correct deductions from the axioms. That is, the results of all predicates are valid tuples in the predicates' relations. We call this *deterministic logic programming*.

Table 9.2 defines a translation scheme T which translates any statement ⟨s⟩ in the relational kernel language into a logical formula $T(\langle s \rangle)$. Procedure definitions are translated into predicate definitions. Note that exceptions are not translated. Raising an exception signals that the normal, logical execution is no longer valid. The logical sentence therefore does not hold in that case. Proving the correctness of this table is beyond the scope of this chapter. We leave it as an interesting exercise for mathematically-minded readers.

A given logical semantics can correspond to many operational semantics. For example, the following three statements:

1. `X=Y` ⟨s⟩
2. ⟨s⟩ `X=Y`
3. **`if`** `X==Y` **`then`** ⟨s⟩ **`else fail end`**

all have the exactly same logical semantics, namely:

$$x = y \wedge T(\langle s \rangle)$$

But their operational semantics are very different! The first statement binds `X` and `Y` and then executes ⟨s⟩. The second statement executes ⟨s⟩ and then binds `X` and `Y`. The third statement waits until it can determine whether or not `X` and `Y` are equal. It then executes ⟨s⟩, if it determines that they are equal.

Writing a logic program consists of two parts: writing the logical semantics and then choosing an operational semantics for it. The art of logic programming consists in balancing two conflicting tensions: the logical semantics should be simple and the operational semantics should be efficient. All the declarative programs of Chapters 3 and 4 can be seen in this light. They are all logic programs. In the Prolog language, this has given rise to a beautiful programming style [182, 21, 139].

Deterministic append

Let us write a simple logic program to append two lists. We have already seen the `Append` function:

```
fun {Append A B}
   case A
   of nil then B
   [] X|As then X|{Append As B}
   end
end
```

Let us expand it into a procedure:

Relational statement	Logical formula
skip	true
fail	false
$\langle \mathtt{s} \rangle_1$ $\langle \mathtt{s} \rangle_2$	$T(\langle \mathtt{s} \rangle_1) \wedge T(\langle \mathtt{s} \rangle_2)$
local X **in** $\langle \mathtt{s} \rangle$ **end**	$\exists x.T(\langle \mathtt{s} \rangle)$
X=Y	$x = y$
X=f(l1:X1 ... ln:Xn)	$x = f(l_1 : x_1, ..., l_n : x_n)$
if X **then** $\langle \mathtt{s} \rangle_1$ **else** $\langle \mathtt{s} \rangle_2$ **end**	$(x = \text{true} \wedge T(\langle \mathtt{s} \rangle_1)) \vee (x = \text{false} \wedge T(\langle \mathtt{s} \rangle_2))$
case X **of** f(l1:X1 ... ln:Xn) **then** $\langle \mathtt{s} \rangle_1$ **else** $\langle \mathtt{s} \rangle_2$ **end**	$(\exists x_1, ..., x_n.x = f(l_1 : x_1, ..., l_n : x_n) \wedge T(\langle \mathtt{s} \rangle_1)) \vee (\neg \exists x_1, ..., x_n.x = f(l_1 : x_1, ..., l_n : x_n) \wedge T(\langle \mathtt{s} \rangle_2))$
proc {P X1 ... Xn} $\langle \mathtt{s} \rangle$ **end**	$\forall x_1, ..., x_n.p(x_1, ..., x_n) \leftrightarrow T(\langle \mathtt{s} \rangle)$
{P Y1 ... Yn}	$p(y_1, ..., y_n)$
choice $\langle \mathtt{s} \rangle_1$ [] ... [] $\langle \mathtt{s} \rangle_n$ **end**	$T(\langle \mathtt{s} \rangle_1) \vee ... \vee T(\langle \mathtt{s} \rangle_n)$

Table 9.2: Translating a relational program to logic

```
proc {Append A B ?C}
   case A
   of nil then C=B
   [] X|As then Cs in
      C=X|Cs
      {Append As B Cs}
   end
end
```

According to Table 9.2, this procedure has the following logical semantics:

$$\forall a, b, c.append(a, b, c) \leftrightarrow$$
$$(a = \mathtt{nil} \wedge c = b) \vee (\exists x, a', c'.a = x\,|\,a' \wedge c = x\,|\,c' \wedge append(a', b, c'))$$

The procedure also has an operational semantics, given by the semantics of the declarative model. The call:

```
{Append [1 2 3] [4 5] X}
```

executes successfully and returns `X=[1 2 3 4 5]`. The call's logical meaning is the tuple $append([1, 2, 3], [4, 5], x)$. After the execution, the tuple becomes:

$$append([1, 2, 3], [4, 5], [1, 2, 3, 4, 5])$$

This tuple is a member of the *append* relation. We see that `Append` can be seen as a logic program.

Another deterministic append

The above definition of `Append` does not always give a solution. For example, the call `{Append X [3] [1 2 3]}` should return `X=[1 2]`, which is the logically correct solution, but the program cannot give this solution because it assumes

`X` is bound to a value on input. The program *blocks*. This shows that the operational semantics is incomplete. To give a solution, we need to write a version of `Append` with a different operational semantics. To calculate `X` from the last two arguments, we change the definition of `Append` as follows:

```
proc {Append A B ?C}
   if B==C then A=nil
   else
      case C of X|Cs then As in
         A=X|As
         {Append As B Cs}
      end
   end
end
```

This version of `Append` expects its last two arguments to be inputs and its first argument to be an output. It has a *different* operational semantics as the previous version, but keeps the *same* logical semantics. To be precise, its logical semantics according to Table 9.2 is:

$$\forall a, b, c.append(a, b, c) \leftrightarrow$$
$$(b = c \wedge a = \texttt{nil}) \vee (\exists x, c', a'.c = x\,\texttt{|}\,c' \wedge a = x\,\texttt{|}\,a' \wedge append(a', b, c'))$$

This sentence is logically equivalent to the previous one.

Nondeterministic append

We have seen two versions of `Append`, with the same logical semantics but different operational semantics. Both versions return exactly one solution. But what if we want the solutions of `{Append X Y [1 2 3]}`? There are four different solutions that satisfy the logical semantics. The declarative model is deterministic, so it can only give one solution at most. To give several solutions, we can use the **`choice`** statement to *guess* the right information and then continue. This is explained in the next section.

