}_1 \mapsto \mathtt{t}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{t}_n\}$, then $\widehat{\theta}(g)$ is the goal obtained by replacing each occurrence of $\mathtt{X}_i$ by $\mathtt{t}_i$ in g. We sometimes call $\widehat{\theta}$ a *lifted form* of θ.

For example, if

$$g = \texttt{member(X, [Y|L])},$$

and

$$\theta = \{\mathtt{X} \mapsto 3,\ \mathtt{L} \mapsto \texttt{[4|M]}\},$$

then

$$\widehat{\theta}(g) = \texttt{member(3, [Y,4|M])}.$$

The lifted forms of substitutions also compose: $\widehat{\theta_2 \circ \theta_1} = \widehat{\theta_2} \circ \widehat{\theta_1}$.

Now we are ready to formalize Prolog's notion of query. A query in Prolog is not simply satisfied—its satisfaction produces a substitution. That is, if the query g contains variables, the interpreter must find terms for those variables such that, after replacing each variable in g by its term, the resulting goal is satisfiable. This chapter contains many examples; the interpreter prints the substitution as a set of equations following the query.

532 ⟨*transcript* 525⟩+≡ ◁531b 533▷

```
-> [query].
?- mapb_coloring(A, B, C, D, E, F).
A = yellow
B = red
C = blue
D = red
E = yellow
F = red
yes
```

Formally, we write the idea "goal g is satisfiable using database D and substitution θ" as the judgment $D \vdash \widehat{\theta} g$. Of course, in the general case, a query has more than one goal, so the general form of the judgment is

$$D \vdash \widehat{\theta} g_1, \ldots, \widehat{\theta} g_n.$$

In the logical interpretation, the satisfaction of the different goals is independent; the only requirement is that the *same* substitution satisfy them all.

$$\frac{D \vdash \widehat{\theta} g_1 \quad \cdots \quad D \vdash \widehat{\theta} g_n}{D \vdash \widehat{\theta} g_1, \ldots, \widehat{\theta} g_n} \qquad \text{(LogicalQueries)}$$

Any individual goal is satisfied if, of course, it can be "made the same" as the left-hand side of some clause, and we can prove the right-hand side. Here, a crucial fact comes into play. *The exact variables used within a clause are arbitrary, bearing no relationship to variables of the same name that may appear in a query or in a subgoal from another clause (or even another instance of the same clause).* In other words, a variable in a Prolog clause is like a formal parameter of a function in another language; just as you can have different activations of a function with different values bound to a formal parameter, you can have different uses of a clause with different things substituted for a variable.

We illustrate this property with an example. Suppose we want to find out if the variable `M` is a member of the list `[1|nil]`. The answer is yes, provided `M=1`.

533 ⟨*transcript* 525⟩+≡ ◁532 537a▷

```
?- member(M, [1|nil]).
M = 1
yes
```

But how do we prove that this query is satisfied? By appealing to one of the clauses in the database: `member(X, [X|M])`. The `M` in the clause *must* be independent of the `M` in the query, because `M` cannot equal both `1` and `nil` at the same time.

In Impcore, μScheme, μSmalltalk, and other languages, we achieve the necessary independence by using an environment to keep track of the values of formal parameters, and every activation of a function gets its own private environment. In Prolog, substitutions play the role of environments, and so we use *different* substitutions for the clause and the query.

Now we can give a precise definition of the semantics of Prolog according to the logical interpretation. A substitution θ satisfies a query g if there is some clause in the database such that another substitution θ' makes the conclusion the same as g, and under substitution θ' we can prove all the premises.

$$\frac{\begin{array}{c} C \in D \qquad C = G \text{ :- } H_1, \ldots, H_m \\ \widehat{\theta'}(G) = \widehat{\theta} g \\ D \vdash \widehat{\theta'}(H_1), \ldots, \widehat{\theta'}(H_m) \end{array}}{D \vdash \widehat{\theta} g} \qquad \text{(LogicalQuery)}$$

Putting all this together, the logical interpretation says that to satisfy a query g, there must be a pair of substitutions θ and θ' such that θ applied to the original query is the same as θ' applied to the head of some clause, and the subgoals produced by applying θ' to the subgoals of that clause are all satisfiable.

The rules above don't specify a deterministic algorithm. To get some insight into what else we need, consider these questions:

- When applying rule LOGICALQUERY, which clause $C \in D$ do we choose?
- Given g and G, how do we discover a *pair* of substitutions such that $\widehat{\theta}(g) = \widehat{\theta'}(G)$?

A suitable pair of substitutions can be found by a form of unification. Choosing a clause is more complicated; Prolog requires a backtracking search, which does not always terminate. To understand the details, we need to move from the *logical* interpretation to the *procedural* interpretation, which we do by way of substitutions.

Pairs of substitutions

Like ML type inference, Prolog search works by computing and accumulating substitutions. We know from Chapter 7 that given goals G and g, we can use *unification* (see sidebar) to find a *single* substitution θ such that $\widehat{\theta}(G) = \widehat{\theta}g$. This section shows how to twist unification to find a pair of substitutions.

To motivate the algorithm, we return to the example above, where simple unification doesn't work. If $g =$ `member(M, [1|nil])` and $G =$ `member(X, [X|M])`, there is no single θ such that $\widehat{\theta}(G) = \widehat{\theta}g$, since no single substitution can simultaneously satisfy the constraints `M` $=$ `X`, `X` $= 1$, and `M` $=$ `nil`. The crux of the difficulty is that variable `M` appears in both G and g, but the semantics of Prolog require that we treat these two instances of `M` differently. (Formally, we need to apply θ' to the `M` in G, but θ to the `M` in g.) The idea of the solution is this: if G and g had no variables in common, maybe we could use a single substitution for both, so why not *rename the variables in* G, then unify?

This renaming is similar to α-conversion in the λ calculus. To make the idea precise, we need one more definition.

Definition 11.5 A *renaming of variables* is a substitution θ_α in which $\theta_\alpha(\text{VAR}(X))$ is always a variable, never an application or an integer.

When considered as a function from terms to terms, a renaming of variables has an inverse function, which is also a substitution; we write that substitution θ_α^{-1}.

It's easy to see that if we rename variables, then unify, the answer gives us the pair of substitutions we want. If θ_α is a renaming of variables, and if $\texttt{unify}(\widehat{\theta_\alpha}(G), g) = \theta$, then $(\widehat{\theta \circ \theta_\alpha})(G) = \widehat{\theta}(g)$, and we have our pair of substitutions.

The hard part is proving the other direction: if there is *any* pair of substitutions that works, we find one by renaming and unifying.

Theorem 11.2 If there exists a pair of substitutions θ, θ' such that $\widehat{\theta}(g) = \widehat{\theta'}(G)$, then there exists a renaming of variables θ_α and a unifier θ_0 such that $\widehat{\theta_0}(g) = \widehat{\theta_0}(\widehat{\theta_\alpha}(G))$.

If we can prove the theorem, we can rewrite the operational semantics as follows:

$$\frac{\begin{array}{cc} C \in D & C = G \texttt{ :- } H_1, \ldots, H_m \\ \widehat{\theta}(\widehat{\theta_\alpha}(G)) = \widehat{\theta}(g) & \theta_\alpha \text{ is a renaming of variables} \\ & \theta' = \theta \circ \theta_\alpha \\ & D \vdash \widehat{\theta'}(H_1), \ldots, \widehat{\theta'}(H_m) \end{array}}{D \vdash \widehat{\theta} g} \qquad \text{(LOGICALQUERY)}$$

This is the way our interpreter works.

Sidebar: Unification

Definition 11.3 A substitution θ_1 is *more general* than a substitution θ_2 if there exists a θ_3 such that $\theta_2 = \theta_3 \circ \theta_1$.

The more general a substitution is, the fewer things it changes.

Definition 11.4 *Unification* is the process of finding, for given goals g_1 and g_2, a substitution θ such that $\widehat{\theta}(g_1) = \widehat{\theta}(g_2)$. Furthermore, θ must be a *most general* subsitution satisfying this equation, i.e., given any θ' such that $\widehat{\theta'}(g_1) = \widehat{\theta'}(g_2)$, then there is a substitution θ'' such that $\theta' = \theta'' \circ \theta$. We do not bother to verify generality in our examples.

Some examples:

1. g_1 = `member(3, [3|nil])`
   g_2 = `member(X, [X|L])`
   θ = $\{$`X` $\mapsto$ `3`, `L` $\mapsto$ `nil`$\}$

2. g_1 = `member(Y, [3|nil])`
   g_2 = `member(X, [X|L])`
   θ = $\{$`Y` $\mapsto$ `3`, `X` $\mapsto$ `3`, `L` $\mapsto$ `nil`$\}$

3. g_1 = `member(3, [4|nil])`
   g_2 = `member(X, [X|L])`
   do not unify, since no substitution can map `X` to both `3` and `4`.

4. g_1 = `length([3|nil], N)`
   g_2 = `member(X, [X|L])`
   do not unify, since obviously no substitution can make them equal.

5. g_1 = `member(X, [X|L])`
   g_2 = `member(Y, cons(mkTree(Y, nil, nil), M))`
   do not unify. Since the `X` in g_1 and the `Y` in g_2 must be replaced by the same term, say e, we end up with goals

$$\begin{aligned} \widehat{\theta}(g_1) &= \texttt{member(}e\texttt{, [}e\texttt{|L])} \\ \widehat{\theta}(g_2) &= \texttt{member(}e\texttt{, [mkTree(}e\texttt{, nil, nil)|L])} \end{aligned}$$

   which are necessarily distinct because e can never equal `mkTree(`e`, nil, nil)`.

Example 5 illustrates a tricky aspect of implementing Prolog. Even if e is a variable, it does not unify with the term `mkTree(`e`, nil, nil)`. It is natural to try to unify a variable `X` with a term `t` using the substitution $\{$`X` $\mapsto$ `t`$\}$, but this substitution works only if `X` does not occur in `t`. Unification of a variable with a term therefore requires an "occurs check," which although expensive is an essential part of the semantics.

A rigorous proof of Theorem 11.2 is beyond the scope of this book, but if you are mathematically inclined, you might enjoy reading the following sketch. Otherwise, you may prefer to skip to the material on the procedural interpretation.

We begin by observing that we can strengthen the hypothesis of the theorem. We can apply a renaming of variables to both sides of the equation $\widehat{\theta}(g) = \widehat{\theta'}(G)$ so that the result has no variables in common with G or g. To simplify notation, we simply assume without loss of generality that $\widehat{\theta}(g)$ has no variables in common with G or g. Also without loss of generality, we assume that θ changes only variables in g, and θ' changes only variables in G.

To prove the theorem, first choose a renaming of variables θ_α such that $\widehat{\theta_\alpha}(G)$ has no variables in common with either g or $\widehat{\theta}(g)$. We know by hypothesis that $\widehat{\theta}(g) = \widehat{\theta'}(G)$; since θ_α has an inverse, we can rewrite that equation as $\widehat{\theta}(g) = \widehat{\theta'}(\widehat{\theta_\alpha^{-1}}(\widehat{\theta_\alpha}(G)))$. Rewriting again, we have

$$\widehat{\theta}(g) = (\widehat{\theta'} \circ \widehat{\theta_\alpha^{-1}})(\widehat{\theta_\alpha}(G))). \qquad (*)$$

Because θ' changes only variables in G, $\theta' \circ \theta_\alpha^{-1}$ changes only variables in $\widehat{\theta_\alpha}(G)$.

Now because $\widehat{\theta_\alpha}(G)$ and g have no variables in common, and because θ changes only variables in g, we have $\widehat{\theta}(\widehat{\theta_\alpha}(G)) = \widehat{\theta_\alpha}(G)$. Similarly, because $\widehat{\theta_\alpha}(G)$ and $\widehat{\theta}(g)$ have no variables in common, and because $\theta' \circ \theta_\alpha^{-1}$ changes only variables in $\widehat{\theta_\alpha}(G)$, we have $(\widehat{\theta'} \circ \widehat{\theta_\alpha^{-1}})(\widehat{\theta}(g)) = \widehat{\theta}(g)$. Substituting equals for equals in $(*)$ yields

$$(\widehat{\theta'} \circ \widehat{\theta_\alpha^{-1}} \circ \widehat{\theta})(g) \quad = \quad (\widehat{\theta'} \circ \widehat{\theta_\alpha^{-1}} \circ \widehat{\theta})(\widehat{\theta_\alpha}(G))).$$

Therefore $\theta_0 = \widehat{\theta'} \circ \widehat{\theta_\alpha^{-1}} \circ \widehat{\theta}$. Of course, in the implementation, we don't bother with this laborious construction. Having proved the theorem, we simply try to unify $\widehat{\theta_\alpha}(G)$ and g; if there is no unifier, the theorem tells us there is no pair of substitutions.

The procedural interpretation

A logic program, like any program, ultimately instructs a computer to take certain actions. The procedural interpretation of Prolog tells us what actions the interpreter takes in attempting to answer a query. In particular, it tells us how the interpreter searches for clauses and how the interpreter computes and composes substitutions.

You cannot be an effective Prolog programmer unless you understand the procedural interpretation; otherwise you risk

- Being unable to understand the performance of your Prolog programs
- Writing Prolog clauses that send the interpreter into an infinite loop

Worse, as we see in Section 11.6.2, some features of Prolog, notably the "cut," can be understood *only* by appealing to the procedural interpretation.

In this section, we present the procedural interpretation in three steps. We present first the search for clauses, then backtracking, both without substitutions. We then give a complete version with substitutions.

The first version applies only to *ground* clauses and goals, i.e., those without variables.

Version 1. We are given database $D = C_1, \ldots, C_n$ and query g, and we wish to know whether g is satisfied, i.e. $D \vdash g$. We look through the clauses C_i *in the order in which they appear in* D; the first time we find a clause whose left-hand side is g, say g :- $H_1, \ldots, H_m$, we attempt to satisfy $H_1, \ldots, H_m$, in that order, following the same procedure for each H_j. When each of the H_j has been satisfied, g has been satisfied; if we are unable to satisfy any one of them, we fail to satisfy g. Also, if no clause exists whose left-hand side matches g, we fail to satisfy g.

Version 1 of the procedural interpretation explains the execution of some variable-free programs. Consider this example:

537a ⟨*transcript* 525⟩+≡ ◁533 537b▷

```
-> [rule].
-> imokay :- youreokay, hesokay.    /* clause C1 */
-> youreokay :- theyreokay.         /* clause C2 */
-> hesokay.                         /* clause C3 */
-> theyreokay.                      /* clause C4 */
-> [query].
?- imokay.
yes
```

The procedural interpretation explains what happened:

1. The goal is `imokay`. The first (and only) matching clause is C_1. The new goals are `youreokay` and `hesokay`.
2. We attempt to satisfy `youreokay` first. Clause C_2 matches and requires that `theyreokay` be satisfied.
3. `theyreokay` is matched by clause C_4, which has no further goals. Thus, `theyreokay` is satisfied, and therefore so is `youreokay`.
4. `hesokay` is matched by C_3, with no further goals. Thus, `hesokay` is satisfied, and so is `imokay`.

For this example, the logical interpretation gives us the same result. In general, the logical interpretation can give us answers even when the procedural interpretation does not. Consider the program obtained by adding three clauses to the previous four:

537b ⟨*transcript* 525⟩+≡ ◁537a 538a▷

```
?- [rule].
-> hesnotokay :- imnotokay.  /* clause C5 */
-> shesokay :- hesnotokay.   /* clause C6 */
-> shesokay :- theyreokay.   /* clause C7 */
-> [query].
?- shesokay.
yes
```

According to the logical interpretation, she is certainly okay, because they are. However, version 1 of the procedural interpretation does not lead us to that conclusion. Rather, it directs us to prove `shesokay` by applying C_6, which in turn requires us to prove `hesnotokay`, for which we apply C_5, which requires us to prove `imnotokay`, which cannot be proven.

