CS442
Module 6: Logic Programming

University of Waterloo

Winter 2023

"In Prolog programming (in contrast, perhaps, to life in general) our goal is to fail as quickly as possible."

— Leon Sterling and Ehud Shapiro, *The Art of Prolog*

Programming languages are often classified as either *declarative* or *imperative*. These two rather broad terms refer to the degree to which the task of programming is abstracted from the details of the underlying machine, and are often associated with the terms *high-level* and *low-level*, respectively. A programming language is declarative if programs in the language describe the tasks to be performed without outlining the specific steps to be taken. Conversely, imperative languages prescribe more of the details of how computations are to be carried out.

Aside: Of course, technically, a declarative language is simply a language in which *declarations* are primary, and an imperative language is a language in which *imperatives* (commands) are primary, but this distinction in level of abstraction is a natural consequence of this structural difference.

In reality, the terms *declarative* and *imperative* are most appropriately treated as relative terms: one language is "more declarative" or "more imperative" than another. In this way, the notions of "declarative" and "imperative" suggest the possibility of constructing a "spectrum" of programming languages arranged according to the level of abstraction at which they operate. While an actual construction of such a spectrum is not really a well-defined task, we may certainly ask ourselves what languages would lie at its boundaries—what are the lowest- and highest-level programming languages?

At the lowest level, we can program a computer by specifying the sequence of electrical impulses that passes through the processor. Programming of this nature amounts essentially to specifying the sequence of 1's and 0's that make up an executable program.

At the highest level, we might imagine a language that allows us to simply describe the characteristics of the program we seek, and then allow the computer to perform the task of meeting the criteria we laid out. The language we use to describe our programs should ideally be completely devoid of any references to computers or implementation details; it should simply outline the constraints to be satisfied in order to solve the problem at hand.

An ideal candidate for such a language is predicate logic. By defining an appropriate problem domain and set of predicates, it is possible to use predicate logic to describe any computable problem. Moreover, predicate logic is a language of mathematics, and its existence predates computers. Indeed, a system in which we may program purely in predicate logic is an ideal to which many in the declarative programming community aspire. In this chapter, we discuss the logic programming paradigm, and its best-known representative language, Prolog.

Prolog is our exemplar, but it's actually more than that for logic programming; virtually all languages in the logic programming paradigm are based on Prolog. In a very real sense, logic programming *is* Prolog.

Logic programming forms one branch of a family of *query languages*, and we will explore query programming more broadly at the end of this module.

1 Programmming in Logic

We begin with a summary of the language of predicate logic. Strings in this language (called formulas) are composed of the following elements:

- *constants*, denoted by $a, b, c, \cdots$, possibly subscripted;
- *function letters*, denoted by $f, f_1, f_2, \cdots, g, g_1, g_2, \cdots$, etc.;
- *predicate symbols*, denoted by (possibly subscripted) capital letters;
- *variables*, denoted by $x, x_1, x_2, \cdots, y, y_1, y_2, \cdots$, etc.;
- *grouping and delimiting symbols*: (,) and ,;
- *logical connectives*: $\neg$, $\rightarrow$, $\vee$, $\wedge$ (negation, implication, or, and and, respectively);
- *quantifiers*: $\forall$ and $\exists$.

With each function symbol and each predicate symbol, we associate a number $n \geq 0$, known as its *arity*. The arity of a function or predicate denotes the number of arguments it will accept. The particular language under consideration depends on our choice of constants, function letters, and predicate symbols. For example, suppose we introduce the unary function symbol s and the binary predicate g. Then the following is a formula in our predicate language: $\forall x.g(s(x), x)$. We might choose to interpret this formula as follows: we take as our domain of consideration the set of natural numbers, so that the variable x may range over the values $1, 2, \cdots$. We then associate the function s with the successor function, $\lambda x.\, x + 1$, and the predicate g with the binary predicate $\lambda(x, y).\, x > y$. Then the formula reads: "For every natural number x, the successor of x is greater than x," which, in this domain of interpretation, happens to be true. We assume at this point that you have sufficient background in logic to distinguish between well-formed and mal-formed formulas in predicate languages.

The logical connectives are those used in logic in general:

- $\neg X$ is true if X is false (logical negation).
- $X \rightarrow Y$ means "X implies Y", and so is true if, in all cases that X is true, Y is also true. If X is always false, then the implication is said to be "vacuously true".
- $X \vee Y$ is true if either X or Y is true (logical or).
- $X \wedge Y$ is true if both X and Y are true.

A sequence of clauses joined by $\vee$ is called a *disjunction*, while a sequence of clauses joined by $\wedge$ is called a *conjunction*.

> **Aside:** The only way this author can ever remember the difference between $\vee$ and $\wedge$ is to observe that $\wedge$ looks a bit like an 'A', as in "AND". Perhaps the same mnemonic will help you.

The logic programming paradigm proceeds as follows: we first assert the truth of a set of formulas in our predicate language. Our set of assertions forms a *database*. We then either query the database about the truth of a particular formula, or we present a formula containing variables and ask the system to determine which instantiations of the variables satisfy the formula, i.e., make it evaluate to true.

This task, as stated above, is quite difficult, as predicate languages admit a wide variety of qualitatively different formulas. To make the task more feasible, we will show how formulas in predicate languages may be placed in a more uniform format known as *clausal form*. The programming language we will eventually get to, Prolog, demands that its predicates be in a clausal form. The exact constraints of clausal form are best described by describing the process of transforming any formula *into* clausal form. Note that the mechanical details of converting a predicate into clausal form are not part of Prolog, they're just what you have to do to express any predicate *in* Prolog, so the main purpose of this section is simply to demonstrate that restricting ourselves to clausal form doesn't actually reject any predicates, as they can all be converted.

We first note that we can get rid of the logical implication symbol, →, by replacing all occurrences of $A \to B$ by $\neg A \lor B$. For example $\forall x.(A(x) \to B(x, f(y)))$ would become $\forall x.(\neg A(x) \lor B(x, f(y)))$.

Next, we can move all of quantifiers to the left. Doing so will produce a formula of the form $\langle quantifiers \rangle\ \langle body \rangle$. To do this we observe the following two rules:

$$\neg\forall x.A \text{ becomes } \exists x.\neg A$$
$$\neg\exists x.A \text{ becomes } \forall x.\neg A$$

Quantifiers may be pulled outside of conjunctions and disjunctions unchanged. Note, however, that if moving a quantifier to the left introduces a variable capture, then we must perform an α-conversion and rename the quantified variable to a fresh one. That is, to simplify an expression like this one:

$$(\forall x.A(x)) \land (\forall x.B(x))$$

we must first rename one or both of the x's, like so:

$$(\forall x.A(x)) \land (\forall x'.B(x'))$$

Then we may move the quantifiers out like so:

$$\forall x.\forall x'.A(x) \land B(x')$$

Note also that we may not reorder quantifiers with respect to each other. That is, when moving quantifiers to the left, we must respect the left-to-right order in which they occurred in the original formula. For example, the formula $\neg(\exists x.A(x) \land \forall x.B(x))$ becomes $\forall x.\neg(A(x) \land \forall x.B(x))$, and then $\forall x.\exists y.\neg(A(x) \land B(y))$.

Next, we can eliminate existentially quantified variables via a process known as *Skolemization*: essentially, we replace them with freshly-invented constants, known as *Skolem constants*. For example, $\exists x.\exists y.(A(x, y) \lor B(x))$ becomes $A(s_1, s_2) \lor B(s_1)$, where s_1 and s_2 are Skolem constants.

Aside: Actually, we've been Skolemizing throughout the course! Every time we've had a metavariable in the Post system, that's actually a Skolemized existential quantifier.

Skolemization becomes more difficult in the presence of universal quantifiers. Consider the expression $\exists y.\forall x.P(x, y)$. This expression asserts that there is a value of y such that $P(x, y)$ holds of every x. Hence we may replace y with a Skolem constant s_1 that represents the particular value of y for which the assertion holds. However, consider the expression $\forall x.\exists y.P(x, y)$. In this case, the assertion does not guarantee that there is a single choice of y for which $P(x, y)$ will hold for all x; rather it asserts that a suitable value of y can be found for each value of x. Thus, the value of y is dependent on the value of x, and therefore it would not be appropriate to replace y with a Skolem constant. Rather, we should replace y with a newly-invented *Skolem function* parameterized by x. Thus, we Skolemize $\forall x.\exists y.P(x, y)$ as $\forall x.P(x, s_1(x))$, where s_1 is a Skolem function.

In general, an existentially quantified variable x is replaced with a Skolem function parameterized by all of the universally quantified variables whose quantifiers occur to the left of x's quantifier. If there are no universal quantifiers to the left of x's quantifier, then x may be replaced with a Skolem constant.