9.3.3 Nondeterministic logic programming

We saw that the `Append` procedure in the declarative model has a logical semantics but the operational semantics is not able to realize this logical semantics for all patterns of inputs and outputs. In the declarative model, the operational semantics is deterministic (it gives just *one* solution) and directional (it works for only *one* pattern of input and output arguments). With relational programming, we can write programs with a more flexible operational semantics, that can give solutions when the declarative program would block. We call this *nondeterministic logic programming*. To see how it works, let us look again at the logical semantics of append:

$\forall a, b, c.append(a, b, c) \leftrightarrow$
$(a = \texttt{nil} \wedge c = b) \vee (\exists x, a', c'.a = x\,|\,a' \wedge c = x\,|\,c' \wedge append(a', b, c'))$

How can we write a program that respects this logical semantics and is able to provide multiple solutions for the call `{Append X Y [1 2 3]}`? Look closely at the logical semantics. There is a disjunction ($\vee$) with a first alternative ($a = \texttt{nil} \wedge c = b$) and a second alternative ($\exists x, a', c'.a = x\,|\,a' \wedge c = x\,|\,c' \wedge append(a', b, c')$). To get multiple solutions, the program should be able to pick *both* alternatives. We implement this by using the **`choice`** statement. This gives the following program:

```
proc {Append ?A ?B ?C}
   choice
      A=nil B=C
   [] As Cs X in
      A=X|As C=X|Cs {Append As B Cs}
   end
end
```

We can search for all solutions to the call `{Append X Y [1 2 3]}`:

```
{Browse {SolveAll
            proc {$ S} X#Y=S in {Append X Y [1 2 3]} end}}
```

To get one output, we pair the solutions `X` and `Y` together. This displays all four solutions:

```
[nil#[1 2 3] [1]#[2 3] [1 2]#[3] [1 2 3]#nil]
```

This program can also handle the directional cases, for example:

```
{Browse {SolveAll
            proc {$ X} {Append [1 2] [3 4 5] X} end}}
```

displays `[[1 2 3 4 5]]` (a list of one solution). The program can even handle cases where no arguments are known at all, e.g., `{Append X Y Z}`. Since in that case there are an infinity of solutions, we do not call `SolveAll`, but just `Solve`:

```
L={Solve proc {$ S} X#Y#Z=S in {Append X Y Z} end}
```

Each solution is a tuple containing all three arguments (`X#Y#Z`). We can display successive solutions one by one by touching successive elements of `L`:

```
{Touch 1 L}
{Touch 2 L}
{Touch 3 L}
{Touch 4 L}
...
```

(`{Touch N L}` is defined in Section 4.5.6; it simply traverses the first `N` elements of `L`.) This displays successive solutions:

```
nil#B#B|
   [X1]#B#(X1|B)|
      [X1 X2]#B#(X1|X2|B)|
```

```
[X1 X2 X3]#B#(X1|X2|X3|B)|_
```

All possible solutions are given in order of increasing length of the first argument. This can seem somewhat miraculous. It certainly seemed so to the first logic programmers, in the late 1960's and early 1970's. Yet it is a simple consequence of the semantics of the **choice** statement, which picks its alternatives in order. Be warned that this style of programming, while it can sometimes perform miracles, is extremely dangerous. It is very easy to get into infinite loops or exponential-time searches, i.e., to generate candidate solutions almost indefinitely without ever finding a good one. We advise you to write deterministic programs whenever possible and to use nondeterminism only in those cases when it is indispensable. Before running the program, verify that the solution you want is one of the enumerated solutions.

9.3.4 Relation to pure Prolog

The relational computation model provides a form of nondeterministic logic programming that is very close to what Prolog provides. To be precise, it is a subset of Prolog called "pure Prolog" [182]. The full Prolog language extends pure Prolog with operations that lack a logical semantics but that are useful for programming a desired operational semantics (see the Prolog section in Chapter 9). Programs written in either pure Prolog or the relational computation model can be translated in a straightforward way to the other. There are three principal differences between pure Prolog and the relational computation model:

- Prolog uses a Horn clause syntax with an operational semantics based on resolution. The relational computation model uses a functional syntax with an operational semantics tailored to that syntax.

- The relational computation model allows full higher-order programming. This has no counterpart in first-order predicate calculus but is useful for structuring programs. Higher-order programming is not supported at all in pure Prolog and only partially in full Prolog.

- The relational computation model distinguishes between deterministic operations (which do not use **choice**) and nondeterministic operations (which use **choice**). In pure Prolog, both have the same syntax. Deterministic operations efficiently perform functional calculations, i.e., it is known which arguments are the inputs and which are the outputs. Nondeterministic operations perform relational calculations, i.e., it is not known which arguments are inputs and outputs, and indeed the same relation can be used in different ways.

9.3.5 Logic programming in other models

So far we have seen logic programming in the declarative model, possibly extended with a choice operation. What about logic programming in other models? In other words, in how far is it possible to have a logical semantics in other models? To have a logical semantics means that execution corresponds to deduction, i.e., execution can be seen as performing inference and the results of procedure calls give valid tuples in a simple logical model, such as a model of the predicate calculus. The basic principle is to *enrich the control*: we extend the operational semantics, which allows to calculate new tuples in the same logical model. Let us examine some other computation models:

- Adding concurrency to the declarative model gives the data-driven and demand-driven concurrent models. These models also do logic programming, since they only change the order in which valid tuples are calculated. They do not change the content of the tuples.

- The nondeterministic concurrent model of Section 5.7.1 does logic programming. It adds just one operation, `WaitTwo`, which can be given a logical semantics. Logically, the call `{WaitTwo X Y Z}` is equivalent to $z = 1 \vee z = 2$, since `Z` is bound to 1 or 2. Operationally, `WaitTwo` waits until one of its arguments is determined. `WaitTwo` is used to manage control in a concurrent program, namely to pick an execution path that does not block.

  The nondeterministic concurrent model is interesting because it combines two properties. It has a straightforward logical semantics and it is almost as expressive as a stateful model. For example, it allows building a client/server program with two independent clients and one server, which is not possible in a declarative model. This is why the model was chosen as the basis for concurrent logic programming.

- The stateful models are another story. There is no straightforward way to give a logical meaning to a stateful operation. However, stateful models can do logic programming if the state is used in a limited way. For example, it can be encapsulated inside a control abstraction or it can be used as a parameter to part of a program. In the first case we are just enriching the control. In the second case, as long as the state does not change, we can reason as if it were constant.

- The constraint-based computation model of Chapter 12 is the most powerful model for doing logic programming that we see in this book. It gives techniques for solving complex combinatoric optimization problems. It is the most powerful model in the sense that it has the most sophisticated mechanisms both for specifying and automatically determining the control flow. From the logic programming viewpoint, it has the strongest deduction abilities.

9.4 Natural language parsing

Section 3.4.8 shows how to do parsing with a difference list. The grammar that it parses is deterministic with a lookahead of one token: it suffices to know the next token to know what grammar rule will apply. This is sometimes a very strong restriction. Some languages need a much larger lookahead to be parsed. This is certainly true for natural languages, but can also be true for widely-used formal languages (like Cobol and Fortran, see below).

The one-token lookahead restriction can be removed by using relational programming. Relational programs can be written to parse highly ambiguous grammars. This is one of the most flexible ways to do parsing. It can parse grammars with absolutely no restriction on the form of the grammar. The disadvantage is that if the grammar is highly ambiguous, the parser can be extremely slow. But if the ambiguity is localized to small parts of the input, the efficiency is acceptable.

This section gives a simple example of natural language parsing in the relational style. This style was pioneered by the Prolog language in the early 1970's. It is fair to say that Prolog was originally invented for this purpose [40]. This section only scratches the surface of what can be done in this area with the relational computation model. For further reading, we recommend [48].