What is missing is the notion of *backtracking*. There can be more than one clause that applies to a given goal, and not all applicable clauses necessarily lead to a solution. The next version of the procedural interpretation, again applicable only to variable-free programs, includes backtracking.

Version 2. To satisfy g, look through the clauses C_i in order; the first time a clause is found whose left-hand side equals g, say $C_i = g$:- $H_1, \ldots, H_m$, we attempt to satisfy $H_1, \ldots, H_m$, in that order, following the same procedure for each that was followed for g. When each of the H_j has been satisfied, g has been satisfied. If we are unable to satisfy any one of them, we go back to the list of clauses, starting from C_{i+1}, and look for another clause whose left-hand side equals g. If the end of the list is reached without finding any matching clause, we fail to satisfy g.

Now we can prove `shesokay`:

1. Apply C_6, since it is the first clause that applies. Our new goal is `hesnotokay`.
2. Clause C_5 applies, and calls for us to prove `imnotokay`.
3. `imnotokay` is not provable, since no clause applies to it.
4. Backtracking, we continue trying to satisfy `hesnotokay`, starting from C_6. C_6 and C_7 do not apply, so we fail.
5. Backtracking, we continue trying to satisfy `shesokay`, starting from C_7. C_7 applies, and our new goal is `theyreokay`.
6. C_4 applies, and we are done.

Version 2 brings the procedural interpretation more into line with the logical interpretation, but the two interpretations still don't agree. Consider these two new clauses:

538a ⟨*transcript* 525⟩+≡ ◁537b 544▷

```
?- [rule].
-> hesnotokay :- shesokay.    /* clause C8 */
-> hesnotokay :- imokay.      /* clause C9 */
```

538b ⟨*bad transcript* 538b⟩≡

```
-> [query].
?- shesokay.
... never returns ...
```

In logic, if a conclusion can be inferred from some set of facts, it can still be inferred when new facts are added. Therefore, in the logical interpretation, `shesokay` is still provable after adding C_8 and C_9. But here is what happens according to version 2 of the procedural interpretation:

1. C_6 applies; the new goal is `hesnotokay`.
2. C_5 applies, leaving goal `imnotokay`, which still cannot be satisfied. Backtrack and continue trying to satisfy `hesnotokay`.
3. C_8 applies; the new goal is `shesokay`.
4. C_6 applies; the new goal is `hesnotokay`.
5. ⋮

According to the procedural interpretation, we are in an infinite loop, and indeed Prolog does go into a loop. This is because *the logical interpretation does not reflect the actual semantics of Prolog.* The procedural interpretation, which prescribes exactly how Prolog searches for clauses, is the accurate one.

According to the procedural interpretation, the order in which clauses appear is critically important. For example, if C_8 and C_9 had been reversed, `shesokay` would have been proven with no difficulty. By contrast, in the logical interpretation, the order of clauses does not affect the ability to satisfy a goal. While we might wish to have a programming language based on pure logic, which always finds a solution when one exists, this is not how Prolog works.

Version 2 is accurate and sufficient for understanding how Prolog executes *variable-free* programs, but it says nothing about variables. We can now state the procedural interpretation in full:

Final version. In the general case, we are given a database D and a query $g_1, \ldots, g_m$, and we need to find θ such that $D \vdash \widehat{\theta}(g_1), \ldots, \widehat{\theta}(g_m)$. If $m = 0$, the empty query is trivially satisfied by the identity substitution. Otherwise, follow one of the two procedures below.

If $m = 1$, to satisfy a single goal g, look through the clauses C_i in order. If a clause is found, say $C_i = G$ `:-` $H_1, \ldots, H_k$, such that there are substitutions θ and θ' making $\widehat{\theta}(g) = \widehat{\theta'}(G)$, attempt to satisfy $\widehat{\theta'}(H_1), \ldots \widehat{\theta'}(H_k)$. If this attempt fails, continue from C_{i+1} looking for an applicable clause; if none is found, the attempt to satisfy g fails.

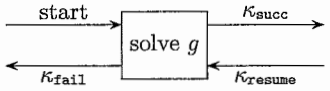
To satisfy a list of goals $g_1, \ldots, g_m$ where $m > 1$, such as might be produced from $\widehat{\theta'}(H_1), \ldots \widehat{\theta'}(H_m)$, use the following rule:

$$\frac{D \vdash \widehat{\theta_1}(g_1) \qquad D \vdash \widehat{\theta'}(\widehat{\theta_1}(g_2)), \ldots, \widehat{\theta'}(\widehat{\theta_1}(g_m))}{D \vdash (\widehat{\theta'} \circ \widehat{\theta})(g_1), \ldots, (\widehat{\theta'} \circ \widehat{\theta})(g_m)} \qquad \text{(PROCEDURALQUERIES)}$$

In other words, follow the procedure for satisfying goal g_1. If it is satisfied, it yields a substitution θ_1; now, attempt to satisfy $\widehat{\theta_1}(g_2)$. Continue in this way, eventually satisfying goal $(\widehat{\theta_{m-1}} \circ \cdots \circ \widehat{\theta_1})(g_m)$, yielding substitution θ_m. The attempt to satisfy $g_1, \ldots, g_m$ has now succeeded, yielding the substitution $\theta_m \circ \theta_{m-1} \circ \cdots \circ \theta_1$.

There is another kind of backtracking possible here that did not arise in version 2, namely backtracking *within right-hand sides.* Suppose the attempt to satisfy some $(\widehat{\theta_{j-1}} \circ \cdots \widehat{\theta_1})(g_j)$ had failed. The problem might be that we satisfied $g_1, \ldots, g_{j-1}$ *using the wrong substitution.* So, we backtrack and attempt to *resatisfy* $(\widehat{\theta_{j-2}} \circ \cdots \circ \widehat{\theta_1})(g_{j-1})$, hopefully yielding a different substitution θ'_{j-1}, and then continue as before, trying to satisfy $(\widehat{\theta'_{j-1}} \circ \widehat{\theta_{j-2}} \circ \cdots \circ \widehat{\theta_1})(g_j)$. Failure occurs only when we backtrack to g_1 and have exhausted all ways of satisfying it.

The procedural interpretation of Prolog can be quite difficult to understand. Luckily there is a conceptual tool, the *Byrd box* (Byrd 1980), which not only makes it easier to understand how Prolog works, but which leads to a very simple implementation in continuation-passing style. You have actually seen the Byrd box already, in Section 3.10.1, in its guise as a solver for Boolean formulas. In Prolog, the Byrd box is a "solver" for a single goal, with this structure:



The idea is simple:

1. We create a Byrd box for every goal g. The Byrd box searches for substitutions such that $D \vdash \widehat{\theta}(g)$.

2. There might be more than one such substitution, and we don't want to compute any more than necessary, so instead of simply having the Byrd box *return* a substitution, we pass it a *success continuation* $\kappa_{\texttt{succ}}$. The continuation takes θ as a parameter.

3. Whether backtracking is needed depends on the goals that *follow* g; these are exactly the goals that $\kappa_{\texttt{succ}}$ tries to satisfy. If they can't be satisfied, we need to go back to our original Byrd box and ask for another substitution. The Byrd box creates another continuation $\kappa_{\texttt{resume}}$ for this purpose.

4. Finally, if the Byrd box fails, or if it simply runs out of substitutions, what do we do? We can't simply give up, because it's possible that backtracking might lead to another solution. So we pass the Byrd box a *failure continuation* $\kappa_{\texttt{fail}}$, which it calls if it can't find a substitution, or if it has to backtrack.

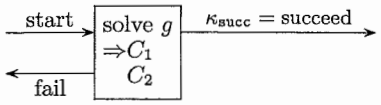
Function `solveOne` in ⟨*search* [[**prototype**]] 555a⟩ implements a Byrd box.

We go through three examples, two based upon `member` and then the `coloring` query. Recall the definition of `member`:

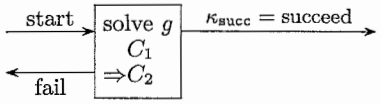
```
member(X, [X|M]).                    /* C1 */
member(X, [Y|M]) :- member(X, M). /* C2 */
```

To answer the query $g =$ `member(3, [4, 3]))`, the procedural interpretation directs us to follow these steps:

1. We create a Byrd box that is prepared to consider clauses C_1 and C_2.[5]



The goal does not unify with C_1, so the Byrd box changes state to look at C_2:



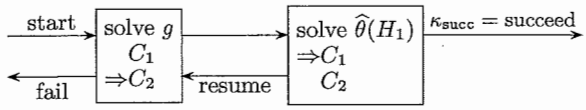
The goal g does unify with C_2 via substitution

$$\theta = \{\texttt{X1} \mapsto 3,\ \texttt{Y1} \mapsto 4,\ \texttt{M1} \mapsto [3]\}$$

(where `X`, `Y`, and `M` in C_2 have been renamed to `X1`, `Y1`, and `M1`). This produces a new subgoal, $\widehat{\theta}(H_1)$, which is `member(3, [3])`.

[5]The semantics actually require that we consider all clauses, but these are the only clauses that could possibly unify with query g.

2. We are now obliged to prove $\widehat{\theta}(H_1)$, which is `member(3, [3])`. We start a new Byrd box. The new Byrd box gets the same success continuation as the current Byrd box. If the new goal fails, we'll continue looking for clauses after C_2.



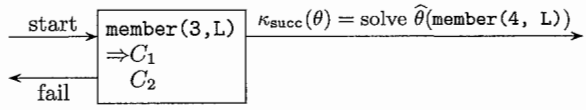
C_1 applies, via $\{$`X2` $\mapsto$ `3`, `L2` $\mapsto$ `nil`$\}$ (renaming `X` and `L` in C_1 to `X2` and `L2`). As C_1 has no subgoals, we succeed without further ado.

This example does not produce any substitution, since the goal has no variables.

Our next query involves "running the program backwards":

```
member(3, L), member(4, L).
```

1. We attempt first to satisfy `member(3, L)`. If the attempt succeeds, we plan to continue by solving `member(4, L)`.

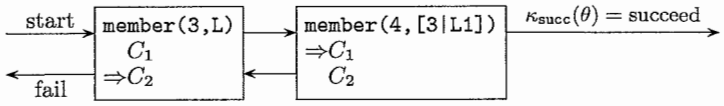


C_1 applies, yielding

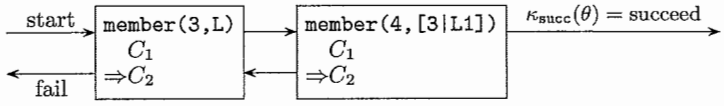
$$\theta_0 = \{\texttt{X1} \mapsto \texttt{3},\ \texttt{L} \mapsto \texttt{[3|L1]}\}$$

(renaming `X` and `L` in C_1 to `X1` and `L1`). We pass θ_0 to κ_{succ}.

2. Now we create a new Byrd box to solve $\widehat{\theta_0}$(`member(4,L)`), which is `member(4, [3|L1])`. If that fails, we will resume the current box at C_2.



C_1 does not apply, because its head `member(X, [X|L])` does not match the goal `member(4,[3|L1])`. The current box moves to C_2.



This example illustrates a general property of Byrd boxes: only the rightmost box is active at any one time.

Continuing, we see that the head of C_2 does match the goal, via

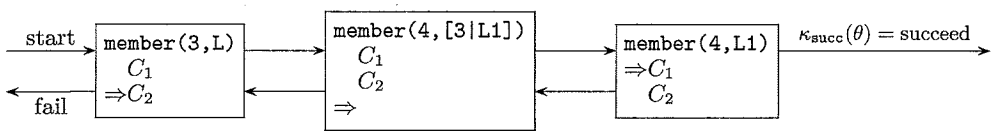
$$\theta_0' = \{\texttt{X2} \mapsto \texttt{4},\ \texttt{Y2} \mapsto \texttt{3},\ \texttt{M2} \mapsto \texttt{L1}\}.$$

(renaming `X`, `Y`, and `M` in C_2 to `X2`, `Y2`, and `M2`).

3. We can't simply pass θ_0' to κ_{succ}; first we must prove the right-hand side of C_2, which, applying the substitution, is:

$$\widehat{\theta_0'}(\texttt{member(X2, M2)}) = \texttt{member(4, L1)}.$$

Again, we create a new box.



C_1 applies, yielding[6] $\theta_1' = \{\texttt{X3} \mapsto \texttt{4}, \texttt{L1} \mapsto \texttt{[4|L3]}\}$.

4. The attempt to satisfy `member(4, [3|L1])` (step 2) has now succeeded with substitution

$$\theta_1 = \theta_1' \circ \theta_0' = \{\texttt{X3} \mapsto \texttt{4}, \texttt{L1} \mapsto \texttt{(cons 4 L3)}, \texttt{X2} \mapsto \texttt{4}, \texttt{Y2} \mapsto \texttt{3}, \texttt{M2} \mapsto \texttt{(cons 4 L3)}\}.$$

5. The original goal is satisfied with

$$\theta_1 \circ \theta_1 = \{\texttt{X1} \mapsto \texttt{3}, \texttt{X3} \mapsto \texttt{4}, \texttt{L1} \mapsto \texttt{[4|L3]}, \texttt{L} \mapsto \texttt{[3,4|L3]}, \ldots\}.$$

The `mapb_coloring` query allows us to illustrate backtracking within right-hand sides, without complicated unification and variable renaming. The computation that solves the query $\texttt{mapb\_coloring}(A, B, C, D, E, F)$ is in fact quite long, so we show only the first few steps. Of course, only one clause applies to this goal, and it leaves us with these subgoals (ignoring the renaming of variables):

$$\texttt{different}(A, B), \texttt{different}(A, C), \texttt{different}(A, D), \ldots$$

These are the steps to be followed according to the procedural interpretation:

1. $\texttt{different}(A, B)$ is immediately satisfied with $\theta_1 = \{\texttt{A} \mapsto \texttt{yellow}, \texttt{B} \mapsto \texttt{blue}\}$.
2. $\widehat{\theta_1}(\texttt{different}(A, C)) = \texttt{different}(\texttt{yellow}, C)$ is immediately satisfied with $\theta_2 = \{\texttt{C} \mapsto \texttt{blue}\}$.
3. $\widehat{\theta_2}(\widehat{\theta_1}(\texttt{different(A, D)})) = \texttt{different(yellow, D)}$ yields $\theta_3 = \{\texttt{D} \mapsto \texttt{blue}\}$.
4. $\widehat{\theta_3}(\widehat{\theta_2}(\widehat{\theta_1}(\texttt{different(A, F)}))) = \texttt{different(yellow, F)}$ yields $\theta_4 = \{\texttt{F} \mapsto \texttt{blue}\}$.
5. $\widehat{\theta_4}(\widehat{\theta_3}(\widehat{\theta_2}(\widehat{\theta_1}(\texttt{different(B, C)})))) = \texttt{different(blue, blue)}$ cannot be satisfied.
6. Backtracking to the previous subgoal, `different(yellow, F)` is resatisfied, yielding $\theta_4' = \{\texttt{F} \mapsto \texttt{red}\}$.
7. $\widehat{\theta_4'}(\widehat{\theta_3}(\widehat{\theta_2}(\widehat{\theta_1}(\texttt{different(B, C)})))) $ still is $\texttt{different(blue, blue)}$.

[6] Here and in subsequent examples, we tacitly rename the variables occurring in clauses, by concatenating successively higher integers to their original names.

8. Backtracking, `different(yellow, F)` cannot be satisfied in any more ways, so we backtrack to the subgoal before that, `different(yellow, D)`, and resatisfy it via $\theta_3' = \{\texttt{D} \mapsto \texttt{red}\}$.
9. Analogous to 4.
10. Analogous to 5.
11. Analogous to 6.
12. Analogous to 7.
13. Backtracking, `different(yellow, F)` cannot be resatisfied. Backtracking further, `different(yellow, D)` cannot be either. Finally, we backtrack to `different(yellow, C)` and resatisfy it via with $\theta_2' = \{\texttt{C} \mapsto \texttt{red}\}$.
14. Satisfy $\widehat{\theta_2'}(\widehat{\theta_1}(\texttt{different(A, D)})) = \texttt{different(yellow, D)}$, yielding $\theta_3 = \{\texttt{D} \mapsto \texttt{blue}\}$.
15. Satisfy `different(yellow, F)` via $\theta_4 = \{\texttt{F} \mapsto \texttt{blue}\}$.
16. Satisfy $\widehat{\theta_4}(\widehat{\theta_3}(\widehat{\theta_2'}(\widehat{\theta_1}(\texttt{different(B, C)})))) = \texttt{different(blue, red)}$.