In this way, we may convert any formula in our predicate language to one without implications, in which the only quantifiers that appear are universal quantifiers, and in which all of the quantifiers occur leftmost in the formula.

Free variables in our predicate language are variables not associated with a quantifier; hence their meaning is assumed to be externally provided. Since we seek self-contained formulas defining the characteristics of the problems we wish to solve, we shall assume that there are no free variables. Hence, after the transformations we have performed so far, all of the variables in the formula are universally quantified. Thus, the presence of the quantifiers provides no additional information and for convenience we shall omit them. For example, we shall write the Skolemized formula $\forall x.P(x, s_1(x))$ as $P(x, s_1(x))$, and understand that the former is meant when we use the latter.

What remains is a quantifier-free expression involving primitive predicates and the logical connectives $\neg$, $\vee$ and $\wedge$. Next, we will push the logical negations inward, so that negation may only be applied to lone predicates. To do this, we use the following transformation rules:

$$\begin{aligned}\neg(A \wedge B) &= \neg A \vee \neg B \\ \neg(A \vee B) &= \neg A \wedge \neg B \\ \neg\neg A &= A\end{aligned}$$

Since we have pushed the negations all the way down to individual predicates, we may consider them to be part of the predicates and ignore them for the moment. We shall call an occurrence of a predicate negated if it has a logical negation operator applied to it.

We now have a formula consisting of (possibly negated) predicates that have been combined using $\wedge$ and $\vee$. We may distribute $\vee$ over $\wedge$ using the following rule:

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

That rule is commutative, since $\vee$ is commutative. Putting this all together, we can convert a formula to what's called *conjunctive normal form* (CNF), in which the formula is written as a set of *clauses*, connected by $\wedge$, and each clause is a set of (possibly negated) predicates, connected by $\vee$. For example, the formula

$$(P(x) \vee Q(x) \vee R(y, z)) \wedge P(y) \wedge (S(z) \vee Q(z))$$

is in CNF.

Since conjunction is commutative and associative, we may dispense with the conjunctions and simply view a formula as a set of clauses; for example, we may view the example from the previous paragraph as the set

$$\{P(x) \vee Q(x) \vee R(y, z), P(y), S(z) \vee Q(z)\}$$

Each clause is a disjunction of (possibly negated) predicates. Since disjunction is commutative and associative, we may reorder the predicates so that the unnegated predicates occur first and the negated predicates occur last. Thus, a clause has the form

$$P_1 \vee \cdots \vee P_j \vee \neg P_{j+1} \vee \cdots \vee \neg P_n$$

After applying deMorgan's law[1], we can collect the negations and convert the clause to the form

$$P_1 \vee \cdots \vee P_j \vee \neg(P_{j+1} \wedge \cdots \wedge P_n)$$

We can also reintroduce implication and convert the clause to the form

$$(P_{j+1} \wedge \cdots \wedge P_n) \rightarrow (P_1 \vee \cdots \vee P_j)$$

Finally, we reverse the formula and introduce the symbol `:-` to replace (and reverse) →, obtaining

$$(P_1 \vee \cdots \vee P_j)\texttt{:-}(P_{j+1} \wedge \cdots \wedge P_n)$$

Thus, we can express any formula in our predicate language as a set of cluases, where each clause is a single implication relating a conjunction to a disjunction. We call this clausal form. The disjunction $P_1 \vee \cdots \vee P_j$ is known as the *head* of the clause, and the conjunction $P_{j+1} \wedge \cdots \wedge P_n$ is called the *body* of the clause. The meaning of the clause is that if the body of the clause is true, then the head of the clause is true.

[1] $\neg(A \wedge B) = \neg A \vee \neg B$

The major advantage of clausal form is that it admits a particularly simple method of inference known as *resolution*, which we may express by the following Post rule[2]:

$$\frac{(P_1 \vee \cdots \vee P_i \vee R)\texttt{:-}(P_{i+1} \wedge \cdots \wedge P_m) \qquad (Q_1 \vee \cdots \vee Q_j)\texttt{:-}(R \wedge Q_{j+1} \wedge \cdots \wedge Q_n)}{(P_1 \vee \cdots \vee P_i \vee Q_1 \vee \cdots \vee Q_j)\texttt{:-}(P_{i+1} \wedge \cdots \wedge P_m \wedge Q_{j+1} \wedge \cdots \wedge Q_n)}$$

That is, you can split an arbitrarily long clause in two around a new predicate symbol R: if the first part of the body is true, then either the first part of the head is true, or *something* from the rest of the head is true, represented by R. If the second part of the body is true or, by way of R, the first part of the body was true but the first part of the head was not, then something from the second part of the head is true. We can repeat this process as often as we like until we're left with single queries.

Resolution has the important property that if a set of clauses is logically inconsistent (i.e. cannot be satisfied), then resolution will be able to derive a contradiction, denoted by the empty clause, `:-`. Conversely, if resolution is able to derive `:-`, then the set of clauses is inconsistent. Thus, to determine whether the set of clauses $\{C_1, \cdots, C_n\}$ implies clause D, we need only show, by resolution, that the set of clauses $\{C_1, \cdots, C_n, \neg D\}$ is inconsistent (i.e. that it can derive `:-`).

We shall concern ourselves primarily with a particular kind of clause, known as a *Horn clause*, defined as follows:

Definition 1. A *Horn clause* is a clause whose head contains at most one predicate. A *headed* Horn clause is a Horn clause whose head contains exactly one predicate. A *headless* Horn clause is a Horn clause whose head contains no predicates.

Since the head is a disjunction of results of an implication, and the body is a conjunction of premises of an implication, a headed horn clause means "if all of these facts are true, then this one fact is true". A headless horn clause implies nothing, so it's simply a query: it either is true or it is not. Horn clauses may also have no body, in which case they are simply a declaration of a fact. A body-less horn clause is written without `:-`.

As it happens, Horn clauses are sufficient to express any computable function. Further, we may express any computable function as a set of Horn clauses with exactly one headless Horn clause. The headless Horn clause represents the "query", and has the form `:-` $P_1 \wedge \cdots \wedge P_n$. By using resolution, coupled with unification for matching, we may decide whether the query represented by the headless Horn clause may be satisfied, and if it can, which substitutions must be applied to the variables in the query in order to satisfy the clauses. This is the essence of logic programming.

2 Introduction to Prolog

Prolog is by far the most famous logic programming language. It was invented around 1970 at the University of Marseilles. The name 'Prolog' comes from *Programmation en Logique.*

Prolog programs have three major components: facts, rules, and queries. Facts and rules are entered into a database (in the form of a Prolog file) before computation. Queries are entered interactively, and are the means by which Prolog programs are "run".

Example 1. The following is an example of a fact in Prolog:

```
cogito(descartes).
```

This fact establishes a predicate "cogito" of arity 1 (that is, the predicate "cogito" takes a single argument), usually denoted `cogito/1`. The token `cogito` is known as the *functor*. Notice the facts in Prolog end with a period (`.`). The token `descartes` represents a symbolic constant that satisfies the predicate `cogito/1`. That is, when the predicate `cogito/1` is supplied with the argument `descartes`, the result is true.

[2] The Post rule for resolution as stated requires that a predicate R be present on the left-hand side of one clause and on the right-hand side of the other. In reality, these occurrences need not be identical; they only need to "match" in the sense of being unifiable according to Robinson's unification algorithm. The MGU of the two occurrences is then applied throughout the clause in the conclusion of the rule. As we will cover unification in detail in later sections, we will omit discussion of it here in the name of brevity.

Interpretation of the meaning of a predicate and its arguments is left to the programmer. In the example above, a reasonable interpretation might be "Descartes thinks" (Descartes cogitat).

We may use queries to retrieve from the database the information stored in facts. Consider the following queries:

```
?- cogito(descartes).
Yes

?- cogito(aristotle).
No
```

In the first query, we ask the database whether the assertion `cogito(descartes)` can be verified. Prolog answers with Yes, since the fact `cogito(descartes)` occurs in the database. When we query `cogito(aristotle)` (presumably asking whether Aristotle thinks), Prolog responds with No, as this assertion cannot be deduced from the database, which currently contains only the fact `cogito(descartes)`.

A database consisting only of facts is generally of little interest, as the amount of information it can provide is limited by the number of facts present. A Prolog database will usually also contain a number of rules, by which new facts may be deduced from existing facts.

Example 2. The following is an example of a rule in Prolog:

```
sum(X) :- cogito(X).
```

Reversing this to more conventional predicate logic, it can be read as

$$\forall X.(cogito(X) \rightarrow sum(X))$$

This rule establishes a predicate `sum/1`. The token `X` is a *variable*, and is universally quantified. Variables in Prolog must begin with a capital letter. All other identifiers must begin with a lowercase letter. A rule is essentially a Horn clause: the rule `sum(X) :- cogito(X)` states that for all `X`, `sum(X)` is satisfied whenever `cogito(X)` is satisfied. That is, cogito, ergo sum: I think, therefore I am.