Examples in Cobol and Fortran

Using relational programming to parse ambiguous grammars is quite practical. For example, it is being used successfully by Darius Blasband of Phidani Software to build transformation tools for programs written in Fortran and Cobol [19]. These two languages are difficult to parse with more traditional tools such as the Unix lex/yacc family. Let us see what the problems are with these two languages.

The problem with parsing Cobol The following fragment is legal Cobol syntax:

```
IF IF=THEN THEN THEN=ELSE ELSE ELSE=IF
```

This `IF` statement uses variables named `IF`, `THEN`, and `ELSE`. The parser has to decide whether each occurrence of the tokens `IF`, `THEN`, and `ELSE` is a variable identifier or a keyword. The only way to make the distinction is to continue the parse until only one unambiguous interpretation remains. The problem is that Cobol makes no distinction between keywords and variable identifiers.

The problem with parsing Fortran Fortran is even more difficult to parse than Cobol. To see why, consider the following fragment, which is legal Fortran syntax:

```
   DO 5 I = 1,10
   ...
 5 CONTINUE
```

This defines a loop that iterates its body 10 times, where I is given consecutive values from 1 to 10. Look what happens when the comma in the `DO` statement is replaced by a period:

```
DO 5 I = 1.10
```

In Fortran, this has the same meaning as:

```
DO5I = 1.10
```

where `DO5I` is a new variable identifier that is assigned the floating point number 1.10. In this case, the loop body is executed exactly once with an undefined (garbage) value stored in I. The problem is that Fortran allows whitespace within a variable identifier and does not require that variable identifiers be declared in advance. This means that the parser has to look far ahead to decide whether there is one token, `DO5I`, or three, `DO`, `5`, and `I`. The parser cannot parse the `DO` statement unambiguously until the . or , is encountered.

This is a famous error that caused the failure of at least one satellite launch worth tens of millions of dollars. An important lesson for designing programming languages is that changing the syntax of a legal program slightly should *not* give another legal program.

9.4.1 A simple grammar

We use the following simple grammar for a subset of English:

⟨Sentence⟩ ::= ⟨NounPhrase⟩ ⟨VerbPhrase⟩
⟨NounPhrase⟩ ::= ⟨Determiner⟩ ⟨Noun⟩ ⟨RelClause⟩ | ⟨Name⟩
⟨VerbPhrase⟩ ::= ⟨TransVerb⟩ ⟨NounPhrase⟩ | ⟨IntransVerb⟩
⟨RelClause⟩ ::= who ⟨VerbPhrase⟩ | ε

⟨Determiner⟩ ::= every | a
⟨Noun⟩ ::= man | woman
⟨Name⟩ ::= john | mary
⟨TransVerb⟩ ::= loves
⟨IntransVerb⟩ ::= lives

Here ε means that the alternative is empty (nothing is chosen). Some examples of sentences in this grammar are:

"john loves mary"
"a man lives"
"every woman who loves john lives"

Let us write a parser that generates an equivalent sentence in the predicate calculus. For example, parsing the sentence "a man lives" will generate the term `exists(X and(man(X) lives(X))` in the syntax of the relational computation model, which represents $\exists x.\text{man}(x) \wedge \text{lives}(x)$. The parse tree is a sentence in predicate calculus that represents the meaning of the natural language sentence.

9.4.2 Parsing with the grammar

The first step is to parse with the grammar, i.e., to accept valid sentences of the grammar. Let us represent the sentence as a list of atoms. For each nonterminal in the grammar, we write a function that takes an input list, parses part of it, and returns the unparsed remainder of the list. For ⟨TransVerb⟩ this gives:

```
proc {TransVerb X0 X}
   X0=loves|X
end
```

This can be called as:

```
{TransVerb [loves a man] X}
```

which parses "loves" and binds `X` to `[a man]`. If the grammar has a choice, then the procedure uses the **`choice`** statement to represent this. For ⟨Name⟩ this gives:

```
proc {Name X0 X}
   choice X0=john|X [] X0=mary|X end
end
```

This picks one of the two alternatives. If a nonterminal requires another nonterminal, then the latter is called as a procedure. For ⟨VerbPhrase⟩ this gives:

```
proc {VerbPhrase X0 X}
   choice X1 in
      {TransVerb X0 X1} {NounPhrase X1 X}
   [] {IntransVerb X0 X}
   end
end
```

Note how `X1` is passed from `TransVerb` to `NounPhrase`. Continuing in this way we can write a procedure for each of the grammar's nonterminal symbols.

To do the parse, we execute the grammar with encapsulated search. We would like the execution to succeed for correct sentences and fail for incorrect sentences. This will not always be the case, depending on how the grammar is defined and which search we do. For example, if the grammar is left-recursive then doing a depth-first search will go into an infinite loop. A *left-recursive* grammar has at least one rule whose first alternative starts with the nonterminal, like this:

⟨NounPhrase⟩ ::= ⟨NounPhrase⟩ ⟨RelPhrase⟩ | ⟨Noun⟩

In this rule, a ⟨NounPhrase⟩ consists first of a ⟨NounPhrase⟩! This is not necessarily wrong; it just means that we have to be careful how we parse with the grammar. If we do a breadth-first search or an iterative deepening search instead of a depth-first search, then we are guaranteed to find a successful parse, if one exists.

9.4.3 Generating a parse tree

We would like our parser to do more than just succeed or fail. Let us extend it to generate a parse tree. We can do this by making our procedures into functions.

For example, let us extend ⟨Name⟩ to output the name it has parsed:

```
fun {Name X0 X}
   choice
      X0=john|X  john
   [] X0=mary|X  mary
   end
end
```

When ⟨Name⟩ parses "john", it outputs the atom `john`. Let us extend ⟨TransVerb⟩ to output the predicate loves(x, y), where x is the subject and y is the object. This gives:

```
fun {TransVerb S O X0 X}
   X0=loves|X
   loves(S O)
end
```

Note that ⟨TransVerb⟩ also has two new inputs, `S` and `O`. These inputs will be filled in when it is called.

9.4.4 Generating quantifiers

Let us see one more example, to show how our parser generates the quantifiers "for all" and "there exists". They are generated for determiners:

```
fun {Determiner S P1 P2 X0 X}
   choice
      X0=every|X
      all(S imply(P1 P2))
   [] X0=a|X
      exists(S and(P1 P2))
   end
end
```

The determiner "every" generates a "for all". The sentence "every man loves mary" gives the term `all(X imply(man(X) loves(X mary)))`, which corresponds to $\forall x.\text{man}(x) \rightarrow \text{loves}(x, \text{mary})$. In the call to ⟨Determiner⟩, `P1` will be bound to `man(X)` and `P2` will be bound to `loves(X mary)`. These bindings are done inside ⟨NounPhrase⟩, which finds out what the ⟨Noun⟩ and ⟨RelClause⟩ are, and passes this information to ⟨Determiner⟩:

```
fun {NounPhrase N P1 X0 X}
   choice P P2 P3 X1 X2 in
      P={Determiner N P2 P1 X0 X1}
      P3={Noun N X1 X2}
      P2={RelClause N P3 X2 X}
      P
   [] N={Name X0 X}
      P1
   end
```

```
fun {Determiner S P1 P2 X0 X}
   choice
      X0=every|X
      all(S imply(P1 P2))
   [] X0=a|X
      exists(S and(P1 P2))
   end
end

fun {Noun N X0 X}
   choice
      X0=man|X
      man(N)
   [] X0=woman|X
      woman(N)
   end
end

fun {Name X0 X}
   choice
      X0=john|X
      john
   [] X0=mary|X
      mary
   end
end

fun {TransVerb S O X0 X}
   X0=loves|X
   loves(S O)
end

fun {IntransVerb S X0 X}
   X0=lives|X
   lives(S)
end
```