Finishing this computation is up to you.

11.2.4 Primitive predicates

The primitive predicates of μProlog are `print`, `not`, `is`, `<`, `>`, `=<`, and `>=`.

`print`: Takes any number of terms as arguments, prints each of them, and succeeds.

`not`: The built-in predicate `not` takes one argument, which is interpreted as a *goal*. The Prolog search engine tries to satisfy that goal. If it is satisfiable, `not` fails; otherwise, `not` succeeds (with the identity substitution). Regrettably, the predicate `not` is not simple logical negation; one has to appeal to the procedural interpretation to understand its behavior (see Section 11.6.2).

`is`: Takes two arguments, the second of which *must* be a term that stands for an arithmetic expression. Such a term can be

- A literal integer
- A variable that is instantiated to an integer
- $e_1 \oplus e_2$, where e_1 and e_2 are terms that stand for arithmetic expressions, and $\oplus$ is one of these operators: `+`, `*`, `-`, or `/`.

It is a checked run-time error to use `is` with any other term.

The predicate `is` works as follows: it computes the value of the expression, then looks at the first argument. If the first argument is an integer, then `is` succeeds if and only if the first argument is equal to the value denoted by the second. If the first argument is a variable, then `is` succeeds and produces the substitution mapping that variable to value denoted by the second argument. If the first argument is neither an integer nor a variable, `is` fails.

544 ⟨*transcript* 525⟩+≡ ◁538a 545b▷

```
-> [query].
?- 12 is 10 + 2.
yes
?- X is 2 - 5.
X = -3
yes
?- X is 10 * 10, Y is (X + 1) / 2.
X = 100
Y = 50
yes
```

<, =<, >, >=: The built-in comparisons take two arguments, both of which must be instantiated to integers. They succeed or fail according to the way the integers compare.

The restrictions on the arguments of numeric predicates prevent infinite backtracking. Suppose `is` were defined so that `X is Y + 10` would succeed, returning some substitution $\{\texttt{X} \mapsto a, \texttt{Y} \mapsto b\}$ such that $a = b + 10$. Presumably, backtracking onto this goal would yield a different substitution $\{\texttt{X} \mapsto a', \texttt{Y} \mapsto b'\}$ with $a' = b' + 10$. Since there are an infinite number of such substitutions, `is` would be a very tricky predicate to use. In particular, any right-hand side it appeared in would have to be *guaranteed* to succeed. Otherwise, it would repeatedly backtrack into the `is` goal, which would always be resatisfied. The result would be nontermination.

11.3 Small examples

11.3.1 Lists

Although there is a close relationship between Scheme functions and Prolog predicates, Prolog supports programming idioms that are impossible in Scheme. It may be easiest to show the differences when discussing lists, since both languages build lists using `cons` and `nil`, although they are written differently. For example, in Scheme, we test list membership using a recursive function:

```
(define member (x l)
    (if (null? l) #f
       (if (= x (car l)) #t (member x (cdr l)))))
```

In Prolog, we use the (recursive) `member` predicate:

```
-> member(X, [X|M]).
-> member(X, [Y|M]) :- member(X, M).
```

The Scheme code uses `null?`, `car`, and `cdr` to examine lists, as well as `=` to compare elements. The Prolog code uses none of these! Instead, the Prolog code uses exactly the same functors—`cons` and `nil`—to *examine* lists as it does to *construct* lists. These differences lead us to two general observations about Prolog:

Pattern matching. In the Scheme code, `car` and `cdr` are used to "destruct" the list `l`, that is, to identify its components. In Prolog, they are not needed because, in unifying the goal with the left-hand side of the second `member` goal, the two parts of the list are associated with variables `Y` and `M`, respectively; the right-hand side of the rule then uses these variables to refer to the two parts of the list. Similarly, the `=` test is implicit in the unification used to invoke clauses.

Prolog was the first widely used language to provide programming by pattern matching. Today, pattern matching is very heavily used in functional languages such as ML and Haskell; we use it in our interpreters, for example.[7]

The closed-world assumption. In the Scheme code, the `null?` test is used to determine when to return false. In the Prolog code, there seems to be no clause that ever returns false. That is, the two clauses tell us only how to infer when `member` is true, not how to infer when it is false.

Indeed, this is a general rule in Prolog. The system assumes you have told it every possible way in which a goal can be proven; it takes its inability to prove a goal as proof that the goal is false. Logically speaking, this conclusion is hardly justified: being unable to prove a theorem is not equivalent to disproving it. Nonetheless, this "closed-world assumption" is so pervasive that we rarely take note of it explicitly.

We now present several familiar list operations, or rather list predicates. At first, we give the logical assertions in English before giving the code; later, we leave the translation to you. While reading, keep the logical interpretation in mind; when you need the procedural interpretation to understand a program, we point it out.

The goal `addtoend(L, X, M)`, for lists L and M and arbitrary term X, is true if M is the list obtained by adding X to the end of L.

545a ⟨*example queries of* `addtoend` 545a⟩≡ (545b) 546a▷

```
?- addtoend([3], 4, [3,4]).
yes
```

The following assertions describe `addtoend`:

- The result of adding X to the end of the empty list is $[X]$, a list of one element.
- The result of adding X to the end of $[Y|L]$ is $[Y|M]$, where M is the result of adding X to the end of L.

These two assertions are codified in Prolog as follows:

545b ⟨*transcript* 525⟩+≡ ◁544 546c▷

```
?- [rule].
-> addtoend([], X, [X]).
-> addtoend([Y|L], X, [Y|M]) :- addtoend(L, X, M).
-> [query].
⟨example queries of addtoend 545a⟩
```

[7] Compared to Prolog, functional languages provide a restricted form of pattern matching; only one of the two terms to be matched may contain variables, and no variable may appear more than once. These restrictions are a blessing, because they enable a pattern match in a functional language to be compiled into very efficient machine code, something that is not possible with Prolog's unification.

These clauses would normally be used to compute M from L and X, as in:

546a ⟨*example queries of* addtoend 545a⟩+≡ (545b) ◁545a 546b▷
```
?- addtoend([3], 4, L).
L = [3, 4]
yes
```

but they could also be used "backwards":

546b ⟨*example queries of* addtoend 545a⟩+≡ (545b) ◁546a
```
-> addtoend(L, 4, [3, 4]).
L = [3]
yes
```

reverse is defined so that goal reverse(L, M) is true if M is the reverse of L.

546c ⟨*transcript* 525⟩+≡ ◁545b 546d▷
```
?- [rule].
-> reverse([], []).
-> reverse([X|L], M) :- reverse(L, N), addtoend(N, X, M).
```

The code can be run either forward or backward; indeed, it returns the same result in both directions:

546d ⟨*transcript* 525⟩+≡ ◁546c 546f▷
```
-> [query].
?- reverse([1, 2], L).
L = [2, 1]
yes
?- reverse(L, [1, 2]).
L = [2, 1]
yes
```

append is a standard Prolog example that works out quite neatly. append(L, M, N) is true if N is the result of appending L to M, as in

546e ⟨*example queries of* append 546e⟩≡ (546f) 546g▷
```
?- append([3, 4], [5], [3, 4, 5]).
yes
```

In the forward direction, append is used to find N given L and M; the backward direction, in which a given N is split into two pieces, also has interesting applications.

546f ⟨*transcript* 525⟩+≡ ◁546d 547a▷
```
?- [rule].
-> append([], L, L).
-> append([X|L], M, [X|N]) :- append(L, M, N).
-> [query].
⟨example queries of append 546e⟩
```

Here are a forward and a backward example:

546g ⟨*example queries of* append 546e⟩+≡ (546f) ◁546e
```
?- append([3, 4], [5, 6], L).
L = [3, 4, 5, 6]
yes
?- append(L1, L2, [5, 6, 7]).
L1 = []
L2 = [5, 6, 7]
yes
```

As an idea of what can be done using append in the backward direction, here is a one-clause definition of member2, which computes the same predicate as our two-clause definition of member:

547a ⟨*transcript* 525⟩+≡ ◁546f 547b▷

```
?- [rule].
-> member2(X, L) :- append(L1, [X|L2], L).
```

Our last list example shows how Prolog allows for code reuse in a novel way. The problem is to define the equivalent of find from μScheme. We represent an association list as a lists whose elements have the form pair(*key*, *attribute*), e.g.

```
[pair(chile, santiago), pair(peru, lima)]
```

The predicate find(K, A, L) is true if attribute A is paired with key K in association list L. It can be defined in a single clause:

547b ⟨*transcript* 525⟩+≡ ◁547a 547c▷

```
-> find(K, A, L) :- member(pair(K, A), L).
```

This example also shows how to use a predicate to name a term, which is a bit like a LET binding; in this case, we associate the name capitals with the list above:

547c ⟨*transcript* 525⟩+≡ ◁547b 547d▷

```
-> capitals([pair(chile, santiago), pair(peru, lima)]).
```

Now any time we want to query the list of capitals, we begin the query with capitals(C) and then use C in the remaining goals.

547d ⟨*transcript* 525⟩+≡ ◁547c 547e▷

```
-> [query].
?- capitals(C), find(peru, CapitalOfPeru, C).
C = [pair(chile, santiago), pair(peru, lima)]
CapitalOfPeru = lima
yes
```

11.3.2 Arithmetic

We present two purely arithmetic predicates, power and fac. For reasons to be explained, the definitions of these predicates can be fully understood only by appealing to the procedural interpretation of Prolog. More precisely, the clauses themselves can be interpreted logically, but there are logically equivalent variants that don't work. To see why the variants don't work, we have to appeal to the procedural interpretation.

We give the power predicate—defined so that power(X, N, Z) is true if $Z = X^N$. This transformation is typical of Prolog; given *any* function f, we can create a corresponding relation by using the equation, e.g., f$(X, Y, Z) = W$.

Our definition of power is based two properties of exponentiation:

1. $X^0 = 1$, for any X.
2. $X^N = X \times X^{N-1}$, for any N and X.

We can express these properties in Prolog using a fact and a rule.

547e ⟨*transcript* 525⟩+≡ ◁547d 548a▷

```
?- [rule].
-> power(X, 0, 1).
-> power(X, N, Z) :- N > 0, N1 is N - 1, power(X, N1, Z1), Z is Z1 * X.
```

The subgoal `N > 0` prevents infinite recursion during backtracking.

We can use `power` in the forward direction:

548a ⟨*transcript* 525⟩+≡ ◁547e 548b▷

```
-> [query].
?- power(3, 5, X).
X = 243
yes
```

The logical interpretation certainly can explain this result, but we need the procedural interpretation to explain why `power` cannot (usually) be used in the backward direction. Consider:

548b ⟨*transcript* 525⟩+≡ ◁548a 548d▷

```
?- power(3, N, 27).
run-time error: Used comparison > on non-integer term
```

What happened? The second `power` clause matched, yielding subgoals `N > 0`, `N1 is N - 1`, and so on. But the predefined predicates `>` and `is N - 1` may be used only when `N` is instantiated to an integer.

Another consequence of the procedural interpretation (and of the definition of `is`) is that `power` will not work, even in the forward direction, if its second clause is changed as follows:

548c ⟨*bad version of* `power` 548c⟩≡

```
-> power(X, N, Z) :- N > 0, N1 is N - 1, Z is Z1 * X, power(X, N1, Z1).
```

We ask you to explain why as Exercise 5.

Our definition of factorial via the predicate `fac` is based on the standard recursive definition.

548d ⟨*transcript* 525⟩+≡ ◁548b 548e▷

```
?- [rule].
-> fac(0, 1).
-> fac(N, R) :- N1 is N - 1, fac(N1, R1), R is N * R1.
```

Predicate `fac` is non-reversible, and the order of the subgoals is critical to its proper operation. There is another subtle problem with `fac`, which we ask you to investigate as Exercise 6.

11.3.3 Sorting

In the introduction to this chapter, our example of a theorem with computational content is a theorem asserting that any list can be sorted. It can be summarized in the Prolog clause:

548e ⟨*transcript* 525⟩+≡ ◁548d 548f▷

```
?- [rule].
-> naive_sort(L, M) :- permutation(L, M), ordered(M).
```

This code can indeed be used as a sorting program in Prolog, given clauses defining `permutation` and `ordered`.

548f ⟨*transcript* 525⟩+≡ ◁548e 549a▷

```
-> ordered([]).
-> ordered([A]).
-> ordered([A, B|L]) :- A =< B, ordered([B|L]).
-> permutation([], []).
-> permutation(L, [H|T]) :- append(V, [H|U], L), append(V, U, W), permutation(W, T).
```

The definition of `ordered` is simple. As for `permutation`, the clauses say that:

- `[]` is a permutation of `[]`.
- `[H|T]` is a permutation of L if H is an element of L and T is a permutation of the remaining elements; more precisely, if L can be split into two parts, V and `[H|U]`, such that T is a permutation of the list obtained by appending U to V. This clause is another example of a backward use of `append`.

A query on `naive_sort` tries all permutations of its argument—as many as $n!$ for a list of length n—until it finds a sorted one.

549a ⟨*transcript* 525⟩+≡ ◁548f 549b▷

```
-> [query].
?- naive_sort([4, 2, 3], L).
L = [2, 3, 4]
yes
```

What an awful sorting algorithm! It is possible to define more efficient sorting algorithms in Prolog, through the usual trick of turning functions into relations. As an example, we present Quicksort.

549b ⟨*transcript* 525⟩+≡ ◁549a 549c▷

```
?- [rule].
-> partition(H, [A|X], [A|Y], Z) :- A =< H, partition(H, X, Y, Z).
-> partition(H, [A|X], Y, [A|Z]) :- H < A,  partition(H, X, Y, Z).
-> partition(H, [], [], []).
-> quicksort([], []).
-> quicksort([H|T], S) :-
      partition(H, T, A, B),
      quicksort(A, A1), quicksort(B, B1),
      append(A1, [H|B1], S).
```

The `quicksort` clauses express the algorithm very directly. The partitioning predicate `partition(H, T, A, B)` is true if `A` and `B` form a partition of `T`, with `A` containing all elements less than or equal to `H` and `B` containing all elements greater than `H`. In a functional, imperative, or object-oriented language, we would have to write a function or method returning a pair containing `A` and `B`. In Prolog, nothing prevents us from writing `partition(H, T, pair(A, B))`, but it is more idiomatic simply to make `partition` a 4-place predicate. In Prolog, using a single predicate to compute multiple values comes naturally.

A use of `partition` in the forward direction has the form `partition(`H`, `T`, `A`, `B`)`, where H and T are instantiated to a specific value and list, respectively, and A and B are variables; satisfying this goal binds values to both A and B.

We can use `quicksort` in the forward direction:

549c ⟨*transcript* 525⟩+≡ ◁549b 550a▷

```
-> [query].
?- quicksort([8, 2, 3, 7, 1], S).
S = [1, 2, 3, 7, 8]
yes
```

We leave it to you (Exercise 8) to explain why `quicksort` can't be used in the backward direction.

11.3.4 Difference lists

There is another representation of lists in Prolog that has remarkable properties; it is called the *difference list*, or *diff list*. A diff list is a term of the form `diff(`L`, `X`)`, where X is a variable and L is an ordinary list which contains as its last cdr the variable X. For example, the diff list

```
diff([3,4|X], X)
```

represents the two-element list containing 3 and 4, i.e. the ordinary list `[3, 4]`. The term `diff(X, X)` represents the empty list.