> **Aside:** Technically this should be "cogitat, ergo sit", they think therefore they are, but let's not get distracted by Latin inflection, or by the fact that putting this philosophical observation in the third person nullifies it.

We may now issue queries containing our new predicate, `sum/1`, to the database:

```
?- sum(descartes).
Yes

?- sum(aristotle).
No
```

As expected, the query `sum(descartes)` succeeds and the query `sum(aristotle)` fails. The former can be deduced from the fact `cogito(descartes)`; the latter cannot be deduced.

Queries may also contain variables. If the query succeeds, then Prolog reports a satisfying assignment for each variable in the query. For example, we may query the predicate `sum/1` as follows:

```
?- sum(X).
X = descartes
```

When a query contains variables, it is possible for more than one instantiation of the variables to satisfy the query. For example, if we add the fact `cogito(aristotle)` to the end of our database, then the query `?- sum(X).` will have two solutions; namely, `X = descartes` and `X = aristotle`. Prolog returns the first solution it finds in the database (`X = descartes`) and then waits for the user to press a key. If we press `;`, then Prolog will return the next solution it finds in the database, or `No`, if none remain. If we press Enter, Prolog abandons the computation and returns to the interactive prompt. Thus, querying a Prolog database is—or at least can be—an interactive process.

A Prolog query is a headless Horn clause. Thus, just as a Horn clause may have a conjunction of several predicates in its body, so may a Prolog query. For example, suppose we add the following fact to the end of our database:

```
philosopher(descartes).
```

Suppose now that we issue the following query:

```
?- cogito(X), philosopher(X).
```

This query is satisfied by only those values of X satisfying both cogito(X) and philosopher(X). Thus, in this case, the only answer returned by Prolog is

```
X = descartes
```

Of course, everyone who thinks is a philosopher in their own way, so we may add the following clause to our database:

```
philosopher(X) :- cogito(X).
```

And if we query philosopher(X) now, we get that both descartes and aristotle are philosophers. Interestingly, the fact philosopher(descartes) is established in two ways: he is a philosopher because he is a thinker, but we have also stated as a plain fact that he is a philosopher. These two ways of stating clauses do not conflict with each other, even if they imply the same result.

3 Prolog Data Structures

The Prolog programming language shares a feature common in untyped functional languages, in particular Lisp and Scheme, known as *uniformity*: Prolog code and Prolog data are identical in appearance. Data structures in Prolog are known simply as structures, and consist of a functor, followed by zero or more components. The following is an example of a Prolog structure:

```
book(greatExpectations, dickens)
```

The functor is book, and the components are greatExpectations and dickens, presumably representing the title and author. Just like predicates, interpretation of the meaning of structures is entirely up to the programmer; you must be consistent, and document your structures well, for book(greatExpectations, dickens) to have any meaning. If we place the functor as a fact in and of itself, it is the same functor as we used before: this is actually a declaration that book(greatExpectations, dickens) is a fact. We will make structures distinct from facts by *contextualizing* them momentarily.

Prolog is an untyped language; hence there is no need to "declare" structures or to use them in a uniform way. In addition to the structure illustrated above, we may also use the following structures in the same program:

```
book(greatExpectations)
book(great,expectations,charles,dickens)
book(greatExpectations,author(dickens,charles))
book
```

Note that a nullary structure (e.g. book above) is called an atom, and is, in fact, a symbol. Also notice from the third example above that structures may be nested.

We see, then, that structures have exactly the same general appearance as predicates. We distinguish one from the other by context. Structures may only appear inside predicates, but predicates cannot usually appear inside other predicates. Consider the following Prolog fact:

```
owns(jean, book(greatExpectations, dickens)).
```

Here, owns/2 is not nested inside of anything else, thus it is a predicate. On the other hand, book/2 is nested inside of owns/2, thus it is a structure. jean, greatExpectations, dickens, and, from before, descartes and aristotle are also structures, namely nullary structures (atoms). Given this fact, we may now issue the following queries:

```
?- owns(jean,X).
X = book(greatExpectations, dickens)

?- owns(X, book(greatExpectations, dickens)).
X = jean

?- owns(jean, book(greatExpectations, X)).
X = dickens
```

```
?- owns(X, book(Y, dickens)).
X = jean, Y = greatExpectations
```

In Prolog, it is not valid to use a variable for the name of a functor, and thus the following query is not valid:

```
?- owns(jean, X(greatExpectations, dickens)).
```

In addition to structures, Prolog supports numbers and lists. Numbers basically work as atoms, but we will discuss further uses of atoms soon.

Lists in Prolog are enclosed in square brackets, and their elements separated by commas. For example,

```
[10, 20, 30]
```

represents the list consisting of 10, followed by 20, followed by 30. The empty list is written as `[]`. Prolog's "cons" operator is the vertical bar, `|`. The expression `[X|Y]` denotes the list whose elements consist of `X`, followed by the elements of `Y` (`Y` is assumed to be a list). For example, the list `[10, 20, 30]` may also be written `[10 | [20, 30]]`. We may also write any finite number of list elements before the `|`. For example, `[10,20,30]` can also be written as `[10, 20 | [30]]` and as `[10, 20, 30|[]]`. This flexible cons syntax allows us to specify queries such as `[10, 20 | X]`, assigning variables to only part of a list. Note that lists offer no power that structures do not—you could always define your own lists with a `pair/2` structure and `nil` atom, as in functional languages—but Prolog interpreters can usually optimize lists written with the standard Prolog syntax.

4 Unification and Search

In general, logic programming is an exercise in theorem proving. Given a set of clauses, we use resolution to prove them consistent or inconsistent, and use the example of consistency or proof of inconsistency to answer a query. Hence, logic programming language interpreters behave much like theorem-proving software.

In the abstract, given a set of clauses, we can use resolution to derive an inconsistency by combining clauses as we see fit. In other words, there is no fixed sequence of alternatives to try, and we are free to rely on our intuition in our search for a derivation of the empty clause.

However, in real programming tasks, we often prize predictable behavior, so that we may make assertions about the properties of programs. For this reason, logic programming languages take a deterministic approach to the resolution problem, and in the case of Prolog, the deterministic order is defined as part of the language.

Effective Prolog programming requires that we understand the mechanisms by which Prolog answers queries. We discuss those mechanisms in this section.

Execution of Prolog programs is based on two principles: *unification*, and *depth-first search*. Here, unification refers to Robinson's unification algorithm, which we saw in the previous module. Depth-first search refers to the strategy that Prolog employs when searching through the database. Essentially, Prolog attempts to satisfy a query by searching the database from top to bottom, attempting to unify each fact, and the head of each rule, with the query. If the query unifies with a fact, then Prolog returns the MGU, or simply "Yes" if the MGU is the empty substitution (i.e. if there are no variables in the query). If a query unifies with the head of the rule, then Prolog applies the MGU to the body of the rule and recursively attempts to satisfy the body of the rule. If the search terminates without finding a match to the query, Prolog returns "No".

This style of execution is so far removed from the step-wise execution we've seen before (the $\rightarrow$ morphism) that it's rarely useful to describe it in that way. Instead, we simply describe the unification and search algorithms; the surprising result is that by implementing these algorithms, we have implemented a programming language, even though no component seems to behave like an interpreter or compiler.

In the context of Prolog, unification is somewhat more complex than what we presented in Module 5. We present the details here.

Definition 2. (Unification Algorithm for Prolog Predicates)

Let the name R, possibly subscripted, range over relations (that is, predicates and structures).

Let O, possibly subscripted, range over objects (atoms).

Let X, possibly subscripted, range over variables.

Let T, possibly subscripted, range over terms (relations, objects, and variables).

Let S, possibly subscripted, range over substitutions.