Figure 9.5: Natural language parsing (simple nonterminals)

```
fun {Sentence X0 X}
P P1 N X1 in
   P={NounPhrase N P1 X0 X1}
   P1={VerbPhrase N X1 X}
   P
end

fun {NounPhrase N P1 X0 X}
   choice P P2 P3 X1 X2 in
      P={Determiner N P2 P1 X0 X1}
      P3={Noun N X1 X2}
      P2={RelClause N P3 X2 X}
      P
   [] N={Name X0 X}
      P1
   end
end

fun {VerbPhrase S X0 X}
   choice O P1 X1 in
      P1={TransVerb S O X0 X1}
      {NounPhrase O P1 X1 X}
   [] {IntransVerb S X0 X}
   end
end

fun {RelClause S P1 X0 X}
   choice P2 X1 in
      X0=who|X1
      P2={VerbPhrase S X1 X}
      and(P1 P2)
   [] X0=X
      P1
   end
end
```

Figure 9.6: Natural language parsing (compound nonterminals)

```
   end
```

Since `P1` and `P2` are single-assignment variables, they can be passed to ⟨Determiner⟩ before they are bound. In this way, each nonterminal brings its piece of the puzzle and the whole grammar fits together.

9.4.5 Running the parser

The complete parser is given in Figures 9.5 and 9.6. Figure 9.5 shows the simple nonterminals, which enumerate atoms directly. Figure 9.6 shows the compound nonterminals, which call other nonterminals. To run the parser, feed both figures into Mozart. Let us start by parsing some simple sentences. For example:

```
fun {Goal}
   {Sentence [mary lives] nil}
end
{Browse {SolveAll Goal}}
```

The `SolveAll` call will calculate all possible parse trees. This displays:

```
[lives(mary)]
```

This is a list of one element since there is only a single parse tree. How about the following sentence:

```
fun {Goal}
   {Sentence [every man loves mary] nil}
end
```

Parsing this gives:

```
[all(X imply(man(X) loves(X mary)))]
```

To see the unbound variable `X`, choose the `Minimal Graph` representation in the browser. Let us try a more complicated example:

```
fun {Goal}
   {Sentence [every man who lives loves a woman] nil}
end
```

Parsing this gives:

```
[all(X
     imply(and(man(X) lives(X))
           exists(Y and(woman(Y) loves(X Y)))))]
```

9.4.6 Running the parser "backwards"

So far, we have given sentences and parsed them. This shows only part of what our parser can do. In general, it can take any input to `Sentence` that contains unbound variables and find all the parses that are consistent with that input. This shows the power of the **`choice`** statement. For example, let us find all sentences of three words:

```
fun {Goal}
   {Sentence [_ _ _] nil}
end
```

Executing this goal gives the following eight parse trees:

```
[all(A imply(man(A) lives(A)))
 all(B imply(woman(B) lives(B)))
 exists(C and(man(C) lives(C)))
 exists(D and(woman(D) lives(D)))
 loves(john john)
 loves(john mary)
 loves(mary john)
 loves(mary mary)]
```

See if you can find out which sentence corresponds to each parse tree. For example, the first tree corresponds to the sentence "every man lives".

The ability to compute with partial information, which is what our parser does, is an important step in the direction of constraint programming. Chapter 12 gives an introduction to constraint programming.

9.4.7 Unification grammars

Our parser does more than just parse; it also generates a parse tree. We did this by extending the code of the parser, "piggybacking" the generation on the actions of the parser. There is another, more concise way to define this: by extending the grammar so that nonterminals have *arguments*. For example, the nonterminal ⟨Name⟩ becomes ⟨Name⟩(N), which means "the current name is `N`". When ⟨Name⟩ calls its definition, `N` is bound to `john` or `mary`, depending on which rule is chosen. Other nonterminals are handled in the same way. For example, ⟨TransVerb⟩ becomes ⟨TransVerb⟩(S O P), which means "the current verb links subject `S` and object `O` to make the phrase `P`". When ⟨TransVerb⟩ calls its definition, the corresponding arguments are bound together. If `S` and `O` are inputs, ⟨TransVerb⟩ constructs `P`, which has the form `loves(S O)`. After extending the whole grammar in similar fashion (following the parser code), we get the following rules:

⟨Sentence⟩(P)	::=	⟨NounPhrase⟩(N P1 P) ⟨VerbPhrase⟩(N P1)
⟨NounPhrase⟩(N P1 P)	::=	⟨Determiner⟩(N P2 P1 P) ⟨Noun⟩(N P3) ⟨RelClause⟩(N P3 P2)
⟨NounPhrase⟩(N P1 P1)	::=	⟨Name⟩(N)
⟨VerbPhrase⟩(S P)	::=	⟨TransVerb⟩(S O P1) ⟨NounPhrase⟩(O P1 P)
⟨VerbPhrase⟩(S P)	::=	⟨IntransVerb⟩(S P)
⟨RelClause⟩(S P1 and(P1 P2))	::=	who ⟨VerbPhrase⟩(S P2)
⟨RelClause⟩(S P1 P1)	::=	ε

⟨Determiner⟩(S P1 P2 all(S imply(P1 P2))) ::= `every`
⟨Determiner⟩(S P1 P2 exists(S and(P1 P2)) ::= `a`
⟨Noun⟩(N man(N)) ::= `man`
⟨Noun⟩(N woman(N)) ::= `woman`
⟨Name⟩(john) ::= `john`
⟨Name⟩(mary) ::= `mary`
⟨TransVerb⟩(S O loves(S O)) ::= `loves`
⟨IntransVerb⟩(S lives(S)) ::= `lives`

These rules correspond exactly to the parser program we have written. You can see the advantage of using the rules: they are more concise and easier to understand than the program. They can be automatically translated into a program. This translation is so useful that the Prolog language has a built-in preprocessor to support it.

This kind of grammar is called a *definite clause grammar*, or DCG, because each rule corresponds to a kind of Horn clause called a definite clause. Each nonterminal can have arguments. When a nonterminal is matched with a rule, the corresponding arguments are unified together. DCGs are a simple example of a very general kind of grammar called *unification grammar*. Many different kinds of unification grammar are used in natural language parsing. The practical ones use constraint programming instead of relational programming.