Difference lists can be transformed to ordinary lists and vice versa. The relation `simplify(`D`, `L`)` is true if L is the simple list representation of D.

550a ⟨*transcript* 525⟩+≡ ◁549c 550b▷

```
?- [rule].
-> simplify(diff(X, X), []).
-> simplify(diff([X|Y], Z), [X|W]) :- simplify(diff(Y, Z), W).
```

The definition is based on these facts:

- `diff(`X`, `X`)`, for any variable X, represents the empty list.
- `diff([`X`|`Y`], `Z`)` represents `[`X`|`W`]` if `diff(`Y`, `Z`)` represents W.

`simplify` is reversible, so it can be used to transform a simple list to a difference list. Here are examples of both kinds of uses:

550b ⟨*transcript* 525⟩+≡ ◁550a 550c▷

```
-> [query].
?- simplify(diff([3, 4|X], X), L).
X = _X5431
L = [3, 4]
yes
?- simplify(L, [3, 4]).
L = diff([3, 4|_X5658], _X5658)
yes
```

What is remarkable about difference lists is that they allow us to code some naturally recursive predicates without recursion. Several examples appear in Exercise 9; we give just one here: the difference-list version of **append**, called `diffappend`. Here is the entire definition:

550c ⟨*transcript* 525⟩+≡ ◁550b 550d▷

```
?- [rule].
-> diffappend(diff(L, X), diff(X, Y), diff(L, Y)).
```

To get some intuition for this rule, look at this algebraic identity:

$$(L - X) + (X - Y) = (L - Y).$$

Here is an example that uses `diffappend` in the forward direction to append the two singleton lists `[3]` and `[4]`:

550d ⟨*transcript* 525⟩+≡ ◁550c 559b▷

```
-> [query].
?- diffappend(diff([3|X], X), diff([4|Y], Y), Z).
X = [4|_Y5738]
Y = _Y5738
Z = diff([3, 4|_Y5738], _Y5738)
yes
```

In this example, the two goals to be unified, in order to apply the `diffappend` clause, are (after variable renaming):

```
diffappend(diff(L1,   X1),diff(X1,   Y1),diff(L1, Y1))
diffappend(diff([3|X],X), diff([4|Y],Y), Z)
```

It is not hard to see that these unify via:

$$\{\ \texttt{L1} \mapsto \texttt{[3,4|Y]},\ \texttt{X1} \mapsto \texttt{[4|Y]},\ \texttt{X} \mapsto \texttt{[4|Y]},\ \texttt{Y1} \mapsto \texttt{Y},\ \texttt{Z} \mapsto \texttt{diff([3,4|Y], Y)}\ \},$$

which gives the result.

11.4 Implementation

Our implementation of μProlog differs most obviously from our other implementations in two ways:

- There are no "values" as distinct from "abstract syntax;" terms do duty as both.
- There is no "evaluation."[8] Instead, we have queries.

The main features of the implementation are the database, substitution and unification (which are left as Exercises 15 and 16), and the backtracking query engine. To print queries and answers, we use code from Appendix J.

551a ⟨*abstract syntax and values* 529a⟩+≡ (559a) ◁530

```
termString   : term   -> string
goalString   : goal   -> string
clauseString : clause -> string
```

⟨*string conversions* 709⟩

11.4.1 The database of clauses

We treat the database of clauses as an abstraction, which we characterize by its operations.

- We can add a clause to the database.
- Given a goal, we can search for clauses whose conclusions may match that goal.

Searching for potentially matching clauses is an important part of Prolog, and it can be worth choosing a representation of the database to make this operation fast (Exercise 19). If we do so, we have to be careful to preserve the *order* of the clauses in the database.

type clause 529c

Our implementation uses a simple list. We treat *every* clause as a potential match.

551b ⟨*clause database* 551b⟩≡ (559a)

```
type database
emptyDatabase    : database
addClause        : clause * database -> database
potentialMatches : goal * database -> clause list
```

```
type database = clause list
val emptyDatabase = []
fun addClause (r, rs) = rs @ [r] (* must maintain order *)
fun potentialMatches (_, rs) = rs
```

[8] Well, hardly any. The primitive `is` does a tiny amount of evaluation.

11.4.2 Substitution and unification

Substitution

A substitution θ is a function from terms to terms. We present only prototype code for substitutions; the full implementation is left as Exercise 15.

552a ⟨*substitution on terms* **[[prototype]]** 552a⟩≡

```
exception LeftAsExercise
type subst = term -> term
infix 7 |-->
fun name |--> term = raise LeftAsExercise
fun idsubst term = term
```

```
type subst
idsubst : subst
|-->    : name * term -> subst
```

552b ⟨*substitution and unification* 552b⟩≡ (559a) 552c▷

⟨*substitution on terms* (left as an exercise)⟩

Given the ability to substitute in a term, we may also want to substitute in goals and clauses. Using the notation of Section 11.2.3, if `theta` represents θ, then `lift theta` represents $\widehat{\theta}$.

552c ⟨*substitution and unification* 552b⟩+≡ (559a) ◁552b 552d▷

```
lift       : subst -> (goal   -> goal)
liftClause : subst -> (clause -> clause)
```

```
fun lift       theta (f, l)    = (f, map theta l)
fun liftClause theta (c :- ps) = (lift theta c :- map (lift theta) ps)
```

Sets of variables

As in Chapter 7, we use sets of variables in unification. We provide a simple implementation using lists. The double prime in the type variable `''a` means that `''a` has to be an "equality type," i.e., variable `''a` may be instantiated only at types that admit equality.

552d ⟨*substitution and unification* 552b⟩+≡ (559a) ◁552c 553a▷

```
type 'a set = 'a list
val emptyset = []
fun member x =
  List.exists (fn y => y = x)
fun insert (x, l) =
  if member x l then l else x::l
fun diff  (s1, s2) =
  List.filter (fn x => not (member x s2)) s1
fun union (s1, s2) = s1 @ diff (s2, s1)   (* preserves order *)
```

```
type 'a set
emptyset : 'a set
member   : ''a -> ''a set -> bool
insert   : ''a     * ''a set  -> ''a set
union    : ''a set * ''a set  -> ''a set
diff     : ''a set * ''a set  -> ''a set
```

The function `freevars` computes the free variables of a term. For readability, we ensure that variables appear in the set in the order of their first appearance in the type, when reading from left to right. We provide similar functions on goals and clauses.

553a ⟨*substitution and unification* 552b⟩+≡ (559a) ◁552d 553b▷

```
freevars       : term   -> name set
freevarsGoal   : goal   -> name set
freevarsClause : clause -> name set
```

```
fun freevars t =
  let fun f(VAR v,     l) = insert (v, l)
        | f(LITERAL _, l) = l
        | f(APPLY(_, args), l) = foldl f l args
  in  rev (f(t, []))
  end
fun freevarsGoal goal = freevars (APPLY goal)
fun freevarsClause (c :- ps) =
  foldl (fn (p, f) => union (freevarsGoal p, f)) (freevarsGoal c) ps
```

"Freshening"

Every time we use a clause, the semantics of Prolog require that we rename its variables to fresh variables. To rename a variable, we put an underscore in front of its name and a unique integer after it. Because the parser in Section J.3 does not accept variables whose names begin with an underscore, these names cannot possibly collide with the names of variables written by users.

553b ⟨*substitution and unification* 552b⟩+≡ (559a) ◁553a 553c▷

```
freshVar : string -> term
```

```
local
  val n = ref 1
in
  fun freshVar s = VAR ("_" ^ s ^ Int.toString (!n) before n := !n + 1)
end
```

Function `freshen` replaces free variables with fresh variables. Value `renaming` represents a renaming θ_α, as in Section 11.2.3.

553c ⟨*substitution and unification* 552b⟩+≡ (559a) ◁553b 554▷

```
freshen : clause -> clause
```

```
fun freshen c =
  let val free     = freevarsClause c
      val fresh    = map freshVar free
      val renaming =
        ListPair.foldl (fn (free, fresh, theta) => theta o (free |--> fresh))
        idsubst (free, fresh)
  in  liftClause renaming c
  end
```

APPLY	529a
idsubst	823
insert	552d
liftClause	552c
LITERAL	529a
union	552d
VAR	529a
|-->	823

Unification

Given goals g and G, `unify`(g, G) returns a substitution θ such that $\widehat{\theta}(g) = \widehat{\theta}(G)$, or if no such substitution exists, it raises `Unify`. The implementation of unification is left as Exercise 16.

553d ⟨*unification* [[**prototype**]] 553d⟩≡

```
unify     : goal      * goal      -> subst
unifyTerm : term      * term      -> subst
unifyList : term list * term list -> subst
```

```
exception Unify
fun unifyTerm (t1, t2) =
  raise LeftAsExercise
and unifyList (ts1, ts2) =
  raise LeftAsExercise
fun unify ((f, args), (f', args')) =
  if f = f' then unifyList (args, args') else raise Unify
```

554 ⟨*substitution and unification* 552b⟩+≡ (559a) ◁553c
⟨*unification* (left as an exercise)⟩

11.4.3 Backtracking search

We implement Prolog search using continuation-passing style. Given a goal g and continuations $\kappa_{\texttt{succ}}$ and $\kappa_{\texttt{fail}}$, `solveOne` g $\kappa_{\texttt{succ}}$ $\kappa_{\texttt{fail}}$ builds and runs a Byrd box for g. As expected for continuation-passing style, the result of the call to `solveOne` is the result of the entire computation.

Unless the predicate is built in, `solveOne` uses internal function `search` to manage the state of the Byrd box. Think of the argument to `search` as the list of clauses to be considered; the ⇒ arrow in Section 11.2.3 points to the head of this list.[9]

To solve a single goal g, using clause G `:-` $H_1, \ldots, H_m$, we rename variables, unify the renamed G with g to get θ, then solve $\widehat{\theta}(H_1), \ldots, \widehat{\theta}(H_m)$. Eventually, the entire composed substitution gets passed to $\kappa_{\texttt{succ}}$. In the code, $G =$ `conclusion` and $H_1, \ldots, H_m =$ `premises` (both after renaming), and $g =$ `goal`.

To solve multiple goals $g_1, \ldots, g_n$, we call `solveMany` $[g_1, \ldots, g_n]$ θ_{id} $\kappa_{\texttt{succ}}$ $\kappa_{\texttt{fail}}$, where θ_{id} is the identity substitution. Function `solveMany` manages interactions between Byrd boxes, composing substitutions as it goes. If substitution θ' solves goal g_1, we apply θ' to the remaining goals $g_2, \ldots, g_n$ before a recursive call to `solveMany`. If that recursive call fails, we transfer control to the `resume` continuation that came from solving g_1, which gives us a chance to produce a *different* substitution that might solve the whole lot.

[9] Clauses preceding the ⇒ arrow are irrelevant to any future computation, and `search` discards them.

Here is the code:

555a ⟨*search* [[**prototype**]] 555a⟩≡

```
query  : database -> goal list -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
solveOne  : goal              -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
solveMany : goal list -> subst -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
search    : clause list -> 'a
```

```
fun 'a query database =
  let val builtins = foldl (fn ((n, p), rho) => bind (n, p, rho))
                       emptyEnv (⟨primops :: 557c⟩ [])
      fun solveOne (goal as (func, args)) succ fail =
            find(func, builtins) args succ fail
            handle NotFound _ =>
              let fun search [] = fail ()
                    | search (clause :: clauses) =
                        let fun resume() = search clauses
                            val conclusion :- premises = freshen clause
                            val theta = unify (goal, conclusion)
                        in  solveMany (map (lift theta) premises) theta succ resume
                        end
                        handle Unify => search clauses
              in  search (potentialMatches (goal, database))
              end
      and solveMany []            theta succ fail = succ theta fail
        | solveMany (goal::goals) theta succ fail =
            solveOne goal
            (fn theta' => fn resume =>
               solveMany (map (lift theta') goals) (theta' o theta) succ resume)
            fail
  in  fn gs => solveMany gs (fn x => x)
  end
```

The environment `builtins` holds the built-in predicates. Because the implementations of these predicates are polymorphic ML functions, ML's "value restriction" prevents us from defining them at top level. We therefore have to build the `builtins` environment dynamically, but as long as we do this only once per query, the cost should be acceptable.

11.4.4 Processing clauses and queries

Where another interpreter might use an environment, the μProlog interpreter uses a database of clauses.

555b ⟨*evaluation* 555b⟩≡ (559a) 557b▷

```
evalcq : bool -> cq * database * (string->unit) -> database
```

```
fun evalcq prompt (t, database : database, echo) =
  case t
    of USE filename => (⟨read from file filename 556a⟩) : database
     | ADD_CLAUSE c => addClause (c, database)
     | QUERY gs     => (⟨query goals gs against database 556b⟩; database)
```

ADD_CLAUSE 530
addClause 551b
type database 551b
QUERY 530
USE 530

To read from a file, we read without prompting. If we fail to open the file, we try adding ".P" to the name; this is the convention used by XSB Prolog, a free Prolog interpreter available from the State University of New York at Stony Brook. If this too fails, we try adding ".pl" to the name; this is the convention used by GNU Prolog.

556a ⟨*read from file* filename 556a⟩≡ (555b)

```
let fun foldOverFileStream tx evalPrint rho filename =
      let val fd = TextIO.openIn filename
          val stream = (tx (filename, streamOfLines fd))
          val cq = (print "getting first cq\n"; streamGet stream)
          val _ = app print ["got ", if isSome cq then "something" else "nothing", "\n"]
      in  streamFold evalPrint rho stream
          before TextIO.closeIn fd
      end
    fun writeln s = app print [s, "\n"]
    fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")

    val try = foldOverFileStream (prologReader false RMODE)
                                 (evalPrint false (writeln, errorln))
                                 database

in  try filename          handle IO.Io _ =>
    try (filename ^ ".P") handle IO.Io _ =>
    try (filename ^ ".pl")
end
```

To issue a query, we provide success and failure continuations. The success continuation uses `showAndContinue` to decide if it should resume the search, looking for another solution, or just declare victory and stop.

556b ⟨*query goals* gs *against* database 556b⟩≡ (555b)

```
query database gs
  (fn theta => fn resume =>
     if showAndContinue prompt theta gs then resume() else print "yes\n")
  (fn () => print "no\n")
```

database 555b
evalPrint 557b
filename 555b
prologReader 718a
prompt 555b
query 826
RMODE 716b
showAndContinue 557a
streamFold 650a
streamGet 647b
streamOfLines 648b

To show a solution, we apply the substitution to the free varables of the query. If we're prompting, we wait for a line of input. If the line begins with a semicolon, we continue; otherwise we quit. If we're not prompting, we're in batch mode, and we produce at most one solution.

557a ⟨*interaction* 557a⟩≡ (559a)

```
showAndContinue : bool -> subst -> goal list -> bool
```

```
fun showAndContinue prompt theta gs =
  let fun showVar v = app print [v, " = ", termString (theta (VAR v))]
      val vars = foldr union emptyset (map freevarsGoal gs)
  in  case vars
        of [] => false
         | h :: t => ( showVar h
                     ; app (fn v => (print "\n"; showVar v)) t
                     ; if prompt then
                         case Option.map explode (TextIO.inputLine TextIO.stdIn)
                           of SOME (#";" :: _) => (print "\n"; true)
                            | _ => false
                       else
                         (print "\n"; false)
                     )
  end
```

11.4.5 The read-eval-print loop

The read-eval-print loop is much the same as in other interpreters, except it takes an additional argument that tells whether to prompt. You may have noticed this argument in ⟨*read from file* `filename` 556a⟩, which arranges *not* to prompt inside `USE`.

557b ⟨*evaluation* 555b⟩+≡ (559a) ◁555b

```
evalPrint : bool -> (string->unit) * (string->unit) -> cq * database -> database
```

```
and evalPrint prompt (echo, errmsg) (cq, database) =
  let fun continue msg = (errmsg msg; database)
  in  evalcq prompt (cq, database, echo)
      handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
      ⟨more read-eval-print handlers 222c⟩
  end
```

11.4.6 Primitives

Here are the built-in predicates.