Then the unification algorithm for Prolog predicates is as follows:

$$
\begin{aligned}
U(O_1, O_2) &= \begin{cases} [] & \text{if } O_1 = O_2 \\ \text{fail} & \text{otherwise} \end{cases} \\
U(O, R(T_1, \cdots, T_n)) &= \text{fail}, n \geq 1 \\
U(X, T) &= [T/X] \\
U(R_1(T_{1,1}, \cdots, T_{1,n}), R_2(T_{2,1}, \cdots, T_{2,m})) &= \begin{cases} \text{fail} & \text{if } R_1 \neq R_2 \\ \text{fail} & \text{if } m \neq n \\ S_1 S_2 \cdots S_n \text{ where } \begin{cases} S_1 & = U(T_{1,1}, T_{2,1}) \\ S_2 & = U(T_{1,2}S_1, T_{2,2}S_1) \\ \cdots & \\ S_n & = U(T_{1,n}(S_1 S_2 \cdots S_{n-1}), \\ & \quad T_{2,n}(S_1 S_2 \cdots S_{n-1})) \end{cases} & \text{otherwise} \end{cases} \\
U(T_1, T_2) &= U(T_2, T_1)
\end{aligned}
$$

The astute reader will notice that this formulation of unification omits the occurs-check. Recall that the occurs-check would place a condition on the rule for $U(X, T)$, which would require that X not occur in T if T is not equal to X. Indeed, omitting the occurs-check carries the same potential for trouble here as it does in the theory of type assignment. Nevertheless, Prolog omits the occurs-check because it is expensive to perform. As a result, it is possible to enter facts and queries into Prolog that would result in circular unification and cause Prolog to return infinite answers. For example, suppose we place the following rule in the database:

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

Then suppose that we issue the following query:

```
?- equal(X, f(X)).
X = f(f(f(f(f(f(f(f(f(f(...))))))))))
```

The result is a nonsensical substitution in which `X` is assigned an expression of infinite length.

This is less of a problem for Prolog than it is for type checking, because while it's critical for type checking to terminate (i.e., the type judgment must be decidable), Prolog is a Turing-complete language, so there exist programs that won't terminate anyway. Nonetheless, in some cases, we might like Prolog to perform the occurs-check despite its cost. In such cases, we may use the built-in predicate `unify_with_occurs_check/2`. Using this predicate, we would rewrite our equality predicate as follows:

```
equal(X,Y) :- unify_with_occurs_check(X,Y).
```

Unification is the mechanism by which Prolog matches queries against facts and rules in the database. To complete our understanding of Prolog's execution model, we now discuss the strategy by which Prolog searches the database for matches.

When attempting to answer a query, Prolog always searches the database from the top down. Prolog attempts to satisfy the predicates in a query one at a time, starting from the left and working to the right. If a predicate in the query matches the head of the rule, then the body of the rule is added to the front of the list of predicates to be satisfied (remember that `:-` is implication in reverse!), and Prolog recursively tries to satisfy the new query. In this way, Prolog's search strategy is based on depth-first search. When adding predicates in this way, because you may have other predicates with similar variable names already in the list of predicates to be resolved, it may be necessary to rename variables, similarly to while substituting in λ-applications, to avoid pairs of unrelated variables which have the same name being interpreted as the same variable.

Example 3. consider the following database:

```
p(X) :- q(X), r(X).
q(X) :- s(X), t(X).
s(a).
s(b).
s(c).
t(b).
t(c).
r(c).
```

Suppose that we then enter the following query:

```
?- p(X).
```

To satisfy this query, Prolog searches the database for a fact, or the head of a rule, that unifies with `p(X)`. The first (indeed, the only) match is the first rule, `p(X) :- q(X),r(X).`; the MGU is the empty substitution. Thus, Prolog replaces `p(X)` in the query with `q(X), r(X)` and attempts to answer the new query. Prolog searches the database for a match for `q(X)`, and finds the rule `q(X) :- s(X), t(X)`. Thus, it replaces `q(X)` with `s(X), t(X)` in the query; the query becomes `s(X), t(X), r(X)`, and Prolog attempts to satisfy `s(X)`. The first match is the rule `s(a)`; the MGU is [`a`/`X`]. Prolog applies the MGU to the remainder of the query and attempts to satisfy the query `t(a), r(a)`. As there is no match for `t(a)` in the database, the current answer fails, and Prolog searches for another answer to `s(X)`. The second match is `s(b).` with MGU [`b`/`X`]. Prolog applies the MGU and then attempts to satisfy `t(b), r(b)`. There is a match for `t(b)` in the database, but not for `r(b)`, and so this answer fails as well. Thus, Prolog backtracks and tries the third and final answer for `s(X)`, namely `s(c)`, with MGU [`c`/`X`]. Prolog applies the MGU and tries to satisfy `t(c), r(c)`. As both of these facts occur in the database, Prolog has found a successful match and returns the MGU, `X` = `c`.

Alternatively, this search with backtracking can be shown with indentation:

```
p(X)
 q(X), r(X)
   s(X), t(X), r(X)
     s(a), t(X), r(X) [a/X]
       t(a), r(a)
        fail
     s(b), t(X), r(X) [b/X]
       t(b), r(b)
         r(b)
          fail
     s(c), t(X), r(X) [c/X]
      t(c), r(c)
       r(c)
        solution: X=c
```

We can best illustrate Prolog's backtracking search strategy using a structure known as a tree of choices [3]. The following three examples were originally presented by Cohen [4].

Example 4. Suppose we have the following simple database:

```
1 p :- q, r, s.
2 p :- t, u.
3 q :- u.
4 t.
5 u.
6 p :- u.
```

Given this database, we may represent the query `?- p, t.` as the tree in Figure 1.

Each internal node in the tree represents the current query being evaluated. Every internal node has exactly six children, one for each entry in the database. If an entry in the database matches the first predicate in the query, then the query is updated and evaluated in the subtree with the same number as the position of the matching entry in the database. If the query is exhausted, we have found a successful match, and output "Yes". Thus, Prolog's search strategy is essentially a depth-first traversal of this tree.

Prolog's search is not guaranteed to terminate. Indeed, since Horn clauses are equivalent to computable functions, whether Prolog does or does not terminate is the halting problem.

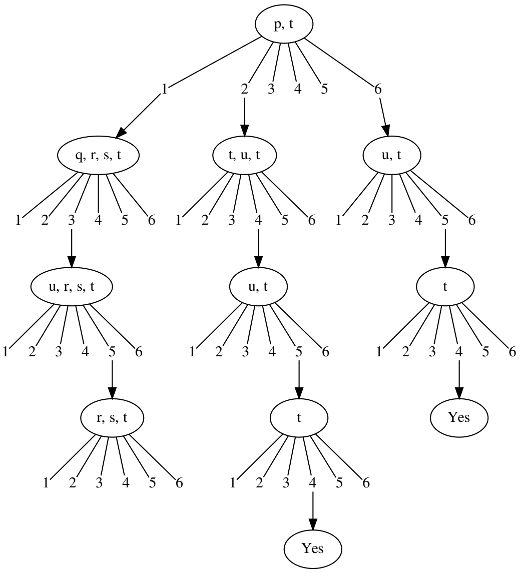

Figure 1: A Prolog search tree. Note that the facts are referred by the line number in which they appear.

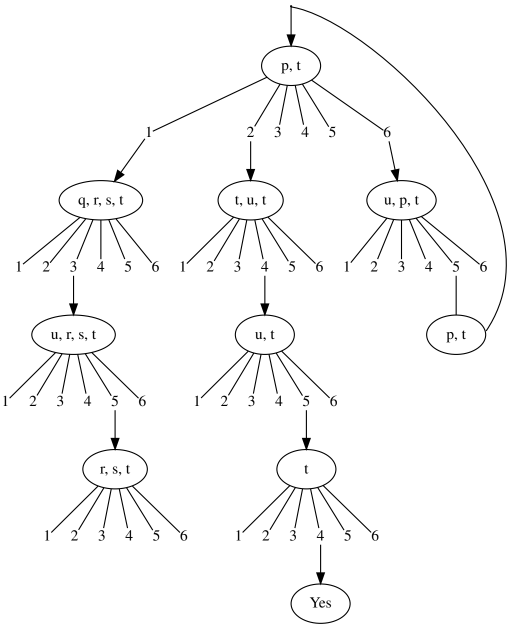

Figure 2: An infinite search tree with a successful match.

Example 5. Consider the following database, which is identical to the previous database, except for a change in the last entry:

```
1 p :- q, r, s.
2 p :- t, u.
3 q :- u.
4 t.
5 u.
6 p :- u, p.
```

The tree corresponding to the query `?- p, t` in this database is illustrated in Figure 2.

The addition of the predicate `p` to the end of the database introduces recursion into the search. Now, when Prolog matches the query against the rule `p :- u, p`, after elimination of `u` by rule 5, we obtain our original query again. Hence, Prolog recursively answers the query `?- p, t` again; since a success had previously been found, the result is an infinite sequence of "Yes" answers.

The behaviour of the query `?- p, t` is dependent upon the way in which the entries in the database. Suppose we reorder the database as follows:

```
1 p :- u, p.
2 p :- q, r, s.
3 p :- t, u.
4 q :- u.
5 t.
6 u.
```

Here, we have simply moved the final entry to the top of the database. As a result, we obtain the search tree in Figure 3.