9.5 A grammar interpreter

The previous section shows how to build simple parser and how to extend it to return a parse tree. For each new grammar we want to parse, we have to build a new parser. The parser is "hardwired": its implementation is based on the grammar it parses. Wouldn't it be nice to have a generic parser that would work for all grammars, simply by passing the grammar definition as an argument? A generic parser is easier to use and more flexible than a hardwired parser. To represent the grammar in a programming language, we encode it as a data structure. Depending on how flexible the language is, the encoding will look almost like the grammar's EBNF definition.

The generic parser is an example of an interpreter. Recall that an interpreter is a program written in language L_1 that accepts programs written in another language L_2 and executes them. For the generic parser, L_1 is the relational computation model and L_2 is a grammar definition.

The generic parser uses the same execution strategy as the hardwired parser. It keeps track of two extra arguments: the token sequence to be parsed and the rest of the sequence. It uses a choice operation to choose a rule for each nonterminal. It is executed with encapsulated search.

9.5.1 A simple grammar

To keep things simple in describing the generic parser, we use a small grammar that defines s-expressions. An *s-expression* starts with a left parenthesis, followed by a possibly empty sequence of atoms or s-expressions, and ends in a right parenthesis. Two examples are `(a b c)` and `(a (b) () (d (c)))`. S-expressions were originally used in Lisp to represent nested lists. Our grammar will parse s-expressions and build the list that they represent. Here is the grammar's definition:

⟨sexpr⟩(s(As)) ::= '(' ⟨seq⟩(As) ')'
⟨seq⟩(nil) ::= ε
⟨seq⟩(A|As) ::= ⟨atom⟩(A) ⟨seq⟩(As)
⟨seq⟩(A|As) ::= ⟨sexpr⟩(A) ⟨seq⟩(As)
⟨atom⟩(X) ::= X & *(X is an atom different from '(' and ')')*

This definition extends the EBNF notation by allowing terminals to be variables and by adding a boolean condition to check whether a rule is valid. These extensions occur in the definition of ⟨atom⟩(X). The argument X represents the actual atom that is parsed. To avoid confusion between an atom and the left or right parenthesis of an s-expression, we check that the atom is not a parenthesis.

9.5.2 Encoding the grammar

Let us encode this grammar as a data structure. We will first encode rules. A rule is a tuple with two parts, a head and a body. The body is a list of nonterminals and terminals. For example, the rule defining ⟨sexpr⟩ could be written as:

```
local As in
   rule(sexpr(s(As)) [´(´ seq(As) ´)´])
end
```

The unbound variable `As` will be bound when the rule is used. This representation is not quite right. There should be a fresh variable `As` each time the rule is used. To implement this, we encode the rule as a function:

```
fun {$} As in
   rule(sexpr(s(As)) [´(´ seq(As) ´)´])
end
```

Each time the function is called, a tuple is returned containing a fresh variable. This is still not completely right, since we cannot distinguish nonterminals without arguments from terminals. To avoid this confusion, we wrap terminals in a tuple with label `t`. (This means that we cannot have a nonterminal with label `t`.) This gives the final, correct representation:

```
fun {$} As in
   rule(sexpr(s(As)) [t(´(´) seq(As) t(´)´)])
end
```

```
r(sexpr:[fun {$} As in
            rule(sexpr(s(As)) [t(´(´) seq(As) t(´)´)])
         end]
  seq:  [fun {$}
            rule(seq(nil) nil)
         end
         fun {$} As A in
            rule(seq(A|As) [atom(A) seq(As)])
         end
         fun {$} As A in
            rule(seq(A|As) [sexpr(A) seq(As)])
         end]
  atom: [fun {$} X in
            rule(atom(X)
                 [t(X)
                  fun {$}
                     {IsAtom X} andthen X\=´(´ andthen X\=´)´
                  end])
         end])
```

Figure 9.7: Encoding of a grammar

Now that we can encode rules, let us encode the complete grammar. We represent the grammar as a record where each nonterminal has one field. This field contains a list of the nonterminal's rules. We have seen that a rule body is a list containing nonterminals and terminals. We add a third kind of entry, a boolean function that has to return **true** for the rule to be valid. This corresponds to the condition we used in the definition of ⟨atom⟩(X).

Figure 9.7 gives the complete grammar for s-expressions encoded as a data structure. Note how naturally this encoding uses higher-order programming: rules are functions that themselves may contain boolean functions.

9.5.3 Running the grammar interpreter

Let us define an ADT for the grammar interpreter. The function `NewParser` takes a grammar definition and returns a parser:

```
Parse={NewParser Rules}
```

`Rules` is a record like the grammar definition in Figure 9.7. `Parse` takes as inputs a goal to be parsed, `Goal`, and a list of tokens, `S0`. It does the parse and returns the unparsed remainder of `S0` in `S`:

```
{Parse Goal S0 S}
```

While doing the parse, it can also build a parse tree because it unifies the arguments of the nonterminal with the head of the chosen rule.

The parser is executed with encapsulated search. Here is an example:

```
{Browse {SolveOne
   fun {$} E in
      {Parse sexpr(E)
             [´(´ hello ´(´  this is an sexpr ´)´ ´)´] nil}
      E
   end}}
```

This returns a list containing the first solution:

```
[s([hello s([this is an sexpr])])]
```

9.5.4 Implementing the grammar interpreter

Figure 9.8 gives the definition of the grammar interpreter. `NewParser` creates a parser `Parse` that references the grammar definition in `Rules`. The parser is written as a **`case`** statement. It accepts four kinds of goals:

- A list of other goals. The parser is called recursively for all goals in the list.
- A procedure, which should be a zero-argument boolean function. The function is called and its result is unified with **`true`**. If the result is **`false`**, then the parser fails, which causes another alternative to be chosen.
- A terminal, represented as the tuple `t(X)`. This terminal is unified with the next element in the input list.
- A nonterminal, represented as a record. Its label is used to look up the rule definitions in `Rules`. Then a rule is chosen nondeterministically with `ChooseRule` and `Parse` is called recursively.

This structure is typical of interpreters. They examine the input and decide what to do depending on the input's syntax. They keep track of extra information (here, the arguments `S0` and `S`) to help do the work.

Dynamic choice points

The parser calls `ChooseRule` to choose a rule for a nonterminal. Using the **`choice`** statement, we could write `ChooseRule` as follows:

```
proc {ChooseRule Rs Goal Body}
   case Rs of nil then fail
   [] R|Rs2 then
      choice
         rule(Goal Body)={R}
      []
         {ChooseRule Rs2 Goal Body}
      end
   end
end
```

```
fun {NewParser Rules}
   proc {Parse Goal S0 S}
      case Goal
      of nil then S0=S
      [] G|Gs then S1 in
         {Parse G S0 S1}
         {Parse Gs S1 S}
      [] t(X) then S0=X|S
      else if {IsProcedure Goal} then
         {Goal}=true
         S0=S
      else Body Rs in /* Goal is a nonterminal */
         Rs=Rules.{Label Goal}
         {ChooseRule Rs Goal Body}
         {Parse Body S0 S}
      end end
   end
   proc {ChooseRule Rs Goal Body}
      I={Space.choose {Length Rs}}
   in
      rule(Goal Body)={{List.nth Rs I}}
   end
in
   Parse
end
```

Figure 9.8: Implementing the grammar interpreter

This definition creates a series of binary choice points. (Note that it calls the rule definition `R` to create a fresh rule instance.) There is another, more flexible and efficient way to write `ChooseRule`. Instead of using the **`choice`** statement, which implies a statically fixed number of choices, we use another operation, `Space.choose`, which works with any number of choices. `Space.choose` is part of the `Space` module, which defines operations on computation spaces. The curious reader can skip ahead to Chapter 12 to find out more about them. But it is not necessary to understand computation spaces to understand `Space.choose`.