Printing a term always succeeds and produces the identity substitution.

557c ⟨*primops* :: 557c⟩≡ (555a) 558c▷

```
("print", fn args => fn succ => fn fail =>
            ( app (fn x => (print (termString x); print " ")) args
            ; print "\n"
            ; succ (fn x => x) fail
            )) ::
```

emptyset 552d
evalcq 555b
freevarsGoal 553a
termString 709
union 552d
VAR 529a

To implement the primitive predicate `is`, we need a very small evaluator. Because it works only with integers, never with variables, there is no environment; it's just a tree walk.

558a ⟨*primitives* 558a⟩≡ (559a) 558b▷

```
exception RuntimeError of string
fun eval (LITERAL n) = n
  | eval (APPLY ("+", [x, y])) = eval x  +  eval y
  | eval (APPLY ("*", [x, y])) = eval x  *  eval y
  | eval (APPLY ("-", [x, y])) = eval x  -  eval y
  | eval (APPLY ("/", [x, y])) = eval x div eval y
  | eval (APPLY ("-", [x]))    = ~ (eval x)
  | eval (APPLY (f, _))        =
      raise RuntimeError (f ^ " is not an arithmetic predicate " ^
                          "or is used with wrong arity")
  | eval (VAR v) = raise RuntimeError ("Used uninstantiated variable " ^ v ^
                                       " in arithmetic expression")
```

```
eval : term -> int
```

We implement t_1 `is` t_2 by evaluating term t_2 as an integer expression and unifying t_1 with the result.

558b ⟨*primitives* 558a⟩+≡ (559a) ◁558a 558d▷

```
fun is [x, e] succ fail = (succ (unifyTerm (x, LITERAL (eval e))) fail
                           handle Unify => fail())
  | is _      _    fail = fail ()
```

558c ⟨*primops :: 557c*⟩+≡ (555a) ◁557c 558e▷

```
("is", is) ::
```

The parser should arrange that each comparison is applied to exactly two arguments. If these arguments aren't integers, it's a run-time error.

558d ⟨*primitives* 558a⟩+≡ (559a) ◁558b

```
fun compare name cmp [LITERAL n, LITERAL m] succ fail =
      if cmp (n, m) then succ idsubst fail else fail ()
  | compare name _ [_, _] _ _ =
      raise RuntimeError ("Used comparison " ^ name ^ " on non-integer term")
  | compare name _ _ _ _ =
      raise RuntimeError ("this can't happen---non-binary comparison?!")
```

558e ⟨*primops :: 557c*⟩+≡ (555a) ◁558c 558f▷

```
("<",  compare "<"  op < ) ::
(">",  compare ">"  op > ) ::
("=<", compare "=<" op <= ) ::
(">=", compare ">=" op >= ) ::
```

APPLY 529a
idsubst 823
LITERAL 529a
Unify 824
unifyTerm 824
VAR 529a

Two more predicates, `!` and `not`, cannot be implemented using this technique; they have to be added directly to the interpreter (Exercises 20 and 21). Here we ensure that they can't be used by mistake.

558f ⟨*primops :: 557c*⟩+≡ (555a) ◁558e

```
("!",   fn _ => raise RuntimeError "The cut (!) must be added to the interpreter") ::
("not", fn _ => raise RuntimeError "The predicate 'not' must be added to the interpreter") ::
```

11.4.7 Putting the pieces together

We stitch together the parts of the implementation in this order:

559a ⟨*upr.sml* 559a⟩≡
⟨*environments* 214⟩
⟨*abstract syntax and values* 529a⟩
⟨*clause database* 551b⟩
⟨*substitution and unification* 552b⟩
⟨*lexical analysis* 710b⟩
⟨*parsing* 714a⟩
⟨*implementation of* `use` 222a⟩
⟨*primitives* 558a⟩
⟨*tracing functions* 583⟩
⟨*search* (left as an exercise)⟩
⟨*interaction* 557a⟩
⟨*evaluation* 555b⟩
⟨*Prolog command line* 719d⟩

11.5 Larger example—The blocks world

The "blocks world" is a classic problem-solving domain in artificial intelligence. It is a simple test bed for ideas in planning robot motions. The "world" consists of a table and three children's blocks. The important information about the state of the world is how the blocks are stacked. The problem is: given an existing state and a desired new state, move the blocks to achieve that state. The allowable "moves" are those that move a single block from whatever it is on (another block or the table) to something else. Our approach to this problem closely follows that of Sterling and Shapiro (1986, p. 222).

We can formulate the problem easily. We represent a state as a list of the form

```
[on(a, a), on(b, b), on(c, c)],
```

where a, b, and c are either `table` or one of the blocks `a`, `b`, and `c`. Here are some predicates outlining the basic parameters of the problem. The blocks are `a`, `b`, and `c`.

559b ⟨*transcript* 525⟩+≡ ◁550d 559c▷

```
?- [rule].
-> block(a).
-> block(b).
-> block(c).
```

The blocks are all different.

559c ⟨*transcript* 525⟩+≡ ◁559b 559d▷

```
-> different(a, b).  different(b, a).
-> different(a, c).  different(c, a).
-> different(b, c).  different(c, b).
```

The table is different from any block.

559d ⟨*transcript* 525⟩+≡ ◁559c 560a▷

```
-> different(X, table) :- block(X).
-> different(table, Y) :- block(Y).
```

Given a state of the blocks world, a surface is clear in that state if there is no item on that surface.

560a ⟨*transcript* 525⟩+≡ ◁559d 560b▷

```
-> clear(Surface, []).
-> clear(Surface, [on(Block, S2)|State]) :-
     different(Surface, S2), clear(Surface, State).
```

We define a shorthand for deciding if an item is on a surface.

560b ⟨*transcript* 525⟩+≡ ◁560a 560c▷

```
-> on(X, Y, State) :- member(on(X, Y), State).
```

We represent a move in the blocks world as a term `move(Block, From, To)`. The surfaces `From` and `To` include the blocks as well as the table. There are no clauses involving only `move`, since `move` is not a predicate. To define a predicate, we show how a move changes state; `update(M, S1, S2)` is satisfied if performing move `M` in state `S1` leaves the blocks world in state `S2`.

560c ⟨*transcript* 525⟩+≡ ◁560b 560d▷

```
-> update(move(Block, From, To), [on(Block, From)|S], [on(Block, To)|S]).
-> update(move(Block, From, To), [on(B2, Surface)|S1], [on(B2, Surface)|S2]) :-
     different(Block, B2), update(move(Block, From, To), S1, S2).
```

The predicate `update` captures only the state-changing aspects of moving blocks. But not every move is permissible in the blocks world.

- We can't move a block if something is on top of it.
- There can be at most one block on top of any other block.
- We can't put a block on top of itself; that's physics.
- Finally, we don't want to "move" a block unless `From` and `To` are different, since we could perform an infinite number of such "moves" without actually changing the state of the world.

We use the predicate `legal_move` to select moves according to these rules. To move a block `B` to the table, we must move it from wherever it is, it must not be on the table already, and there must be nothing on top of it.

560d ⟨*transcript* 525⟩+≡ ◁560c 560e▷

```
-> legal_move(move(B, From, table), State) :-
     on(B, From, State), different(From, table), clear(B, State).
```

To move a block `B1` onto another block `B2`, we must move `B1` from wherever it is, it must not be on `B2` already, `B1` and `B2` must be different, and neither block can have anything on top of it.

560e ⟨*transcript* 525⟩+≡ ◁560d 562a▷

```
-> legal_move(move(B1, From, B2), State) :-
     block(B2), on(B1, From, State),
     different(From, B2), different(B1, B2),
     clear(B1, State), clear(B2, State).
```

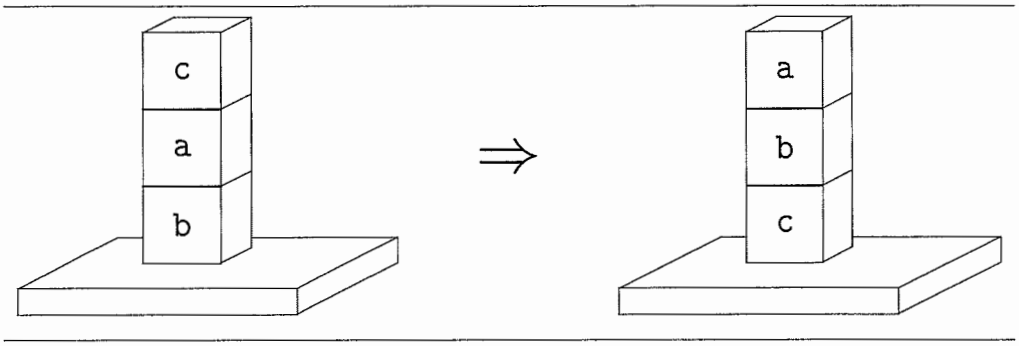


Figure 11.3: The sample transformation problem in the blocks world

The only interesting problem in the blocks world is to find a plan that gets us from one state to another, where a plan is simply a list of moves. We might imagine we could compute such a list this way:

561a ⟨*nonterminating version of* `transform` 561a⟩≡ 561b▷

```
-> transform(State, State, []).
-> transform(State1, State2, [Move|Moves]) :-
     legal_move(Move, State1),
     update(Move, State1, State),
     transform(State, State2, Moves).
```

Regrettably, this idea won't work. For example, the query:

561b ⟨*nonterminating version of* `transform` 561a⟩+≡ ◁561a

```
-> state1([on(a, b), on(b, table), on(c, a)]).
-> state2([on(a, b), on(b, c), on(c, table)]).
-> [query].
?- state1(S1), state2(S2), transform(S1, S2, Plan).
```

asks us to make the transformation pictured in Figure 11.3. It does not terminate. To see why, change the second clause of `transform`[10]:

561c ⟨*nonterminating version of* `transform`, *with debugging code* 561c⟩≡

```
-> transform(State1, State2, [Move|Moves]) :-
     legal_move(Move, State1),
     update(Move, State1, State),
     print(moved(Move, State)),
     transform(State, State2, Moves).
```

Now we can see what is going on:

561d ⟨*output from nonterminating version of* `transform`, *with debugging code* 561d⟩≡

```
moved(move(c, a, table), [on(a, b), on(b, table), on(c, table)])
moved(move(a, b, table), [on(a, table), on(b, table), on(c, table)])
moved(move(b, table, a), [on(a, table), on(b, a), on(c, table)])
moved(move(b, a, table), [on(a, table), on(b, table), on(c, table)])
moved(move(b, table, a), [on(a, table), on(b, a), on(c, table)])
 ...
```

[10]N.B.: There is no direct way to simply *redefine* a predicate, since new clauses related to a predicate do not invalidate existing clauses. Hence, "change the second clause" means "terminate the session and start over using the new clauses."

The plan cheerfully puts block b on top of block a and takes it off again, and it will do this forever. The problem is that states are repeating themselves. To protect against such repetition, we need to keep a list of the states already visited. We need a *4-argument* version of `transform`[11]. Predicate `transform(State1, State2, Visited, Plan)` holds if `Plan` leads from `State1` to `State2`, while passing through the states in `Visited`.

562a ⟨*transcript* 525⟩+≡ ◁560e 562b▷

```
-> transform(State, State, Visited, []).
-> transform(State1, State2, Visited, [Move|Moves]) :-
     legal_move(Move, State1),
     update(Move, State1, State),
     not_member(State, Visited),
     transform(State, State2, [State|Visited], Moves).
-> transform(State1, State2, Plan) :- transform(State1, State2, [], Plan).
```

The new predicate `not_member` tests membership of a state in the list `Visited` so as to avoid returning to that state.

562b ⟨*transcript* 525⟩+≡ ◁562a 562c▷

```
-> not_member(X, []).
-> not_member(X, [Y|L]) :- different(X, Y), not_member(X, L).
```

To make this code work, we must extend `different` to states.

562c ⟨*transcript* 525⟩+≡ ◁562b 562d▷

```
-> different([on(A, X)|State1], [on(A, Y)|State2]) :- different(X, Y).
-> different([on(A, X)|State1], [on(A, X)|State2]) :- different(State1, State2).
```

With these new clauses, we get:

562d ⟨*transcript* 525⟩+≡ ◁562c 563a▷

```
-> state1([on(a, b), on(b, table), on(c, a)]).
-> state2([on(a, b), on(b, c), on(c, table)]).
-> [query].
?- state1(S1), state2(S2), transform(S1, S2, Plan).
S1 = [on(a, b), on(b, table), on(c, a)]
S2 = [on(a, b), on(b, c), on(c, table)]
Plan = [move(c, a, table), move(a, b, table), move(b, table, a), move(b, a, c), move(a, table, b)]
yes
```

[11] The 4-argument predicate is analogous to an auxiliary function. It is all right to use the same name for predicates with different numbers of arguments; because goals with different numbers of arguments can never unify, the name clash can never affect a computation.

We find a working plan, but not the best plan. In particular, we find a plan that has two consecutive moves of block b, something that is never sensible. This problem can be fixed directly, but we prefer to leave it as an exercise. Another view of the difficulty is that the search for a solution is entirely "forward," that is, it starts from State1 and tries every possible move until eventually, with luck, State2 is reached. If the goal, State2, could be taken into account in looking for a plan, possibly a better plan could be found more quickly. We define transform2, which instead of choosing any legal move, uses choose_move to choose a move that takes State2 into account.

563a ⟨*transcript* 525⟩+≡ ◁562d 563b▷

```
?- [rule].
-> transform2(State, State, Visited, []).
-> transform2(State1, State2, Visited, [Move|Moves]) :-
     choose_move(Move, State1, State2),
     update(Move, State1, State),
     not_member(State, Visited),
     transform2(State, State2, [State|Visited], Moves).
-> transform2(State1, State2, Plan) :- transform2(State1, State2, [], Plan).
```

choose_move in turn uses suggest, which looks at State2 and suggests a move that will achieve a placement which occurs there. Thus, we are attempting to reason backward from the goal.

563b ⟨*transcript* 525⟩+≡ ◁563a 563c▷

```
-> choose_move(Move, State1, State2) :-
     suggest(Move, State2), legal_move(Move, State1).
-> choose_move(Move, State1, State2) :- legal_move(Move, State1).
-> suggest(move(X, Y, Z), State) :- member(on(X, Z), State).
```

This program does result in a better plan:

563c ⟨*transcript* 525⟩+≡ ◁563b

```
-> [query].
?- state1(S1), state2(S2), transform2(S1, S2, Plan).
S1 = [on(a, b), on(b, table), on(c, a)]
S2 = [on(a, b), on(b, c), on(c, table)]
Plan = [move(c, a, table), move(a, b, table), move(b, table, c), move(a, table, b)]
yes
```

11.6 Prolog as it really is

11.6.1 Syntax

While there are minor syntactic variations in different implementations of Prolog, most are more or less close to the syntax of Edinburgh Prolog (Clocksin and Mellish 1994), on which the syntax of μProlog is based.

11.6.2 Semantics

The semantics of Prolog is very much as we have presented it, except that many early implementations of Prolog left out the occurs check in unification.[12] There are some embellishments, of which we present the most important: the "cut", `not`, and `assert` and `retract`. There are many more built-in predicates, for everything from I/O to arithmetic, which we do not discuss.

The cut is a way of giving the programmer additional control over the computation by indicating where backtracking is impermissible. Specifically, the cut is written as an exclamation mark (`!`) occurring as a goal in the right-hand side of a clause such as

```
G :- H, !, I.
```

Suppose this clause is chosen to satisfy a goal g with which `G` unifies. An attempt is made to satisfy `H`. If successful, `I` is proven. If the proof of `I` succeeds, then g is proven; in this case, the cut has no part to play. If, however, the proof of `I` fails, rather than backtrack and try to re-prove `H`, the presence of the cut causes the goal g to fail immediately; this occurs even if there are further clauses that might apply to g.