In this tree, the infinite recursion appears first. Hence, when attempting to evaluate the query `?- p, t`, Prolog attempts to evaluate `?- u, p, t` and then `?- p, t` again. Thus, Prolog immediately gets caught in an infinite evaluation sequence, before ever generating a successful match for the query. Thus, in this case, Prolog falls into an "infinite loop" and never reaches the match indicated at the bottom of the tree.

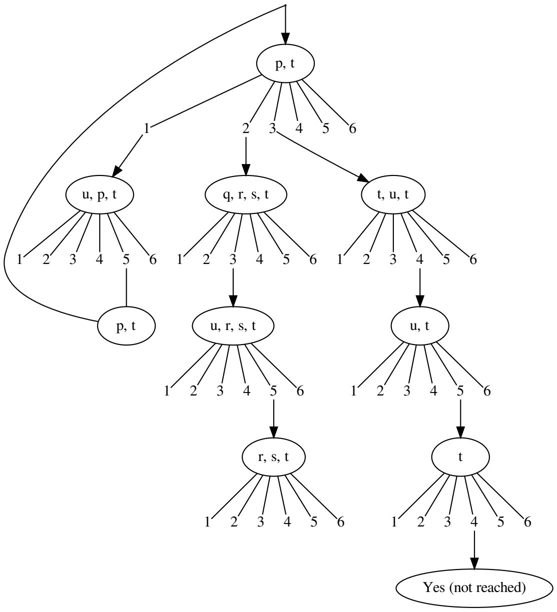

Figure 3: An infinite search tree without a successful match.

5 Values and Operators in Prolog

The structures and lists we've already discussed are values in Prolog, but Prolog also supports numbers. Prolog clauses may contain numbers, and operators over those numbers. Prolog's arithmetic comparison operators are `>`, `<`, `>=`, `=<`, `is`, `=:=` and `=\=`. The last two operators represent numeric equality and inequality, respectively. Note that less-than-or-equal-to is `=<`, not the more common `<=`.

For instance, the following clause makes `greaterThanTen(X)` a fact when `X` is greater than ten:

```
greaterThanTen(X) :- X >= 10.
```

However, Prolog cannot search for facts about numbers in the same way that it can for structures. For instance, the query `?- greaterThanTen(X).` fails, because `>=` does not appear as a rule in the database to search; `X` must already have a fixed value for `X >= 10` to be verifiable. That is, Prolog cannot *discover* a substitution for a variable to make a numerical comparison about numbers true, so some previous query needs to fix the variable to a value.

`is` and `=:=` are both numeric equality operators, differing only in how they treat integers as compared to floating-point numbers. `1 =:= 1.0` is true, but `1 is 1.0` is false. `is` is more common, simply because floating-point arithmetic is rare in Prolog in the first place. Prolog *can* unify using `is`, but only if one side has been fully resolved, and the other side is just a variable. For instance, `X is Y + Z` is unifiable only if `Y` and `Z` have already been resolved; if `X` and `Y` have already been resolved, it is of course mathematically trivial to discover the value of `Z`, but Prolog cannot unify in this way. That is, Prolog allows `U(X, n)`, where n is an arithmetic expression with no variables after substitution.

In addition, Prolog has an `=` operator, which simply forces two values to unify. For instance, we could rewrite `equal` above to:

```
equal(X, Y) :- X = Y.
```

Note that this is unification, which is distinct from numeric equality. If one of `X` or `Y` is a number, then `X = Y` only if `X is Y`. But, if either is not a number, then `X = Y` is true if they're structurally identical and `X is Y` is never true.

Most versions of Prolog also support strings, but we will not discuss them here, as they're no more powerful than lists of character codes, which we can already build with lists and numbers. Strings are, of course, a bit more practical than lists of character codes.

6 Programming in Prolog

In this section, we consider the task of writing real programs in Prolog. Whereas the languages we have considered so far have been based on writing and combining functions, the only entities we may define in Prolog are predicates, which are based on facts and rules.

To mimic the behavior of a function with a predicate, we create a predicate with arguments for each of the arguments of the function, as well as the result(s). We then use facts and rules to describe the characteristics of the output with respect to the input. For example, suppose we wish to mimic the behaviour of the function f with arguments a and b and result c. Then we would write a predicate `f(a, b, c)` and proceed to write facts and rules relating c to a and b. So, rather than saying that `f(a, b) = c`, which is meaningless since all predicates are boolean, we say that f relates a, b, and c.

We first consider the problem of appending one list to another. To append lists, we will invent a ternary predicate `append/3`, which we interpret as follows: `append(X, Y, Z)` means that `Z` is the result of appending `Y` to `X`. It remains to relate `X`, `Y`, and `Z`, as follows:

```
append([], Y, Y).
append([X|T], Y, [X|Z]) :- append(T, Y, Z).
```

From the first rule, we see that the result of appending any list after the empty list is the list itself. From the second rule, we see that, if `Z` is the result of appending `Y` to `T`, then `[X|Z]` is the result of appending `Y` to `[X|T]`. We may then invoke append as follows:

```
?- append([1,2,3], [4,5,6], R).
R = [1,2,3,4,5,6]
```

It's worth taking a moment to consider the style of programming here. We have not stated an algorithm for appending lists per se; rather, we have stated two facts:

- That the concatenation of an empty list and any list is the second list, and
- that if `Z` is the concatenation of `T` and `Y`, then `[X|Z]` is the concatenation of `[X|T]` and `Y`.

By reversing that, we can *discover* an algorithm for concatenating lists, by prepending one element at a time, but that algorithm is just the implied combination of these two rules and Prolog's search algorithm.

Exercise 1. Using the search algorithm described in Section 4 and the definition of `append/3` above, follow through the steps to resolve `append([1, 2, 3], [4, 5, 6], R)`.

Next, we will write a Prolog predicate to reverse a list. Using `append/3` above, we may implement a predicate `reverse/2` as follows:

```
reverse([], []).
reverse([X|Y], R) :- reverse(Y, Z), append(Z, [X], R).
```

Here, the reversal of the empty list is the empty list, and the reversal of any other list is the reversal of the tail of the list, followed by the head of the list. We may also implement `reverse` using an accumulator, as follows:

```
reverse(X, Y) :- reverse2(X, [], Y).
reverse2([], Y, Y).
reverse2([X|T], Y, R) :- reverse2(T, [X|Y], R).
```

Here, we repeatedly push list elements from the first argument of `reverse2` onto the second argument of `reverse2`. We continue until the first argument is empty, at which point the second argument contains the answer.

Again, what we've actually done is state some facts about lists, and it is Prolog's search algorithm that actually reverses lists. In the case of `reverse2`, it's unclear what fact `reverse2` even corresponds to, since one would not usually describe reversing a list in terms of an accumulator; the hallmark of an effective logic programmer is the ability to discover or invent these clauses that cause computation to proceed, though they don't correspond directly to any particular real fact.

Exercise 2. Using the search algorithm, follow through the steps to resolve `reverse([1, 2, 3], R)`.

7 Control Flow and the Cut

Consider now the problem of determining whether an object is a member of a list. To answer this question, we might implement a predicate `member/2`, as follows:

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

Note that the underscore, `_`, is a "wildcard" placeholder, indicating that the value occurring at the underscore is not being used.

A related problem is that of lookup in association lists. Under the assumption that an association list stores pairs of the form `pair(X,Y)`[3], we might implement `lookup/3` as follows:

```
lookup(X, [pair(X,R)|_], R).
lookup(X, [_|Y], R) :- lookup(X, Y, R).
```

Notice that, in the cases of both `member` and `lookup`, we do not need to explicitly handle the empty list. Since the empty list cannot match a pattern of the form `[X|Y]`, invoking member or lookup with a second argument equal to the empty list will simply cause Prolog to fall through both clauses and return "No".

While both `member` and `lookup` work as advertised, their behavior becomes somewhat dubious if the list passed as the second argument contains duplicate entries. If the element `X` occurs in the list `Y` multiple times, then the query `?- member(X,Y).` will return "Yes" once for each time it occurs. Similarly, if the pair `pair(X, _)` occurs multiple times in the list, then `lookup` will return multiple results. Depending on its intended use, this behavior of `lookup` may be appropriate, but if our association list is meant to be a map, it is incorrect. Similarly, in the case of `member`, we are generally interested only in whether the element `X` occurs in the list `Y`; if it does, a single "Yes" will suffice.

In fact, we've seen this problem before: the list of pairs we've made here is exactly equivalent in structure to a type environment, Γ, and if we were using Prolog to check types, then we would want `lookup` to have the same ordering constraint was we discussed for type environments. Ideally, `lookup(X, Γ, Z)` should be exactly the same relation as $\Gamma(\texttt{X}) = \texttt{Z}$.

To produce the expected behavior, we need some facility by which we can affect Prolog's search; that is, we need a control-flow facility. However, given the tree search paradigm by which Prolog operates, conventional control flow constructs (loops, conditionals, exceptions, etc.) do not seem appropriate, or even possible to express.