The call `I={Space.choose N}` creates a choice point with `N` alternatives and returns `I`, the alternative that is picked by the search strategy. `I` ranges from 1 to `N`. The number of alternatives can be calculated at run-time, wherease in the **`choice`** statement it is statically known as part of the program's syntax. In fact, the **`choice`** statement is a linguistic abstraction that is implemented with `Space.choose`. The following statement:

```
choice
   ⟨stmt⟩1
[] ... []
```

$\langle \text{stmt} \rangle_n$
end

is translated as:

```
case {Space.choose N}
of 1 then ⟨stmt⟩1
...
[] N then ⟨stmt⟩n
end
```

So the `Space.choose` operation is the real basic concept and the **`choice`** statement is a derived concept.

Meta-interpreters

Our interpreter is actually a special kind of interpreter called a *meta-interpreter* since it uses the relational model's unify operation directly to implement the grammar's unify operation. In general, any interpreter of L_2 that uses operations of L_1 directly to implement the same operations in L_2 is called a meta-interpreter. Writing meta-interpreters is a standard programming technique in languages whose primitive operations are complex. It avoids having to reimplement these operations and it is more efficient. A popular case is Prolog, which has unification and search as primitives. It is easy to explore extensions to Prolog by writing meta-interpreters.

9.6 Databases

A *database* is a collection of data that has a well-defined structure. Usually, it is assumed that the data are long-lived, in some loose sense, e.g., they survive independently of whether the applications or the computer itself is running. (The latter property is often called *persistence*.)

There are many ways to organize the data in a database. One of the most popular ways is to consider the data as a set of relations, where a *relation* is a set of tuples. A database organized as a set of relations is called a *relational database*. For example, a graph can be defined by one relation, which is a set of tuples where each tuple represents one edge (see Figure 9.9):

```
edge(1 2)  edge(2 1)  edge(2 3)  edge(3 4)
edge(2 5)  edge(5 6)  edge(4 6)  edge(6 7)
edge(6 8)  edge(1 5)  edge(5 1)
```

A relational database explicitly stores these tuples so that we can calculate with them. We can use the relational computation model of this chapter to do these calculations. Typical operations on a relational database are query (reading the data) and update (modifying the data):

- A *query* is more than just a simple read, but is a logical formula whose basic elements are the relations in the database. It is the role of the DBMS (database management system) to find all tuples that satisfy the formula.

- An *update* means to add information to the database. This information must be of the right kind and not disturb the organization of the database. The update is usually implemented as a transaction (see Section 8.5).

This section touches on just a few parts of the area of databases. For more information, we refer the reader to the comprehensive introduction by Date [42].

Relational programming is well-suited for exploring the concepts of relational databases. There are several reasons for this:

- It places no restrictions on the logical form of the query. Even if the query is highly disjunctive (it has many choices), it will be treated correctly (albeit slowly).

- It allows to experiment with deductive databases. A *deductive database* is a database whose implementation can deduce additional tuples that are not explicitly stored. Typically, the deductive database allows defining new relations in terms of existing relations. No tuples are stored for these new relations, but they can be used just like any other relation.

The deep reason for these properties is that the relational computation model is a form of logic programming.

9.6.1 Defining a relation

Let us first define an abstraction to calculate with relations. For conciseness, we use object-oriented programming to define the abstraction as a class, `RelationClass`.

- A new relation is an instance of `RelationClass`, e.g., `Rel={New RelationClass init}` creates the initially empty relation `Rel`.

- The following operations are possible:

  - `{Rel assert(T)}` adds the tuple `T` to `Rel`. Assert can only be done *outside* a relational program.
  - `{Rel assertall(Ts)}` adds the list of tuples `Ts` to `Rel`. Assertall can only be done *outside* a relational program.
  - `{Rel query(X)}` binds `X` to one of the tuples in `Rel`. `X` can be any partial value. If more than one tuple is compatible with `X`, then search can enumerate all of them. Query can only be done *inside* a relational program.

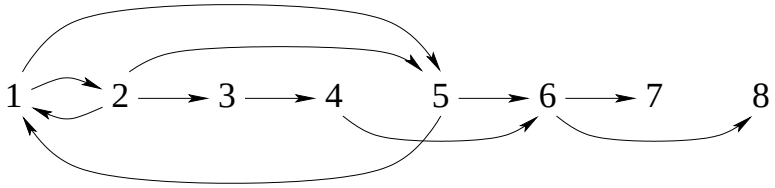

Figure 9.9: A simple graph

These operations are similar to what a Prolog system provides. For example, `assert` is a limited version of Prolog's `assert/1` that can assert facts (i.e., tuples), not complete clauses. For the examples that follow, we assume that `Rel` has efficiency similar to a good Prolog implementation [29]. That is, the set of tuples is stored in a dictionary that indexes them on their first argument. This makes it possible to write efficient programs. Without this indexing, even simple lookups would need to do linear search. More sophisticated indexing is possible, but in practice first-argument indexing is often sufficient. Section 9.6.3 gives an implementation of `RelationClass` that does first-argument indexing.

9.6.2 Calculating with relations

An example relation

Let us show an example of this abstraction for doing calculations on graphs. We use the example graph of Figure 9.9. We define this graph as two relations: a set of nodes and a set of edges. Here is the set of nodes:

```
NodeRel={New RelationClass init}
{NodeRel
 assertall([node(1) node(2) node(3) node(4)
            node(5) node(6) node(7) node(8)])}
```

The tuple `node(1)` represents the node 1. Here is the set of edges:

```
EdgeRel={New RelationClass init}
{EdgeRel
 assertall([edge(1 2) edge(2 1) edge(2 3) edge(3 4)
            edge(2 5) edge(5 6) edge(4 6) edge(6 7)
            edge(6 8) edge(1 5) edge(5 1)])}
```

The tuple `edge(1 2)` represents an edge from node 1 to node 2. We can query `NodeRel` or `EdgeRel` with the message `query`. Let us define the procedures `NodeP` and `EdgeP` to make this more concise:

```
proc {NodeP A} {NodeRel query(node(A))} end
proc {EdgeP A B} {EdgeRel query(edge(A B))} end
```

With these definitions of `NodeP` and `EdgeP` we can write relational programs.