An example is the definition of `not_equal`:

```
not_equal(X,Y) :- equal(X,Y), !, fail.
not_equal(X,Y).
```

where the definition of `equal` is the single clause:

```
equal(X,X).
```

The goal `not_equal(X,Y)` should be satisfied if `X` and `Y` are not equal; it should be unsatisfiable if `X` and `Y` are equal. Variables `X` and `Y` should be bound to ground terms when attempting to satisfy this goal.

Here is what happens on query `not_equal(1,2)`:

1. The first clause matches, so we try to prove `equal(1,2)`. It fails, so we backtrack and look for another clause.
2. The second clause matches, and has no subgoals, so the original goal is proven. In other words, `not_equal(1,2)` succeeds, as it should.

On the other hand, if the query is `not_equal(2,2)`:

1. The first clause matches, so we try to prove `equal(2,2)`.

[12] Implementors have omitted the occurs check purely on efficiency grounds. The idea is that without the occurs check, a term and a variable can be unified in constant time. With the occurs check, it takes time linear in the size of the term. Omitting the occurs check changes the semantics of Prolog, but for some programs, omitting the occurs check can make an otherwise quadratic algorithm linear. It is not yet settled whether or when omitting the occurs check is a good idea.

2. It succeeds, so we pass over the cut and try to prove `fail`.

3. `fail` is assumed to be a zero-argument predicate to which no clause applies. It fails.

4. We attempt to backtrack past the cut, which causes the original goal, `not_equal(2,2)`, to fail as it should.

Another use of the cut is to prevent backtracking in cases where there are no variables, so that resatisfying the goal would be pointless. For example, if `member` is expected to be used only in the forward direction, i.e. as `member(`u`,`v`)`, where u and v are ground terms, we might write its clauses like this:

```
member(X,[X|L]) :- !.
member(X,[Y|M]) :- member(X,M).
```

Now, suppose `member` is used in the subgoals of some clause. If it has been found to be true, it can only be because the first clause in its definition matched the goal. Suppose that some subgoal further along in the list of subgoals fails and control backtracks into the `member` goal. Assuming, to repeat, that its arguments are ground terms, then there is no point in trying to resatisfy it. In this case, the cut guarantees that the `member` goal will fail immediately, with no attempt to resatisfy it.

This is risky code! We have not only gotten a little performance improvement; we have dramatically changed the semantics of `member`. In the new semantics, a backwards query such as `member(X, [1, 2, 3])` will have unexpected effects. Unfortunately, it is very difficult to make some Prolog programs perform well without the cut.

Predicate `not_equal` is just one example of negation. In the blocks world there was another, the predicate `not_member`. We could have defined `not_member` by the same "cut-fail" method:

```
not_member(X,Y) :- member(X,Y), !, fail.
not_member(X,Y).
```

As an abbreviation for this idiom, Prolog includes the predicate `not`, with which `not_member` could be written

```
not_member(X,Y) :- not(member(X,Y)).
```

The predicate `not` is no ordinary predicate: its argument is a *goal*, not a term. What it does, as implied by the clauses it abbreviates, is to try to satisfy its argument, and to succeed if its argument fails. This behavior is called "negation as failure," and it is really just another name for the closed-world assumption.

Prolog's `not` must be used with care, because it is different from the intuitive idea of negation. For example, suppose we wanted to satisfy the goal `not(member(X,[2,4,6]))`. The logical interpretation is that we want to find some `X` for which this predicate holds. However, what happens operationally is that the system attempts to satisfy `member(X,[2,4,6])`, *succeeds*, and therefore the goal fails. Thus, `not` should be applied only to a goal whose variables have been instantiated by ground terms.

It is easy to add both the cut and `not` to μProlog; see Exercises 20 and 21.

Two more significant additions to Prolog are the predicates `assert` and `retract`, which allow a program to modify the database of clauses. Each of these predicates takes a clause as its argument. They do the following:

- `assert(`C`)` places C into the global list of clauses, at an undetermined position. (More commonly used are the variants `asserta` and `assertz`, which place the new clause at the beginning and end, respectively, of the clauses database.)
- `retract(`C`)` removes the first clause that matches C from the clauses database.

Any new clause can be added, or old clause removed, by these predicates, but they are most commonly used with ground predicates, to give the effect of global variables. The simplest example is using them to define a global counter, call it `ctr`. It can be set to zero by asserting:

```
ctr(0).
```

Then, the predicate `zeroCtr` resets it to zero, and `incrCtr` adds one to it:

```
zeroCtr :- retract(ctr(X)), assert(ctr(0)).
incrCtr :- retract(ctr(X)), Y is X+1, assert(ctr(Y)).
```

The extensions mentioned here, and many more in real implementations of Prolog, can be understood only via the procedural interpretation. Indeed, a common criticism of Prolog is that the logical interpretation often fails to explain real programs, yet at the same time the procedural interpretation can be difficult to understand, even with the help of the Byrd boxes. Serious Prolog programmers know that they can't treat Prolog as simple first-order logic; they expect to use such non-logical features as the cut.

11.7 Prolog and mathematical logic

Mathematical logic is the study of the meaning of "proof" in mathematics. It has been studied heavily since about the mid-19th century, when mathematicians began to be concerned about the soundness of mathematical reasoning. Logicians have tried to formalize the language that mathematicians use and the reasoning processes by which theorems are deduced.

When computer scientists began to explore the limits of machine intelligence, one of their first goals was to develop programs that displayed mathematical reasoning. The formalizations developed by logicians were tailor-made. The creators of Prolog were researchers in the field of *mechanical theorem proving*. In this section, we review some of the background knowledge possessed by those men, placing Prolog into the overall framework of mechanized logic.

We first discuss the simplest kind of logic, *propositional logic*, and then *first-order predicate logic*, probably the most studied of all logical systems and the one that forms the basis of almost all work in mechanical theorem-proving. Prolog corresponds to a subset of first-order logic called *Horn clause logic*.

11.7.1 Propositional logic

Propositional logic is the logic of *combinations* of elementary propositions, the truth of which are not considered to be at issue. Consider the following statements:

> Either the President or the Secretary of State prevails, but not both. When the President is sure, he prevails. When the President is unsure, the Secretary of State prevails. The Secretary of State prevailed. Therefore, the President was unsure.

Propositional logic tells us that the line of reasoning used here is proper, independent of whether the statements are actually true. In other words, if one accepts the truth of the first four statements (the *hypotheses*), then he is bound to accept the final statement (the *conclusion*). Indeed, suppose we make the following substitutions:

p for "the President is sure".
q for "the President prevails".
$\neg p$ for "the President is unsure".
r for "the Secretary of State prevails".

Then we can recast our argument in these terms:

> Either q or r, but not both. When p, q. When $\neg p$, r. r. Therefore, $\neg p$.

This reformulation points to the essential fact about propositional logic, and indeed about all formal logic: it concerns *arguments*, not *facts*. The argument just made is valid no matter what p, q, and r are, assuming only that $\neg p$ is the opposite of p. For example, the following is an equally valid argument; indeed, it is the identical argument:

> Either Hamlet is crazy, or he is pretending, but not both. If he's crazy, he will kill Polonius. If he's pretending, he will kill Claudius. He kills Polonius. Therefore, he is crazy.

A careful study of propositional logical calls for formal definitions. The set of propositions is given by the grammar:

proposition	$\longrightarrow$	`t` | `f` | propositional-variable
	|	proposition $\wedge$ proposition
	|	proposition $\vee$ proposition
	|	proposition $\Longrightarrow$ proposition
	|	$\neg$ proposition | (proposition)
propositional-variable	$\longrightarrow$	name (usually p, q, r, or s)

Here, "$\wedge$" represents "and" or *conjunction*; "$\vee$" is "or" or *disjunction*;[13] "$\Longrightarrow$" is *implication*; and "$\neg$" is *negation*. Our example becomes the proposition:

$$(((q \vee r) \wedge \neg(q \wedge r)) \wedge (p \Longrightarrow q) \wedge (\neg p \Longrightarrow r) \wedge r) \Longrightarrow \neg p.$$

[13]The symbol $\wedge$ resembles the A in "and;" $\vee$ comes from the Latin word *vel*, which means "or."

Definition 11.6 A proposition containing no propositional-variables is either true or false according to the following inductive definition:

- `t` is true.
- `f` is false.
- $P \wedge Q$ is true *iff* P is true and Q is true.
- $P \vee Q$ is true *iff* P is true or Q is true, or both.
- $P \implies Q$ is true *iff* P is false or Q is true, or both.
- $\neg P$ is true *iff* P is false.
- (P) is true *iff* P is true.

A proposition containing variables is not simply true or false, since its truth may depend upon whether the variables it contains are true or false.

Definition 11.7 A *truth-value assignment* for proposition P is a substitution θ mapping each of its variables to `t` or `f`. $\widehat{\theta}$ is defined in the obvious way (Definition 11.2) as a map from propositions to propositions. Say P *is true for* θ if $\widehat{\theta}(P)$ is true. (Note that there are a finite number of truth-value assignments for any P; specifically, there are 2^n if P has n distinct variables.)

In our sample proposition, the variables are p, q, and r, so one truth-value assignment is $\theta = \{p \mapsto \mathtt{t}, q \mapsto \mathtt{f}, r \mapsto \mathtt{t}\}$; applying $\widehat{\theta}$ yields:

$$(((\mathtt{f} \vee \mathtt{t}) \wedge \neg(\mathtt{f} \wedge \mathtt{t})) \wedge (\mathtt{t} \implies \mathtt{f}) \wedge (\neg\mathtt{t} \implies \mathtt{t}) \wedge \mathtt{t}) \implies \neg\mathtt{t}.$$

Application of definition 11.6 shows that this proposition is true. In fact, the sample proposition is true no matter what truth-value assignment is made:

Definition 11.8 A proposition P is *valid* if for any truth-value assignment θ for P, $\widehat{\theta}(P)$ is true.

There is an obvious way to test the validity of a proposition: check each of the 2^n truth-value assignments. This is the *truth-table method.* The main purpose of this section is to introduce another method to solve this problem, the method of *resolution.* First, we need more definitions.

Definition 11.9 A proposition P is *satisfiable* if there is some truth-value assignment making P true; otherwise it is *unsatisfiable.*

Theorem 11.3 P is valid if and only if $\neg P$ is unsatisfiable.

Theorem 11.4 If $P \implies Q$ is valid, and Q is unsatisfiable, then P is unsatisfiable.

By theorem 11.3, testing validity and unsatisfiability are problems of equivalent difficulty; if we can solve one, we can solve the other. An algorithm for proving unsatisfiability is called a *refutation procedure*. Resolution is a refutation procedure. It applies not to arbitrary propositions, but only to the subset of propositions that are in *conjunctive normal form*. These can be defined by the following grammar:

CNF-proposition	$\longrightarrow$	(clause) | (clause) $\wedge$ CNF-proposition
clause	$\longrightarrow$	literal | literal $\vee$ clause
literal	$\longrightarrow$	positive-literal | negative-literal
positive-literal	$\longrightarrow$	propositional-variable
negative-literal	$\longrightarrow$	$\neg$propositional-variable

It so happens that for every proposition P there is an *equivalent* CNF-proposition $\tau(P)$, equivalent in the sense that for any truth-value assignment θ, $\widehat{\theta}(P)$ is true if and only if $\widehat{\theta}(\tau(P))$ is true. We do not show how to do this translation; see any of the references on mechanical theorem proving, which are given at the end of the chapter. From now on we assume we are looking at a CNF-proposition of the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n.$$

Theorem 11.5 (*The resolution principle*) Given a CNF-proposition as above, suppose there are C_i and C_j, $i \neq j$, of the form:

$$\begin{aligned} C_i &= (p \vee l_1 \vee \ldots \vee l_m) \\ C_j &= (\neg p \vee l'_1 \vee \ldots \vee l'_{m'}) \end{aligned}$$

(possibly reordering the literals in C_i and C_j). Define a new clause:

$$C_i \Box C_j = l_1 \vee \ldots \vee l_m \vee l'_1 \vee \ldots \vee l'_{m'}.$$

Then

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \implies C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge (C_i \Box C_j)$$

is valid.

$C_i \Box C_j$ is called the *resolvent* of C_i and C_j; C_i and C_j are called its *parents*. In case $m = m' = 0$, so that $C_i \Box C_j$ is empty, it is written as "$\Box$".

Theorem 11.6 If the CNF-proposition above has $\Box$ as a resolvent of any two clauses, then it is unsatisfiable.

These two theorems, together with theorem 11.4, suggest a method for proving CNF-propositions unsatisfiable:

Definition 11.10 (*The resolution method*) To prove a CNF-proposition P unsatisfiable, do as follows: Repeatedly apply the resolution principle (choosing the parent clauses from among the clauses in P as well as the resolvents produced in previous steps) until one of the following occurs:

1. $\Box$ is derived. Stop and announce that P is unsatisfiable.
2. No new resolvents can be found. Stop and announce that P is satisfiable.

Theorem 11.7 The resolution method is *sound* in the sense that whatever result it announces is correct; it is also *complete* in the sense that if P is unsatisfiable, it is possible to derive "$\square$".

Resolution is the basis for computation in Prolog (considering here only variable-free programs). To see this, observe first that a Prolog clause

$$P \texttt{ :- } Q, \ldots, R,$$

which in propositional logic would be written

$$(Q \wedge \ldots \wedge R) \implies P,$$

is equivalent to the clause[14]

$$P \vee \neg Q \vee \ldots \vee \neg R.$$

Thus, Prolog clauses can be regarded as the clauses of CNF-propositions.

An entire Prolog program consists of a list of such clauses, $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, together with a query, $G_1, \ldots, G_m$. This program may be regarded as the proposition

$$(C_1 \wedge C_2 \wedge \ldots \wedge C_n) \implies (G_1 \wedge \ldots \wedge G_m).$$

To show this is valid is to show its negation unsatisfiable. Its negation is

$$\neg((C_1 \wedge C_2 \wedge \ldots \wedge C_n) \implies (G_1 \wedge \ldots \wedge G_m)),$$

which can be shown equivalent to[15]

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg(G_1 \wedge \ldots \wedge G_m),$$

which is equivalent to

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge (\neg G_1 \vee \ldots \vee \neg G_m).$$

We have already shown each of the Prolog clauses to be equivalent to a CNF clause, so this is a CNF-proposition. The resolution method can therefore be applied to it.

As an example, let us return to the code of section 11.2.3. Abbreviating goals by single letters ("i" for "imokay", etc.), the code was:

```
-> i :- y, h.
-> y :- t.
-> h.
-> t.
-> [query].
?- i.
```

[14] In general, $Q \implies P$ is equivalent to $P \vee \neg Q$; you may want to use truth tables to prove this.

[15] Again, this is a simple fact of propositional logic which can be proven via truth tables.

This program is equivalent to this CNF-proposition, obtained, as just explained, by negating the query and transforming each Prolog clause to the equivalent CNF clause (and numbering each one):

$$\begin{array}{ll}(C_1) & (i \vee \neg y \vee \neg h)\\ (C_2) & \wedge(y \vee \neg t)\\ (C_3) & \wedge h\\ (C_4) & \wedge t\\ (C_5) & \wedge\neg i\end{array}$$

The first step of resolution resolves C_1 and C_5:

$$(C_6) \quad \wedge(\neg y \vee \neg h)$$

Next, resolve C_6 and C_2:

$$(C_7) \quad \wedge(\neg t \vee \neg h)$$

Next, C_7 and C_4:

$$(C_8) \quad \wedge\neg h$$

Finally, C_8 and C_3:

$$(C_9) \quad \wedge\square,$$

refuting the proposition, and thereby showing the Prolog program to be valid (i.e. showing that the query can be deduced from the clauses).

Resolution can be very inefficient, as there are in general many different resolvents available at each step. One way the Prolog designers have attempted to deal with this potential inefficiency is by restricting the class of clauses that can be asserted:

Definition 11.11 A *Horn clause* is a clause containing no more than one positive-literal.