Aside: It would also be possible to accomplish this with a "not equal" predicate, but as it turns out, the solution we're implementing here is more powerful!

In Prolog, we alter control flow via a construct known as the *cut*, which is denoted by the predicate `!`. The effect of the cut is to commit Prolog to choices made so far in the search, thus pruning the search tree. Its behavior is best illustrated by example.

Example 6. Suppose we have the following database:

```
a :- b, !, c.
b :- d.
b.
d.
c :- e.
c.
e.
a :- d.
f :- a.
f.
```

Then consider the query `?- f.` The corresponding search tree is illustrated in Figure 4.

When Prolog attempts to satisfy a rule containing the cut, it first attempts to satisfy all of the predicates occurring before the cut. If all the predicates before the cut are satisfied when it reaches the cut, Prolog commits

[3] And remember, since structures can have any form we want, the use of `pair` to represent a pair is just for our own clarity. We could just as well name it `association`, or `foo`.

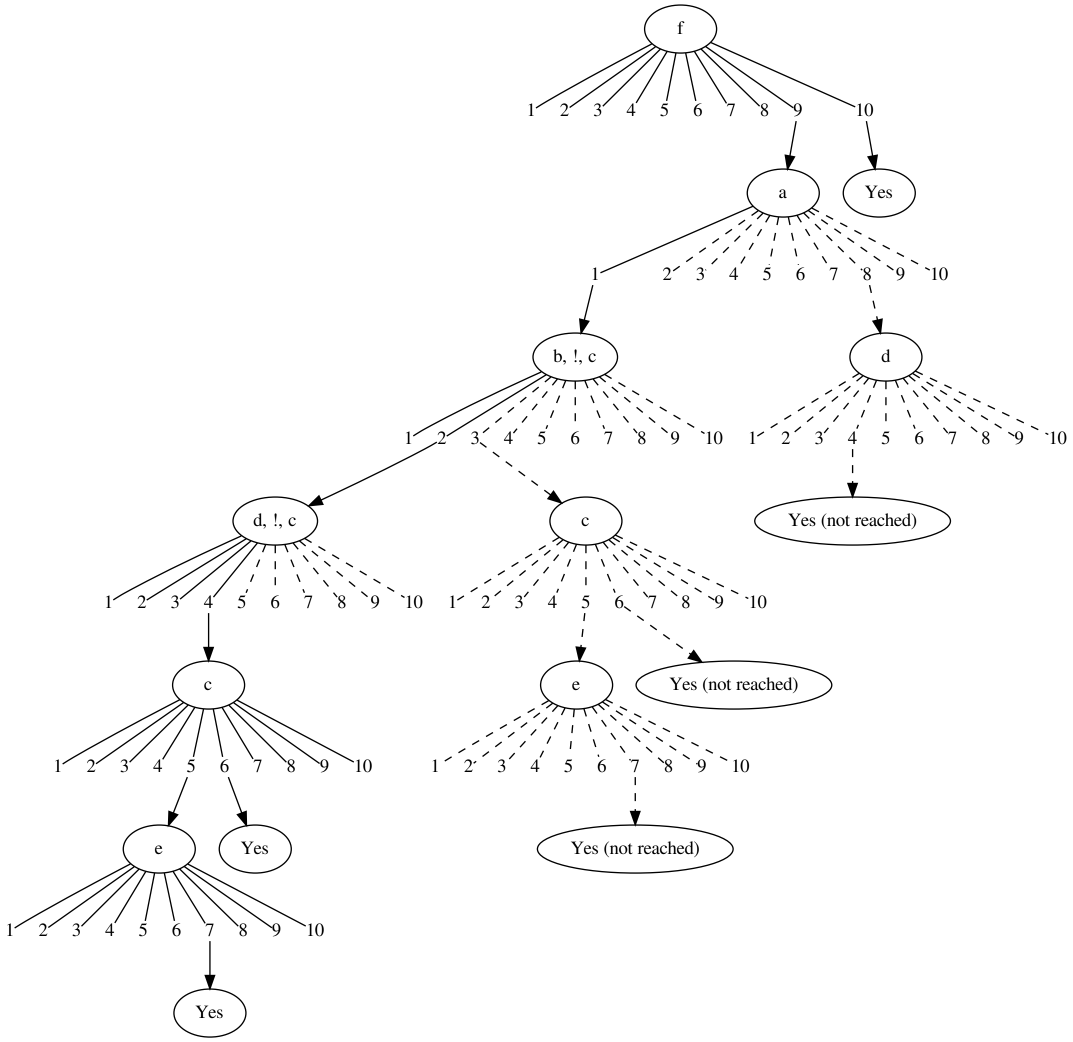

Figure 4: The cut. Dotted lines indicate paths which are not explored.

to all choices made to satisfy the predicates to the left of the cut. Prolog also commits to the choice of rule made to satisfy the predicate on the left hand side of the rule. For example, when Prolog attempts to satisfy the rule `a :- b, !, c`, it first attempts to satisfy `b`. If it succeeds, then Prolog commits to the first answer for `b`. Further, Prolog does not consider any other rules that might produce a solution for `a`. Visually, Prolog prunes the search tree at each subtree in which the query to be satisfied contains a cut. Further, if the query to be satisfied in a subtree contains a cut, then Prolog prunes its parent tree as well. In Figure 4, portions of the search tree that are pruned by the cut are indicated with dashed lines. Notice that three potential "Yes" answers are eliminated by the cut. Also note that, since the rule `a :- b, !, c` contains a cut, both the subtrees for `b, !, c` and for `a` are pruned. Since `d, !, c` contains a cut, both the subtrees for `d, !, c` and for `b, !, c` (which was already pruned) are pruned.

In general, there are three main uses of the cut:

1. to confirm that the "correct" match has been found;
2. to indicate "exceptions to the rule";
3. to prevent needless extra computation.

To illustrate the first use of the cut, let us consider the creation of a predicate `sum_to/2`, in which `sum_to(N, R)` binds `R` to the sum $1 + 2 + \cdots + \mathsf{N}$. We might define this predicate as follows:

```
sum_to(1, 1).
sum_to(N, R) :- M is N - 1, sum_to(M, P), R is P + N.
```

This formulation of `sum_to` will compute the correct value of `R` given `N`, but if the user presses `;` after Prolog returns the value of R, Prolog goes into an infinite loop, searching for other solutions. The reason for this behavior is that the constant `1` matches both the pattern `1` in the first rule and the pattern `N` in the second rule. Hence, when evaluating `?- sum_to(1, R)`, Prolog actually matches the query against both rules, even though only the first was intended. When Prolog matches `?- sum_to(1, R)` to the second rule, the result is an infinite recursion. Further, since the recursion ultimately reduces every query of `sum_to` to a query of `?- sum_to(1, R)`, every query of `sum_to` produces an infinite loop after the first answer.

To remedy this problem, we may use the cut to tell Prolog that the first rule is the "correct" rule to use when the first argument is `1`:

```
sum_to(1, 1) :- !.
sum_to(N, R) :- M is N - 1, sum_to(M, P), R is P + N.
```

In this case, when a query `?- sum_to(1, R)` matches the first rule, the cut prevents it from also matching the second rule. In this way, we prevent infinite recursion, and `sum_to` only returns a single, correct answer.

Our motivating example of `lookup/3` also fits this case, if we wish to look up only the first match. For that, we can rewrite `lookup` like so:

```
lookup(X, [pair(X,Y)|_], Y) :- !.
lookup(X, [_|Z], R) :- lookup(X, Z, R).
```

If the first case matches, so a pair is found, then it will only return that value. The cut will prevent it from trying to discover further solutions.

To illustrate the second use of the cut (exceptions), suppose that we have the following database:

```
smallMammal(cat).
smallMammal(rabbit).
goodPet(X) :- smallMammal(X).
```

In this database, we assert that cats and rabbits are small mammals, and that small mammals make good pets. Suppose now that we add the following fact to the database:

```
smallMammal(porcupine).
```

Working under the assumption that porcupines do not make good pets, we now have an exception to the rule that small mammals make good pets. To indicate this exceptional condition, we make use of a special predicate, `fail`, which always fails. On its own, `fail` is of limited use, as no rule containing `fail` can ever be satisfied. However, when combined with the cut, `fail` may be used to cause Prolog to return "No" when exceptional cases arise:

```
smallMammal(cat).
smallMammal(rabbit).
smallMammal(porcupine).
goodPet(X) :- X = porcupine, !, fail.
goodPet(X) :- smallMammal(X).
```

Since we know that `=` simply forces unification, this program may be written equivalently as follows:

```
smallMammal(cat).
smallMammal(rabbit).
smallMammal(porcupine).
goodPet(porcupine) :- !, fail.
goodPet(X) :- smallMammal(X).
```

If we now issue the query `?- goodPet(porcupine).`, Prolog will match the query against the first rule for `goodPet/1`. Since it matches, Prolog then processes the cut, which commits it to choosing the first rule. Finally Prolog processes the predicate `fail`, which makes the rule fail. Here, Prolog is not permitted to backtrack and try the second rule for `goodPet/1`, since it has processed the cut. Hence, the query `?- goodPet(porcupine).` results in an answer of "No". Note that it's critical that the cut come *before* the `fail`, because Prolog will not proceed to attempting to satisfy any other conditions once one has failed, so a cut (or anything else) after a `fail` will never be reached.