Some queries

Let us start with a very simple query: what edges are connected to node 1? We define the query as a one-argument procedure:

```
proc {Q ?X} {EdgeP 1 X} end
```

This calls `EdgeP` with first argument 1. We calculate the results by using `Q` as argument to a search operation:

```
{Browse {SolveAll Q}}
```

This displays:

```
[2 5]
```

Here is another query, which defines paths of length three whose nodes are in increasing order:

```
proc {Q2 ?X} A B C D in
   {EdgeP A B} A<B=true
   {EdgeP B C} B<C=true
   {EdgeP C D} C<D=true
   X=path(A B C D)
end
```

We list all paths that satisfy the query:

```
{Browse {SolveAll Q2}}
```

This displays:

```
[path(3 4 6 7) path(3 4 6 8) path(2 3 4 6)
 path(2 5 6 7) path(2 5 6 8) path(1 2 3 4)
 path(1 2 5 6) path(1 5 6 7) path(1 5 6 8)]
```

The query `Q2` has two kinds of calls, generators (the calls to `EdgeP`) and testers (the conditions). Generators can return several results. Testers can only fail. For efficiency, it is a good idea to call the testers as early as possible, i.e., as soon as all their arguments are bound. In `Q2`, we put each tester immediately after the generator that binds its arguments.

Paths in a graph

Let us do a more realistic calculation. We will calculate the paths in our example graph. This is an example of a *deductive database* calculation, i.e., we will perform logical inferences on the database. We define a *path* as a sequence of nodes such that there is an edge between each node and its successor and no node occurs more than once. For example, `[1 2 5 6]` is a path in the graph defined by `EdgeP`. We can define path as a derived relation `PathP`, i.e., a new relation defined in terms of `EdgeP`. Figure 9.10 shows the definition.

The relation `{PathP A B Path}` is true if `Path` is a path from `A` to `B`. `PathP` uses an auxiliary definition, `Path2P`, which has the extra argument `Trace`, the list of already-encountered nodes. `Trace` is used to avoid using the same node

```
proc {PathP ?A ?B ?Path}
   {NodeP A}
   {Path2P A B [A] Path}
end

proc {Path2P ?A ?B Trace ?Path}
   choice
      A=B
      Path={Reverse Trace}
   [] C in
      {EdgeP A C}
      {Member C Trace}=false
      {Path2P C B C|Trace Path}
   end
end
```

Figure 9.10: Paths in a graph

twice and also to accumulate the path. Let us look more closely at `Path2P`. It has two choices, each of which has a logical reading:

- In the first choice, `A=B`, which means the path is complete. In that case, the path is simply the reverse of `Trace`.
- In the second choice, we extend the path. We add an edge from `A` to another node `C`. The path from `A` to `B` consists of an edge from `A` to `C` and a path from `C` to `B`. We verify that the edge `C` is not in `Trace`.

The definition of `Path2P` is an example of logic programming: the logical definition of `Path2P` is used to perform an algorithmic calculation. Note that the definition of `Path2P` is written completely in the relational computation model. It is an interesting combination of deterministic and nondeterministic calculation: `EdgeP` and `Path2P` are both nondeterministic and the list operations `Reverse` and `Member` are both deterministic.

9.6.3 Implementing relations

Figure 9.11 shows the implementation of `RelationClass`. It is quite simple: it uses a dictionary to store the tuples and the **`choice`** statement to enumerate query results. The choice is done in the procedure `Choose`, which successively chooses all elements of a list. First-argument indexing is a performance optimization. It is implemented by using a new operation, `IsDet`, to check whether the argument is bound or unbound. If the first argument is unbound, then all tuples are possible results. If the first argument is bound, then we can use it as an index into a much smaller set of possible tuples.

```
proc {Choose ?X Ys}
   choice   Ys=X|_
   [] Yr in Ys=_|Yr {Choose X Yr} end
end

class RelationClass
   attr d
   meth init
      d:={NewDictionary}
   end
   meth assertall(Is)
      for I in Is do {self assert(I)} end
   end
   meth assert(I)
      if {IsDet I.1} then
         Is={Dictionary.condGet @d I.1 nil} in
         {Dictionary.put @d I.1 {Append Is [I]}}
      else
         raise databaseError(nonground(I)) end
      end
   end
   meth query(I)
      if {IsDet I} andthen {IsDet I.1} then
         {Choose I {Dictionary.condGet @d I.1 nil}}
      else
         {Choose I {Flatten {Dictionary.items @d}}}
      end
   end
end
```

Figure 9.11: Implementing relations (with first-argument indexing)

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Procedure application
choice ⟨s⟩$_1$ [] ... [] ⟨s⟩$_n$ **end**	Choice
fail	Failure
{IsDet ⟨x⟩ ⟨y⟩}	Boundness test
{NewCell ⟨x⟩ ⟨y⟩}	Cell creation
{Exchange ⟨x⟩ ⟨y⟩ ⟨z⟩}	Cell exchange

Table 9.3: The extended relational kernel language

Extended relational computation model

The implementation in Figure 9.11 extends the relational computation model in two ways: it uses stateful dictionaries (i.e., explicit state) and the operation `IsDet`.[2] This is a general observation: to implement useful relational abstractions, we need state (for modularity) and the ability to detect whether a variable is still unbound or not (for performance optimization). Table 9.3 shows the kernel language of this extended computation model. Because of encapsulated search, a running relational program can only read state, not modify it. The boolean function `{IsDet X}` returns true or false depending on whether `X` is not an unbound variable or is an unbound variable. A variable that is not unbound is called *determined*. `IsDet` corresponds exactly to the Prolog operation `nonvar(X)`.

9.7 The Prolog language

Despite many extensions and new ideas, Prolog is still the most popular language for practical logic programming [182]. This is partly because Prolog has a quite simple operational model that easily accommodates many extensions and partly because no consensus has yet been reached on a successor. The computation model of the "pure" subset of Prolog, i.e., Prolog minus its extralogical features, is exactly the relational computation model.

Modern implementations of Prolog are efficient and provide rich functionality for application development (e.g., [29]). It is possible to compile Prolog with similar execution efficiency as C; the Aquarius and Parma systems are construc-

[2] Leaving aside exceptions, since they are only used for detecting erroneous programs.

tive proof of this [194, 188]. The successful series of conferences on Practical Applications of Prolog is witness to the usefulness of Prolog in industry.

Prolog is generally used in application areas in which complex symbolic manipulations are needed, such as expert systems, specialized language translators, program generation, data transformation, knowledge processing, deductive databases, and theorem proving. There are two application areas in which Prolog is still predominant over other languages: natural language processing and constraint programming. The latter in particular has matured from being a subfield of logic programming into being a field in its own right, with conferences, practical systems, and industrial applications.

Prolog has many advantages for such applications. The bulk of programming can be done cleanly in its pure declarative subset. Programs are concise due to the expressiveness of unification and the term notation. Memory management is dynamic and implicit. Powerful primitives exist for useful non-declarative operations. The `call/1` provides a form of higher-orderness (first-class procedures, but without lexical scoping). The `setof/3` provides a form of encapsulated search that can be used as a database query language.

The two programming styles

Logic programming languages have traditionally been used in two very different ways:

- For *algorithmic problems*, i.e., for which efficient algorithms are known. Most applications written in Prolog, including expert systems, are of this kind.