All CNF clauses derived from Prolog clauses have a single positive-literal, and zero or more negative-literals. The query contains only negative-literals. Thus, Prolog clauses are Horn clauses. Horn clauses are more amenable to resolution, because in each clause only one literal is a candidate to be cancelled by a negative literal.

11.7.2 First-order predicate logic

Propositional logic is concerned with how arguments are formed by combining basic propositions using logical connectives like disjunction and negation. However, these basic propositions are not *about* anything. First-order logic deals with assertions that describe properties of values drawn from some universe. These assertions, called *formulas*, are defined by the following grammar:

formula	$\longrightarrow$	`t` | `f` | atomic-formula
	|	formula $\wedge$ formula | formula $\vee$ formula | formula $\Longrightarrow$ formula
	|	$\neg$ formula | (formula)
	|	$\forall$ variable.formula | $\exists$ variable.formula
atomic-formula	$\longrightarrow$	predicate | (predicate term*)
term	$\longrightarrow$	variable | optr | (optr term*)
predicate	$\longrightarrow$	name (usually p, q, r, or s)
variable	$\longrightarrow$	name (usually x, y, or z)
optr	$\longrightarrow$	name (usually f or g)

The symbols $\forall$ and $\exists$ are called the *universal* and *existential* quantifiers, respectively. Formula $\forall x.(p\ x)$ should be read: "For all x, $(p\ x)$ is true," and $\exists x.(p\ x)$ as: "There is at least one x such that $(p\ x)$ is true." Some typical statements in first-order logic are:

$$(\forall x.(p\ (f\ (g\ x)))) \implies \exists y.(p\ (f\ y))$$

$$(\forall x.(p\ x)) \implies \exists x.(p\ x)$$

$$(\exists x.(p\ x)) \implies \forall x.(p\ x).$$

The definition of *validity* here is not obvious, and is somewhat too technical to present. Intuitively, the first two examples above are valid, while the third is not. In general, a formula is *valid* if it is true regardless of the meanings of the operations and predicates occurring in it. It is *satisfiable* if some meaning can be assigned to its operations and predicates so as to make it true; otherwise, it is *unsatisfiable*. It still holds true that a formula F is valid if and only if $\neg F$ is unsatisfiable.

It is also not obvious here how to prove validity or unsatisfiability, even disregarding questions of efficiency. It turns out that the notions of conjunctive normal form and resolution can be generalized to obtain a more general resolution method that is still sound and complete. First, here is the generalization of CNF to first-order logic:

CNF-formula	$\longrightarrow$	prefix matrix
prefix	$\longrightarrow$	quant*
quant	$\longrightarrow$	$\forall$variable.
matrix	$\longrightarrow$	(clause) | (clause) $\wedge$ matrix
clause	$\longrightarrow$	literal | literal $\vee$ clause
literal	$\longrightarrow$	positive-literal | negative-literal
positive-literal	$\longrightarrow$	atomic-formula
negative-literal	$\longrightarrow$	$\neg$atomic-formula

The list of universally quantified variables in the prefix of a CNF-formula must include all variables occurring in the matrix. An important consequence of this is that we are allowed to rename the variables in individual clauses, because (writing $\equiv$ for "is equivalent to"):

$$\forall x.(C_1 \wedge C_2) \equiv (\forall x.C_1) \wedge (\forall x.C_2) \equiv (\forall x.C_1) \wedge (\forall y.C_2') \equiv \forall x.\forall y.(C_1 \wedge C_2'),$$

where C_2' is C_2 with all xs replaced by ys.

The resolution method for proving unsatisfiability of CNF-formulas is based on two theorems similar to those that justified resolution of CNF-propositions. Again assume we are looking at a typical CNF-formula F of the form:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n.$$

Theorem 11.8 (*The resolution principle*[16]) Given F as above, suppose there are C_i and C_j, $i \neq j$, having no variables in common (renaming variables if necessary):

$$\begin{aligned} C_i &= (p \vee l_1 \vee \ldots \vee l_m) \\ C_j &= (\neg q \vee l'_1 \vee \ldots \vee l'_{m'}) \end{aligned}$$

(possibly reordering the literals in C_i and C_j), and p and q are unifiable via θ. Define a new clause:

$$C_i \Box C_j = (\widehat{\theta}(l_1) \vee \ldots \vee \widehat{\theta}(l_m) \vee \widehat{\theta}(l'_1) \vee \ldots \vee \widehat{\theta}(l'_{m'})).$$

Then

$$F \implies F \wedge (C_i \Box C_j).$$

Again, if $m = m' = 0$, write $C_i \Box C_j$ as "$\Box$".

Theorem 11.9 If F has $\Box$ as a resolvent of any two clauses, then it is unsatisfiable.

As for CNF-propositions, these two theorems imply a method for proving unsatisfiability of CNF-formulas: apply the resolution principle until $\Box$ can be derived.

This method can in turn be applied to prove validity of arbitrary formulas. For any formula F, a CNF-formula $\tau(F)$ can be constructed such that F is (un)satisfiable if and only if $\tau(F)$ is (un)satisfiable[17]. To prove the validity of any F, apply the resolution method to $\tau(\neg F)$.

How does this relate to Prolog? As in the propositional case, each Prolog clause can be regarded as equivalent to a CNF-formula, or, more specifically, to a Horn clause (definition 11.11). In particular,

$$G :\!- H, \ldots, I$$

is interpreted as:

$$\forall \vec{x}.(H \wedge \ldots \wedge I \implies G)$$

(where $\vec{x}$ includes all variables occurring in G, H, ..., I), which is equivalent to:

$$\forall \vec{x}.(\neg H \vee \ldots \vee \neg I \vee G).$$

[16]Technically, what we define here is *binary* resolution. *Full* resolution is more complicated, especially notationally, though the idea is the same. Binary resolution is adequate for our purposes, but it is, in principle, strictly less powerful than full resolution; in particular, the analog of the completeness result of Theorem 11.7 is true only for full resolution.

[17]Again, the complexity of the translation puts it beyond the scope of this presentation. Note that the translation is not as strong is for propositional logic: F and $\tau(F)$ are not, in general, equivalent.

The program as a whole, consisting of clauses C_1, ..., C_n and query $G_1, \ldots, G_m$ is interpreted as the assertion:

$$C_1 \wedge \ldots \wedge C_n \implies \exists \vec{z}.(G_1 \wedge \ldots \wedge G_m)$$

($\vec{z}$ containing all the variables in the G_i). Negating this yields:

$$\begin{array}{ll} & \neg(C_1 \wedge \ldots \wedge C_n \implies \exists \vec{z}.(G_1 \wedge \ldots \wedge G_m)) \\ \equiv & C_1 \wedge \ldots \wedge C_n \wedge \neg\exists \vec{z}.(G_1 \wedge \ldots \wedge G_m) \\ \equiv & C_1 \wedge \ldots \wedge C_n \wedge \forall \vec{z}.\neg(G_1 \wedge \ldots \wedge G_m) \\ \equiv & C_1 \wedge \ldots \wedge C_n \wedge \forall \vec{z}.(\neg G_1 \vee \ldots \vee \neg G_m) \end{array}$$

This is not quite a CNF-formula, because it contains embedded quantifiers:

$$= \forall \vec{x}.{C_1}' \wedge \ldots \wedge \forall \vec{y}.{C_n}' \wedge \forall \vec{z}.(\neg G_1 \vee \ldots \vee \neg G_m)$$

But we are justified in pulling all the variables to the front, so:

$$\equiv \forall \vec{x}. \ldots \forall \vec{y}. \forall \vec{z}.({C_1}' \wedge \ldots \wedge {C_n}' \wedge (\neg G_1 \vee \ldots \vee \neg G_m)),$$

which is a CNF-formula.

In summary, by simply negating the query, a Prolog program can be regarded as the CNF-formula which is the negation of the assertion made implicitly by the program. Refuting this formula is equivalent to satisfying the query. Because it is a CNF-formula, the resolution method is directly applicable.

We now give several examples illustrating this idea. Compare the resolution refutations to the Prolog computations. All the examples are based on the `member` predicate, which we abbreviate as "`mem`" throughout. We start with a simple forward use of the predicate, the query:

$$\texttt{mem}(3, [4, 3])$$

The two clauses for `member` together with this query correspond to the formula:

```
(     ∀X. ∀L. mem(X, [X|L])
∧     ∀X.∀Y.∀M. mem(X, M)  ⟹  mem(X, [Y|M]) )
⟹   mem(3, [4, 3])
```

After negating the query and transforming to conjunctive normal form (and omitting the quantifiers as redundant):

```
(1)       mem(X, [X|L])
(2)   ∧   [mem(X, [Y|M]) ∨ ¬mem(X, M)]
(3)   ∧   ¬mem(3, [4, 3])
```

We can now follow the refutation procedure:

```
(4)   ∧   ¬mem(3, [3])   (= 3□2)
(5)   ∧   □   (= 4□1)
```

Our next example is the refutation of the query:

```
mem(3, L), mem(4, L).
```

The first three clauses give the translation of the program; variables are renamed when resolving with (1) or (2):

(1) `mem(X, [X|L])`
(2) $\wedge$ $[$`mem(X, [Y|M])` $\vee \neg$`mem(X, M)`$]$
(3) $\wedge$ $[\neg$`mem(3, L)` $\vee \neg$`mem(4, L)`$]$
(4) $\wedge$ $\neg$`mem(4, [3|L1])`
$(= 3\square 1 = \widehat{\theta}(\neg$`mem(4, L)`$)$, where $\theta = \{$`X1` $\mapsto 3,$ `L` $\mapsto$ `[3|L1]`$\})$
(5) $\wedge$ $\neg$`mem(4, L1)`
$(= 4\square 2 = \widehat{\theta}(\neg$`mem(X2, M2)`$)$, where $\theta = \{$`X2` $\mapsto 4,$ `Y2` $\mapsto 3,$ `M2` $\mapsto$ `L1`$\})$
(6) $\wedge$ $\square(= 5\square 1$, where $\theta = \{$`X3` $\mapsto 4,$ `L1` $\mapsto$ `[4|L3]`$\})$

The goal has been satisfied, or rather its negation has been refuted. The "answer"—the list `L`—can now be computed by composing the unifying substitutions that were used in the refutation.

The resolution method does not prescribe any particular strategy for choosing clauses to resolve. Indeed, an implementation may incorrectly choose clauses so as to miss a proof that exists, as in this attempt to solve the same query:

(1) `mem(X, [X|L]))`
(2) $\wedge$ $[$`mem(X, [Y|M])` $\vee \neg$`mem(X, M)`$]$
(3) $\wedge$ $[\neg$`mem(3, L)` $\vee \neg$`mem(4, L)`$]$
(4) $\wedge$ $[\neg$`mem(3, M1)` $\vee \neg$`mem(4, [Y1|M1])`$]$
$(= 3\square 2$, where $\theta = \{$`X1` $\mapsto 3,$ `L` $\mapsto$ `[Y1|M1]`$\})$
(5) $\wedge$ $[\neg$`mem(3, M2)` $\vee \neg$`mem(4, [Y1,Y2|M2])`$]$
$(= 4\square 2$, where $\theta = \{$`X2` $\mapsto 3,$ `M1` $\mapsto$ `[Y2|M2]`$\})$

We can continue in this way, generating larger clauses *ad infinitum*. Thus, although resolution is in principle complete—any unsatisfiable CNF-formula can be proven so by a judicious choice of resolvents—any particular implementation may fail by repeatedly making injudicious choices. The Prolog evaluation mechanism is an example. It is Prolog's failure to provide a fully general strategy for performing resolution that accounts for the inconsistencies between the logical and procedural interpretations.

To better understand this problem, think of all possible resolutions as forming a tree. The root contains the original set of clauses, and the children of node N are labelled by the resolvents of clauses occurring at or above N. The completeness of resolution simply says that if the original set of clauses is unsatisfiable, $\square$ occurs somewhere in the tree. It remains the responsibility of the implementation to *search the tree* in such a way as to find the $\square$. A breadth-first (level-order) search would certainly suffice; a depth-first (pre-order) search might not, since it may get lost following an infinite branch.

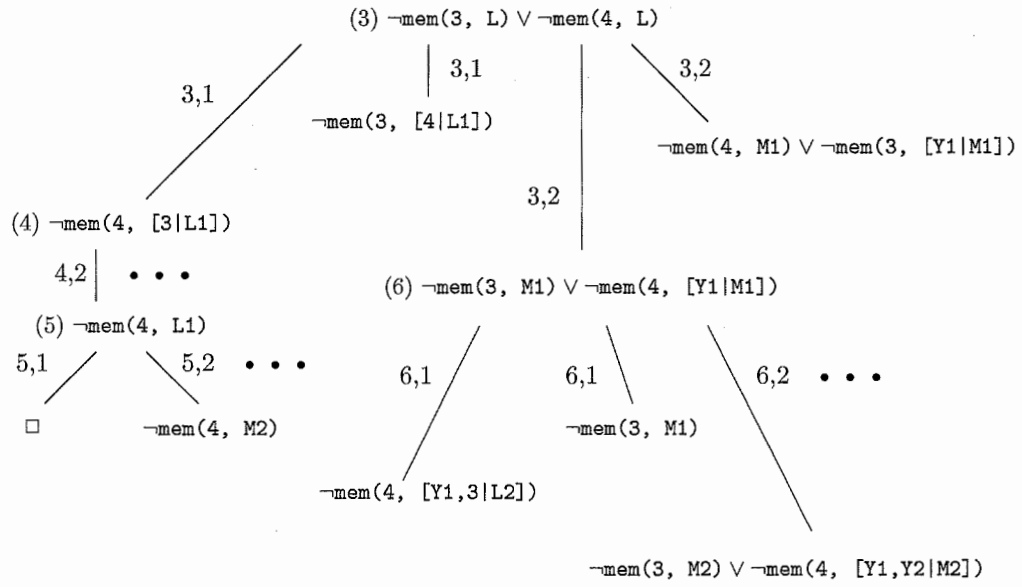


Figure 11.4: Part of a Prolog resolution search tree

Prolog evaluation can be viewed in exactly this way. The tree in question is not precisely the one just described, but a smaller one:

Definition 11.12 A *Prolog resolution search tree* for a CNF-formula $C_1 \wedge \ldots \wedge C_n \wedge Q$ (each C_i and Q Horn clauses) is defined as follows:

- The root is labelled with Q.
- The children of node N are labelled with resolvents of pairs of clauses (Q', C_i), Q' occurring in the tree at or above N, and C_i one of the clauses $C_1 \ \ldots \ C_n$.
- The order of the children of N is determined by the parents of each resolvent. Children occurring to the left are those whose first parent is the most recently-generated clause (i.e. lowest in the tree), whose second parent comes earliest in the list $C_1, \ldots, C_n$, and whose resolving literal occurs furthest to the left.

These trees are huge. A small part of the tree corresponding to the last resolution example is presented in Figure 11.4. In that tree, we have labelled each edge with the clauses resolved (it is possible to get two different resolvents from the same pair of clauses, by resolving on different literals). The ellipsis are used to indicate places where not all the children of a node are presented. All nodes have children, even when none are shown, except, of course, the one node labelled "□".

Despite the many restrictions on the form of this tree (especially the restriction on the parents of resolvents), it can be shown that it is also complete: if $C_1 \wedge \ldots \wedge C_n \wedge Q$ is unsatisfiable, the tree contains $\square$. Prolog computation can be regarded as searching for $\square$ in this tree. However, its search is depth-first, and that is why it sometimes fails to find a solution when one exists. For example, in the middle of Figure 11.4 the beginning of an infinite branch can be seen; if clauses (1) and (2) (those defining `mem`) were exchanged, this would be the leftmost branch and a depth-first traversal would follow it into oblivion. We invite you to present these re-ordered clauses and the query to the μProlog interpreter and observe that it does, indeed, go into an infinite loop; by instrumenting the interpreter, you can even see that the generated subgoals correspond exactly to those given along that branch of the tree.