On the other hand, if we issue either the query `?- goodPet(cat).` or `?- goodPet(rabbit).`, then Prolog will not be able to match the query to the first rule for `goodPet/1`, because neither `cat` nor `rabbit` unifies with `porcupine`. Thus, Prolog does not process the cut; instead, it backtracks and tries the second rule, attempting to satisfy `?- smallMammal(cat).` and `?- smallMammal(rabbit).`, respectively. In both cases, Prolog succeeds and returns "Yes".

Finally, the cut may be used to prevent needless computation. Our motivating example of `member/2` demonstrates this use of the cut: we are interested only whether an object occurs in a list at all, and don't care if it appears multiple times.

Using the cut, we may reimplement `member/2` as follows:

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

Under this reformulated definition, Prolog aborts the search after the first match it finds, so the needless extra computation required to find subsequent matches is eliminated.

8 Negation

Even though negation is not required for programming with Horn clauses to be Turing-complete, Prolog provides a negation operator as a matter of convenience. Prolog's negation operator is the meta-predicate[4] `not/1`. Given a predicate `P`, the goal `not(P)` attempts to satisfy `P`, and returns success if and only if the attempt to satisfy P failed. Any bindings of variables to values that occurred during the evaluation of `P` are erased by the application of `not`. A simple example of the use of `not` is a rule stating that a mineral is a thing that is not animal or vegetable:

```
mineral(X) :- thing(X), not(animal(X)), not(vegetable(X)).
```

Here, the rule defining `mineral/1` succeeds when `thing/1` succeeds and both `animal(X)` and `vegetable(X)` fail.

Prolog programmers may make other meta-predicates by use of the `call` meta-predicate, which simply treats its argument as a goal and attempts to solve it. For instance, the `id` meta-predicate is:

```
id(X) :- call(X).
```

With this definition, if `sum(descartes)` is true, then `id(sum(descartes))` is also true, as is `id(id(sum(descartes)))`, etc.

`call` is rarely used, but with it, the cut, and `fail`, we can define `not` directly in Prolog as follows:

```
not(X) :- call(X), !, fail.
not(X).
```

[4] `not/1` is called a meta-predicate because its argument is Prolog code in the form of a predicate, rather than data.

not attempts to satisfy its argument. If it succeeds, it forces Prolog to return failure; otherwise, if the argument cannot be satisfied, then the first rule fails, and Prolog tries the second rule. As the second rule always succeeds, Prolog returns "Yes".

Even though not tends to capture the behavior we generally associate with logical negation, it is not a true logical negation operator. Consider the following database:

```
animal(dog).
vegetable(carrot).
both(X) :- not(not(animal(X))), vegetable(X).
```

If not were a true logical negation, then we could equivalently define both/1 above as

```
both(X) :- animal(X), vegetable(X).
```

and expect the query ?- both(X). to return "No", as no symbol in the database is both an animal and a vegetable. However, when we use the original formulation of both/1, with the double negation, the query ?- both(X). returns the answer X = carrot. To see this, consider how Prolog attempts to evaluate the query. It first needs to satisfy the query ?- animal(X)., which it does by setting X equal to dog. Thus, the goal not(animal(X)) fails, since an answer was found. But, a failed query does not generate a substitution, so X is uninstantiated. Since the goal not(animal(X)) fails, the goal not(not(animal(X))) succeeds, but X remains uninstantiated. Hence, Prolog now attempts to satisfy the query vegetable(X). Since X is uninstantiated, Prolog is free to set X equal to carrot, thus satisfying the query.

Reversing the order of the predicates corrects the behavior, since X will have a substitution when the not is investigated:

```
both(X) :- vegetable(X), not(not(animal(X))).
```

Thus, it is important to be sure which variables will have already been substituted when not is used.

Because not does not have true logical negation semantics, some newer Prolog implementations prefer to use the name \+ (called "cannot be proven") for this operator instead.

9 Example: Sorting a List

In this section we present more examples of Prolog programming through the extended example of sorting a list. We shall focus our efforts on a naïve, purely declarative notion of sorting:

```
sort(X, Y) :- permute(X, Y), sorted(Y), !.
```

That is, Y is a sorted reordering of X if Y is a reordering (or permutation) of X, and Y is sorted.

This implementation of sort/2 is based on the notion of generators and tests. The generator, permute/2, generates all permutations of the list Y (albeit, again, simply by stating the permutations as facts; we never write algorithms directly in Prolog). The test, sorted/1, determines whether a particular permutation is sorted. In this way, we reject all unsorted permutations of X. Because there may be several sorted permutations of X (if the same value appears multiple times in the list), and only one is relevant, we use a cut to terminate our search once we have found one sorted permutation of X.

The implementation of sorted/1 is quite easy, and is outlined below:

```
sorted([]).
sorted([_]).
sorted([X,Y|Z]) :- X =< Y, sorted([Y|Z]).
```

In words, empty and singleton lists are sorted; any other list is sorted if its first element is less than or equal to its second, and its tail is sorted.

We give two implementations of `permute/2`. In our first implementation, we create a generator `to_X/2`, where `to_X(N, R)` is taken to mean "`R` is an element of the set $\{0, 1, \cdots, \texttt{N} - 1\}$". The implementation of `to_X` is outlined below:

```
to_X(X, _) :- X =< 0, !, fail.
to_X(_, 0).
to_X(X, R) :- X1 is X - 1, to_X(X1, M), R is M + 1.
```

Invocation of `to_X` yields the expected result:

```
?- to_X(2, R).
R = 0 ;
R = 1 ;
No
```

Mathematically, this is equivalent to

```
to_X(X, R) :- X => 1, X =< R - 1.
```

but Prolog cannot discover resolutions for `X` and `R` with only these bounds.

Next we give a predicate for finding the length of a list:

```
length([], 0).
length([_|T] , R) :- length(T, Y), R is Y + 1.
```

Our intent is to use `to_X` to choose an element to remove from the list we wish to permute. To accomplish this feat, we define a predicate `select/4`, where `select(L, N, A, B)` binds `A` to the `N`-th element of `L`, and binds `B` to `L` with `A` removed:

```
select([H|T], 0, H, T) :- !.
select([H|T], N, A, [H|B]) :- N1 is N - 1, select(T, N1, A, B).
```

With `select` in place, we may now define `permute` as follows:

```
permute([], []) :- !.
permute(L, R) :- length(L, Len), to_X(Len, I), select(L, I, A, B), permute(B, BP), append([A], BP, R).
```

That is, to permute a list, we find its length, `Len`, and bind `I` to each integer in $\{0, \cdots, \texttt{len} - 1\}$. For each `I`, we take the `I`-th element out of `L` and permute the remainder. Note how this style of stating facts has made the "for each" completely implicit: since it is fact that `to_X(Len, I)` *for each* `I < Len`, Prolog can simply backtrack to another resolution of `to_X` as many times as it needs to, behaving like a loop with no explicit loops or even recursion specified. Finally, we put the `I`-th element at the front of the permuted remainder.

We can also write `permute` in a much shorter way:

```
permute([], []).
permute(L, [H|T]) :- append(U, [H|V], L), append(U, V, W), permute(W, T).
```

Here, we are using `append` as a generator. Rather than supply the arguments and retrieve a result, we supply the result, `L`, to `append`. `append` then generates all lists whose concatenation is equal to `L`. We then capture an element `H` from the middle of `L`, and permute the remainder, placing `H` at the front of the list.

With `sorted/1` and `permute/2` implemented, `sort/2` now works, but as you can doubtless imagine, it is hilariously inefficient. This naïve implementation of `sort/2` will find a sorted list by checking every possible permutation, so it takes $O(n!)$ time where n is the length of the list, which is exponential. To sort things efficiently takes far more thought.

10 Example 2: Merge-Sorting a List

To make our sort more efficient, let's implement the classic merge-sort algorithm. Merge sort is a recursive algorithm: to merge sort a list, we split it in half, recursively merge sort each half, and then merge the result. This requires three sub-parts: dividing a list in half, merging two sorted lists, and merge sort itself. We will address them in that order. Note that this example was inspired by but significantly changed from an example given in Roman Barták's *On-Line Guide to Prolog Programming* [1].