- For *search problems*, i.e., for which efficient algorithms are not known, but that can be solved with search. For example, combinatoric optimization or theorem proving. Most applications in constraint programming are of this kind.

Prolog was originally designed as a compromise between these two styles. It provides backtracking execution, which is just built-in depth-first search. This compromise is not ideal. For algorithmic problems the search is not necessary. For search problems the search is not good enough. This problem has been recognized to some extent since the original conception of the language in 1972. The first satisfactory solution, encapsulating search with computation spaces, was given by the AKL language in 1990 [70, 92]. The unified model of this book simplifies and generalizes the AKL solution (see Chapter 12).

9.7.1 Computation model

The Prolog computation model has a layered structure with three levels:

- The core consists of a simple theorem prover that uses Horn clauses and that executes with SLDNF resolution [114]. The acronym SLDNF has a long history; it means approximately "Selection in Linear resolution for Definite clauses, augmented by Negation as Failure". It defines a theorem prover that executes like the relational computation model. *Negation as failure* is a practical technique to deduce some negative information: if trying to prove the atom $\langle \texttt{a} \rangle$ fails finitely, then deduce $\neg\langle \texttt{a} \rangle$. *Finite failure* means that the search tree (defined in Section 9.1.2) has only a finite number of leaves (no infinite loops) and all are failed. This can easily be detected in the relational model: it means simply that `Solve` finds no solutions and does not loop. Negation as failure is incomplete: if the theorem prover loops indefinitely or blocks trying to prove $\langle \texttt{a} \rangle$ then we cannot deduce anything.
- The second level consists of a series of extralogical features that are used to modify and extend the resolution-based theorem prover. These features consist of the `freeze/2` operation (giving data-driven execution, implemented with coroutining), the `bagof/3` and `setof/3` operations (giving aggregate operations similar to database querying), the `call/1` operation (giving a limited form of higher-order programming), the cut operation "`!`" (used to prune search), and the `var/1` and `nonvar/1` operations (also used to prune search).
- The third level consists of the `assert/1` and `retract/1` operations, which provide explicit state. This is important for program design and modularity.

The Prolog computation model is the heart of a whole family of extensions. One of the most important extensions is constraint logic programming. It retains the sequential search strategy of Prolog, but extends it with new data structures and constraint solving algorithms. See Chapter 12 for more information.

There is a second family of computation models for logic programming, called concurrent logic programming. These are centered around the nondeterministic concurrent model, which is explained in Section 5.7.1. This model allows to write logic programs for long-lived concurrent calculations that interact amongst each other and with the real world. This makes it possible to write operating systems.

In the late 1980's, the first deep synthesis of these two families was done by Maher and Saraswat, resulting in concurrent constraint programming [117, 163]. This model was first realized practically by Haridi and Janson [70, 92]. The general computation model of this book is a concurrent constraint model. For more information about the history of these ideas, we recommend [196].

Future developments

There are three major directions in the evolution of logic programming languages:

- **Mercury**. An evolution of Prolog that is completely declarative, statically typed and moded, and higher-order. It focuses on the algorithmic programming style. It keeps the Horn clause syntax and global backtracking.

- **Oz**. An evolution of Prolog and concurrent logic programming that cleanly separates the algorithmic and search programming styles. It also cleanly integrates logic programming with other computation models. It replaces the Horn clause syntax with a syntax that is closer to functional languages.

- **Constraint programming**. An evolution of Prolog that consists of a set of constraint algorithms and ways to combine them to solve complex optimization problems. This focuses on the search programming style. Constraint techniques can be presented as libraries (e.g., ILOG Solver is a C++ library) or language extensions (e.g., SICStus Prolog and Oz).

9.7.2 Introduction to Prolog programming

Let us give a brief introduction to programming in Prolog. We start with a simple program and continue with a more typical program. We briefly explain how to write good programs, which both have a logical reading and execute efficiently. We conclude with a bigger program that shows Prolog at its best: constructing a KWIC index. For more information on Prolog programming, we recommend one of many good textbooks, such as [182, 21].

A simple predicate

Let us once again define the factorial function, this time as a Prolog predicate.

```
factorial(0, 1).
factorial(N, F) :- N>0,
   N1 is N-1, factorial(N1, F1), F is N*F1.
```

A Prolog program consists of a set of predicates, where each predicate consists of a set of clauses. A predicate corresponds roughly to a function or procedure in other languages. Each clause, when considered by itself, should express a property of the predicate. This allows us to do purely logical reasoning about the program. The two clauses of `factorial/2` satisfy this requirement. Note that we identify the predicate by its name and its number of arguments.

A particularity about Prolog is that all arguments are *terms*, i.e., tuples in our terminology. This shows up clearly in its treatment of arithmetic. The syntax `N-1` denotes a term with label `'-'` and two arguments `N` and 1. To consider the term as a subtraction, we pass it to the predicate `is/2`, which interprets it and does the subtraction.[3] This is why we have to use the extra variables `N1` and `F1`. Let us call the predicate with `N` bound and `F` unbound:

```
| ?- factorial(10, F).
```

[3]Most Prolog compilers examine the term at compile time and generate a sequence of instructions that does the arithmetic without constructing the term.

(The notation `| ?-` is part of the Prolog system; it means that we are performing an interactive query.) This returns with `F` bound to 3628800. How is this answer obtained? The Prolog system considers the clauses as precise operational instructions on how to execute. When calling factorial, the system tries each clause in turn. If the clause head unifies with the caller, then the system executes the calls in the clause body from left to right. If the clause head does not unify or a body call fails, then the system backtracks (i.e., undoes all bindings done in the clause) and tries the next clause. If the last clause has been tried, then the whole predicate fails.

Calculating logically with lists

Factorial is a rather atypical Prolog predicate, since it does not use the power of unification or search. Let us define another predicate that is more in the spirit of the language, namely `sublist(L1, L2)`, which is true for lists `L1` and `L2` whenever `L1` occurs as a contiguous part of `L2`:

```
sublist(L1, L2) :- append(V, T, L2), append(H, L1, V).
```

Logically, this says "`L1` is a sublist of `L2` if there exist lists `H` and `T` such that appending together `H`, `L1`, and `T` gives `L2`". These variables do not need an explicit declaration; they are declared implicitly with a scope that covers the whole clause. The order of the `append/3` calls might seem strange, but it does not change the logical meaning. We will see later why this order is important. We define `append/3` as follows:

```
append([], L2, L2).
append([X|M1], L2, [X|M3]) :- append(M1, L2, M3).
```

In Prolog syntax, `[]` denotes the empty list `nil` and `[X|M1]` denotes the list pair `X|M1`. In the relational model of Chapter 9, this program can be written as follows:

```
proc {Sublist L1 L2} H V T in
   {Append V T L2} {Append H L1 V}
end

proc {Append L1 L2 L3}
   choice
      L1=nil  L3=L2
   [] X M1 M3 in
      L1=X|M1 L3=X|M3 {Append M1 L2 M3}
   end
end
```

Each clause is an alternative in a **`choice`** statement. All the local variables in the clause bodies are declared explicitly.