11.8 Summary

Prolog is the best known exemplar of logic programming. The Prolog programmer programs by writing assertions, and the Prolog interpreter attempts to validate those assertions, in the process computing values. Prolog has attracted many avid proponents; it is probably the leading language for artificial intelligence after Lisp. It has also been the subject of research aimed at producing a more expressive logic programming language and at bringing the procedural interpretation more in line with the logical interpretation.

11.8.1 Glossary

Closed-world assumption The assumption that the clauses in a Prolog program constitute a complete picture of the world, insofar as the predicates it names are concerned. This justifies the rule of **negation as failure**, whereby a goal that cannot be proven is thereby assumed to be false, a central tenet of Prolog programming.

Conjunctive normal form, or clause form The subset of first-order formulas of the form $\forall \vec{x}.C_1 \wedge \ldots \wedge C_m$, where the C_i are **clauses** of the form $l_1 \vee \ldots \vee l_p$, each l_j an atomic formula or the negation of an atomic formula. This is a "normal form" in the sense that any first-order formula can be translated to clause form in such a way that unsatisfiability is preserved.

The cut A special Prolog goal, written as an exclamation mark (`!`), which cannot be backtracked over. That is, if it appears in the goal list of a clause, and the goals that follow it are not satisfied, it causes the goal for which the clause was invoked to immediately fail. It is used to control backtracking, primarily for efficiency, but also to implement a kind of negation, as in the definition of `not_equal`.

Difference lists Lists represented in the form

```
diff([a,b,...,X], X).
```

Such lists have many interesting uses; see Sterling and Shapiro (1986, Chapter 15) for a particularly good discussion.

First-order logic A logic that subsumes propositional logic by permitting universal and existential quantification over the elements of some domain of values.

Horn clause A clause $l_1 \vee l_2 \vee \ldots \vee l_n$ as in conjunctive normal form, with the additional restriction that at most one of the l_i is a "positive literal," i.e. an unnegated atomic proposition. The Prolog clause l_1 :- $l_2, \ldots, l_n$ is equivalent to the Horn clause $l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n$.

Without the restriction on the presence of positive literals, such clauses would be equivalent in power to arbitrary first-order formulas, as explained for conjunctive normal form. The restriction makes the search for proofs much more efficient, at the cost of a loss of expressive power.

Logic programming A programming-language concept in which programs are regarded as assertions in a logic, and computation corresponds to proving or satisfying the assertions. Logic programming languages differ in the logics on which they are based, and in the additional, non-logical, features they include. Prolog is by far the best known logic programming language, based upon the logic of Horn clauses; its most prominent non-logical feature is the cut.

Logical variable The name sometimes used for Prolog variables, to emphasize the difference between them and the variables in ordinary programming languages. Logical variables get bound to values by the unification process, whereas ordinary variables get their values by assignment or binding during function calls.

Mechanical theorem proving, or automated deduction The study of how to program computers to solve mathematical problems, usually assuming those problems are stated in first-order logic. The resolution method is the best known, but by no means the only, method in this area.

Occurs check The test made, when unifying a variable with a non-variable term, that the variable does not occur in the term. Technically, this test must be made for the unification process to be correct and for the underlying logic to be sound. However, this test is believed to be expensive, adding 10–20% or more to the cost of unification, and because the occurs check is often not needed in practice, many Prolog implementations omit it.

Propositional logic The logic that deals with the combination, by way of *logical connectives* such as $\wedge$ and $\vee$, of assertions whose internal structure is not considered.

Resolution A method of proving the unsatisfiability of first-order formulas, presented as sets of clauses. It uses a single, easily programmed proof rule—the resolution principle—which can be stated in its simplest form as: "from $(p \vee q)$ and $(\neg p \vee r)$, infer $(q \vee r)$." Resolution is a *complete* method in that any unsatisfiable first-order formula in conjunctive normal form can be refuted if the method is applied correctly, given enough time.

Unification The process whereby two terms containing variables are "matched up." The result of unification is a substitution that, applied to the two terms, makes them equal.

11.8.2 Further Reading

We often recommend going to the source, in this case articles by Colmerauer et al. (1973) and Kowalski (1974, 1979a), as well as the book by Kowalski (1979b). The very first description of Prolog was given in the paper by Colmerauer, which describes a system for processing natural language. Three articles on the early history of Prolog by Cohen (1988), Kowalski (1988), and Robinson (1983) name many more contributors, and they document the intimate connection between theorem-proving research and the development of Prolog.

For a general introduction to the language, there are a number of excellent books to choose from; the list given here is not exhaustive. The book by Kowalski, treating logic programming in general, is highly recommended. The book by Clocksin and Mellish (1994), now in its fourth edition, has long been the standard introduction to Prolog. There are other introductory texts by Hogger (1984) and Sterling and Shapiro (1986). Maier and Warren (1988) take an approach similar to our text, giving pseudo-Pascal interpreters for various subsets of Prolog. Rowe (1988) presents artificial intelligence using Prolog; he assumes no prior knowledge of Prolog.

The Byrd box was originally proposed as a conceptual tool for understanding Prolog, not as an implementation technique (Byrd 1980). Proebsting (1997) shows how to use Byrd boxes to implement Icon, another language that has backtracking built in (Griswold and Griswold 1996).

Logic programming is an active area of research, with frequent conferences and its own journals (*Theory and Practice of Logic Programming* and the *Journal of Logic and Algebraic Programming*). Many enhancements to Prolog and alternative languages have been, and continue to be, proposed. Clark and Tarnlund (1983) and DeGroot and Lindstrom (1986) give the flavor of this activity. Prolog has also been studied as a language for parallel processing; see Ringwood 1988 and the references cited there.

Automated deduction has been pursued almost as long as there have been computers; the two-volume collection edited by Siekmann and Wrightson (1983) includes the most important papers on the subject published between 1957 and 1970. The great breakthrough was the discovery of the resolution principle by Robinson (1965); most mechanical theorem-proving programs developed since have been based on it. The books by Chang and Lee (1973), Gallier (1986) and Wos et al. (1984) are highly recommended; the chapter on predicate calculus by Manna (1974) is also excellent. Maier and Warren (1988), cited above, present an extensive introduction to this subject, especially as it relates to Prolog. Along the same lines, Lloyd (1984) introduces logic and theorem proving to a Prolog programmer.

The blocks world is discussed in many books on artificial intelligence (Winograd 1972; Winston 1977; Nilsson 1980). Kowalski (1979b) gives a complete discussion in the context of logic programming. As mentioned, our approach is derived from the book by Sterling and Shapiro (1986).

11.9 Exercises

Learning about the language

1. Finish the computation of the query for map `b`, begun on page 542. Give the clauses and query for map a in Figure 11.1. What happens when you evaluate the query? Give the steps of the computation.

2. Program the following predicates on lists:

   (a) `length`—`length(`l`,` n`)` is true if list l has length n.

   (b) `remove`—`remove(`x`,` l`,` m`)` is true if list m is the result of removing all occurrences of x from l.

   (c) `insert_sort`—`insert_sort(`l`,` m`)` is true if list m is obtained from list l by insertion sorting.

   (d) `flatten`—analogous to `flatten` in Lisp.

3. A Boolean formula is a term in the following form:

   - Any variable is a formula.
   - `true` and `false` are formulas.
   - If f is a formula, the term `not(`f`)` is a formula.
   - If f_1 and f_2 are formulas, the term `and(`f_1`,` f_2`)` is a formula.
   - If f_1 and f_2 are formulas, the term `or(`f_1`,` f_2`)` is a formula.

   Write clauses for a Prolog predicate `satisfied` such that if f is a formula, the query `satisfied(`f`)` succeeds if and only if there is an assignment to f's variables such that f is satisfied. Issuing the query should also produce the assignment.

580 ⟨*exercise transcripts* 580⟩≡ 581▷

```
?- satisfied(and(A, and(B, not(C)))).
A = true
B = true
C = false
yes
```

4. Implement merge sort, by writing predicate `msort(L, S)` if `S` is a sorted version of `L`. Assume list `L` contains integers.

5. These problems relate to the predicate `power`:

   (a) Under exactly what circumstances will `power` work in the backward direction?

   (b) Explain why the version of `power` in ⟨*bad version of* `power` 548c⟩ doesn't work.

6. Consider the definition of the predicate `fac` in chunk 548d. Do queries involving `fac` always terminate? If so, prove termination. If not, give an example query that fails to terminate, explain the problem, and show how to correct it.

7. What predicates given in the text, if any, depend upon the occurs check? (Hint: the easiest way to solve this is to modify the interpreter to report when the occurs check is true.)

8. Explain why `quicksort` can't be run backwards.

9. Program the following operations on difference lists. Don't simply transform them to ordinary lists.

   (a) `diffaddtoend`

   (b) `diffreverse`

   (c) `diffquicksort`

10. The primitive predicate `print` prints a term when solved, but does nothing during backtracking. Create a predicate `backprint` which does nothing when solved, but which prints a term during backtracking. Perhaps surprisingly, `backprint` does not need to be a primitive predicate; you can write it in Prolog. Together, `print` and `backprint` make a crude tracing mechanism.

581 ⟨*exercise transcripts* 580⟩+≡ ◁580 582a▷

```
?- member(X, [1, 2, 3]), print(trying(x, X)), backprint(failed(x, X)),
   member(Y, [3, 2, 1]), print(trying(y, Y)), backprint(failed(y, Y)),
   X > Y.
trying(x, 1)
trying(y, 3)
failed(y, 3)
trying(y, 2)
failed(y, 2)
trying(y, 1)
failed(y, 1)
failed(x, 1)
trying(x, 2)
trying(y, 3)
failed(y, 3)
trying(y, 2)
failed(y, 2)
trying(y, 1)
X = 2
Y = 1
yes
```

11. These problems concern the blocks-world code:

    (a) Solve the problem of having two moves in a row that move the same block.

    (b) Change the representation of states to `state(`a`, `b`, `c`)`, where a, b, and c are as in the current representation. Modify the program accordingly.

    (c) Represent moves by `move(`x`, `y`)`, meaning "move x to y," and modify the program accordingly.

    (d) How much backtracking is done by the two versions of `transform/4` (the 4-argument `transform`) that work? In particular, how often do they attempt to satisfy a goal of the form `transform(...)`?

12. The natural numbers can be represented using the two functors `zero` and `succ`. For example, the term `succ(succ(succ(zero)))` represent the number 3. Using this representation, and not using any built-in predicates, define: `equals`, `plus`, `minus` `times`, and `div`. (Let `minus` fail if the result would not be a natural number.) Use the built-in predicates to define `print_int`, which is satisfied for any such term and prints its value as an ordinary integer:

582a ⟨*exercise transcripts* 580⟩+≡ ◁581 582c▷

```
?- print_int(succ(succ(zero))).
2
yes
```

13. The cut is different from ordinary backtracking. Write rules for two Prolog predicates that behave differently and that are identical except that one uses a cut and one doesn't. Show a query that illustrates the difference between the two predicates.

14. Throughout this book, we express operational semantics using inference rules. Since inference rules can be expressed directly in Prolog, we can easily write an interpreter based directly on the semantics. For example, consider these rules from the semantics of μML:

$$\frac{}{\langle \text{VAL}(v), \rho \rangle \Downarrow v} \qquad \text{(CONSTANT)}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \qquad v_1 = \text{BOOL}(\#\texttt{t}) \qquad \langle e_2, \rho \rangle \Downarrow v_2}{\langle \text{IF}(e_1, e_2, e_3), \rho \rangle \Downarrow v_2} \qquad \text{(IFTRUE)}$$

Let us write `eval`(e, ρ, v) for $\langle e, \rho \rangle \Downarrow v$. Then we can write these rules in Prolog:

582b ⟨*sample rules for μML evaluation* 582b⟩≡

```
eval(val(V), Rho, V).
eval(if(E1, E2, E3), Rho, V) :- eval(E1, Rho, true), eval(E2, Rho, V).
```

Write a complete set of rules of `eval` so that it forms an interpreter for μML.

582c ⟨*exercise transcripts* 580⟩+≡ ◁582a

```
?- eval(apply(val(plus), [val(2), val(2)]), [], V).
V = 4
yes
?- eval(apply(lambda([x], apply(val(plus), [var(x), var(x)])), [val(3)]), [], V).
V = 6
yes
```

Learning about the interpreter

15. Implement substitution. That is, complete the definition of function `|-->` in chunk 552a.

16. Implement unification. That is, complete the definitions of functions `unify` and `unifyList` from chunk 553d so that they find a most general unifier of two terms or of two lists of terms.

17. Modify the μProlog interpreter so that if a user tries to define a clause in which the left-hand side is a built-in predicate, the interpreter issues an error message and refuses to add the clause to the database. For example, the following rule should cause an error:

```
Z is X ^ N :- power(X, N, Z).
```

18. Create a tracing version of the interpreter that logs every entry to and exit from a Byrd box. Use the following functions:

583 ⟨*tracing functions* 583⟩≡ (559a)

```
fun logSucc goal succ rho resume =
  ( app print ["SUCC: ", goalString goal, " becomes ", goalString (lift rho goal), "\n"]
  ; succ rho resume
  )
fun logFail goal fail () =
  ( app print ["FAIL: ", goalString goal, "\n"]
  ; fail ()
  )
fun logResume goal resume () =
  ( app print ["REDO: ", goalString goal, "\n"]
  ; resume ()
  )
fun logSolve solve goal succ fail =
  ( app print ["START: ", goalString goal, "\n"]
  ; solve goal succ fail
  )
```

19. Every time it tries to satisfy a goal, our implementation of μProlog searches the *entire* database for matching clauses. More serious implementations use hash tables that are keyed on the *name* and *number of arguments* in the goal. Even without a hash table, one could cut down on searches by using

```
type database = clause list env vector,
```

where element 0 of the vector contains 0-argument predicates, element 1 contains 1-argument predicates, and so on. Use either this data structure or some other one to change the implementation of the μProlog database, and measure the resulting speedups.

goalString 710a
lift 552c

20. Add the cut to the μProlog interpreter.

    - Each Byrd box must take *three* continuations: $\kappa_{\texttt{succ}}$, $\kappa_{\texttt{fail}}$, and $\kappa_{\texttt{cut}}$. Supposing we are solving goal g_i based on the rule

      $$g \texttt{ :- } g_1, \ldots, g_n,$$

      the continuations play these roles:

      $\kappa_{\texttt{succ}}$ If we successfully satisfy $\widehat{\theta}(g_i)$, we pass θ to $\kappa_{\texttt{succ}}$. We also pass a resumption continuation so that if the solution of $g_{i+1}, \ldots, g_n$ fails, we can backtrack into g_i.

      $\kappa_{\texttt{fail}}$ If we fail to find a θ satisfying $\widehat{\theta}(g_i)$, we call $\kappa_{\texttt{fail}}()$, which is set up to backtrack to g_{i-1}.

      $\kappa_{\texttt{cut}}$ If g_i is a cut, we succeed and pass θ_{id} to $\kappa_{\texttt{succ}}$, but we *don't* pass a resumption continuation; if we backtrack into the cut, the entire goal g fails, *not* just g_i. Therefore the resumption continuation for $\kappa_{\texttt{succ}}$ must be the failure continuation for g.

    - Change the implementation of function `query` in ⟨*search* [[**prototype**]] 555a⟩ to add support for the cut. Functions `solveOne` and `solveMany` will both need an extra continuation argument $\kappa_{\texttt{cut}}$; the types of functions `search` and `query` should remain unchanged.

21. Add the primitive predicate `not` to the μProlog interpreter. You will not be able to do this simply using the existing mechanism for primitives, because implementing `not` requires a call to `solveOne`. Instead, you should treat `not` as a special case within `solveOne`.

22. Add a primitive predicate `/=` (not equal), which fails when it is applied to two identical integers or symbols and succeeds otherwise. Incorporate it into the blocks-world code, using it to replace the `different` predicate. How much of an efficiency improvement do you get?