First, dividing a list. When implementing merge sort, one typically divides a list into the first n elements and the last m elements, where $n+m$ is the length of the list. However, any fair division works, and a style of division that is much more easily expressed in Prolog is dividing a list into the elements with even and odd indices. We express this with two mutually-recursive predicates, `divide/3` and `divide2/3`:

```
divide([], [], []).
divide([X|LI], [X|L1], L2) :- divide2(LI, L1, L2).

divide2([], [], []).
divide2([X|LI], L1, [X|L2]) :- divide(LI, L1, L2).
```

`divide/3` declares that its first argument is divided into its second and third arguments if the first element of its first argument is the first element of its second argument (`L1`), and the remainder are divided by `divide2/3`. `divide2/3` is similar, with the only distinction being that it requires the first element to be in the last list, `L2`. In either case, the division of an empty list is simply two empty lists. Consider the division of a simple list `[1,2,3]`:

```
divide([1, 2, 3], A, B)
divide2(LI, L1, L2) [1/X] [[2,3]/LI] [[X|L1]/A] [L2/B]
divide2([2,3], L1, L2)
divide(LI_, L1_, L2_) [2/X] [[3]/LI_] [L1_/L1] [[X|L2_]/L2]
divide([3], L1_, L2_)
divide2(LI_2, L1_2, L2_2) [3/X] [[]/LI_2] [[X|L1_2]/L1_] [L2_2/L2_]
divide2([], L1_2, L2_2)
L1_2 = [], L2_2 = []
```

By following the substitutions backwards, we can get that `A = [1,3]` and `B = [2]`.

Next, we need to be able to merge sorted lists. First, let's handle the base case: if we merge an empty list with any list `L`, the result is `L`:

```
merge([],L,L) :- !.
merge(L,[],L).
```

We used the cut simply to avoid processing both cases when merging two empty lists. Now, we simply need cases to merge two non-empty lists, by choosing the lesser element:

```
merge([X|T1],[Y|T2],[X|T]) :- X=<Y, merge(T1,[Y|T2],T).
merge([X|T1],[Y|T2],[Y|T]) :- X>Y, merge([X|T1],T2,T).
```

These rules say that the lists `[X|T1]` and `[Y|T2]` merge to `[X|T]` if `X=<Y`, and `[Y|T]` otherwise. `T` is the merger of the remainder of the lists.

Finally, we use these two algorithms to implement merge sort. Merge sort's base case is when the list is empty or of length one, and in either of these cases, the list is trivially sorted:

```
merge_sort([], []) :- !.
merge_sort([X], [X]) :- !.
```

In both cases, we use the cut to make sure that the recursive case cannot apply. The merge sort algorithm itself is the last case, and it is written as a division, two recursive merge sorts, and a merge, like so:

```
merge_sort(List,Sorted):-
    divide(List,L1,L2),
    merge_sort(L1,Sorted1), merge_sort(L2,Sorted2),
    merge(Sorted1,Sorted2,Sorted),
    !.
```

Once again, we use the cut to free Prolog of any need to search for alternative solutions; there is only one sorting of a list.

Note that `merge_sort` always generates the same result as `sort`, but when Prolog's search algorithm is applied to it, it will usually find that result more quickly. This is why effective Prolog programming requires an understanding of the search algorithm.

Exercise 3. Follow through the steps to resolve `merge_sort([3, 1, 2], S)`.

11 Manipulating the Database

Prolog provides facilities by which we may programmatically alter the contents of the database. This is done primarily via the predicates asserta/1, assertz/1, and retract/1.

The predicates asserta and assertz add a clause to the database. Given a clause X, asserta(X) adds X to the beginning of the database, while assertz(X) adds X to the end of the database. Where a clause appears in the database is, of course, important because of the well-specified resolution order of Prolog.

The predicate retract removes a clause from the database. Given a clause X, retract(X) removes the first clause matching X from the database. If no clause matching X is found, then retract(X) fails.

Note that the effects of asserta, assertz, and retract are not undone by backtracking. Thus, if we wish to undo the effects of asserta or assertz, we must do so explicitly, via a call to retract, and vice versa. Undoing retract can be essentially impossible, since we can only add a clause to the beginning or the end of the database, not the middle.

Excessive use of asserta, assertz, and retract tends to lead to highly unreadable and difficult to understand programs, and is discouraged.

12 Difference Lists

In Section 6, we created a predicate to reverse a list, as follows:

```
reverse([], []).
reverse([X|Y], R) :- reverse(Y, Z), append(Z, [X], R).
```

We also introduced a version of reverse based on accumulators, as follows:

```
reverse(X, Y) :- reverse2(X, [], Y).
reverse2([], Y, Y).
reverse2([X|T], Y, R) :- reverse2(T, [X|Y], R).
```

Even though, as declarative programmers, we would prefer not to worry about efficiency, when we compare these two formulations of reverse/2, we cannot help but notice that the second formulation is more efficient. Assuming a linear implementation of append/3, we see that the second formulation of reverse/2 requires only a linear number of query evaluations, while the first formulation requires a quadratic number of query evaluations.

In general, the use of accumulators is closely related to the notion of tail recursion in functional programming, and predicates based on accumulators are often more efficient than their counterparts without accumulators. However, accumulators can pose problems of their own. Consider the following predicate, which is based on accumulators, and whose purpose is to increment each element of a list:

```
incall(X, Y) :- incall2(X, [], Y).
incall2([], Y, Y).
incall2([X|T], Y, R) :- Z is X + 1, incall2(T, [Z|Y], R).
```

When we invoke incall/2, the result is as follows:

```
incall([1,2,3], Y).
Y = [4,3,2]
```

We see that incall/2 does indeed increment all of the elements of the list; however, the list returned by incall/2 is the reverse of what we intended! It is reasonably clear why this happened: at each step of the execution, we simply removed an element from the head of the input, incremented it, and placed it at the head of the accumulator. Thus, elements occurring later in the input are pushed onto the accumulator later, and therefore occur earlier in the accumulator list than elements which appeared before them in the input. The result is a list reversal.

To reap the benefits of programming with accumulators without the inconvenience imposed by the list reversals, Prolog programmers use a technique known as difference lists, which we discuss in this section.

We may think of a difference list as a list with an uninstantiated tail. For example, by using difference lists we would represent the list [1,2,3] as [1,2,3|X]. We may then use X as an argument to other computation. Effectively, what we obtain through X is a "pointer to the tail" of the list, often called "the hole". Predicates using difference lists take as arguments both the head and the hole, with the expectation that the hole is still a variable while the head is not. By instantiating X, we may add elements to the tail of the list. Further, if we instantiate X to a difference list, the the result is still a difference list. This fact suggests the following implementation of `append`:

```
appendDL(List1, Hole1, List2, Hole2, List3, Hole3) :- Hole1 = List2, List3 = List1, Hole3 = Hole2.
```

Here, each difference list is represented by two parameters, `List` and `Hole`, representing the list and the hole at its tail, respectively. The result is that the list represented by `List3` and `Hole3` is bound to the result of appending the second list to the first. We may implement `appendDL` more succinctly as follows:

```
appendDL(List1, List2, List2, Hole2, List1, Hole2).
```

Our immediate observation is striking: this implementation of append runs in constant time! Since a caller had to retain a reference, so to speak, to both the beginning and tail of the list, all Prolog needs to do is unify them, with no extra step of walking through the list. We can then finalize the list by unifying the final hole with the empty list, again in constant time. Appending linked lists is usually constant time in imperative languages, but this is the first constant-time append operation that we have encountered in this course. This fact alone suggests that difference lists are indeed a valuable tool.

Consider now our motivating example of incrementing the elements of a list via accumulators. To keep our code clean, we will represent a difference list by a structure with functor `dl` and components for the list and its uninstantiated tail. Using difference lists, we implement `incall` as follows:

```
incall(X, Y) :- incall2(X, dl(Z, Z), dl(Y, [])).
incall2([], Y, Y).
incall2([X|T], dl(AL, [X1|AH]), R) :- X1 is X + 1, incall2(T, dl(AL, AH), R).
```

Here, we use a difference list as an accumulator, and as a placeholder for the result. When we invoke `incall`, `incall` calls `incall2` and sets the accumulator equal to the empty difference list (denoted `dl(Z, Z)`, since an empty list's head is its tail). As `incall2` traverses the list, it repeatedly replaces the hole in the accumulator with a singleton difference list, representing the incremented head of the current list. When the input list is exhausted, `incall2` simply returns the accumulator. `incall` then fills the hole in the result with the empty list (`[]`), thus converting the result to a normal list. Thus, when we invoke `incall`, as in the following:

```
?- incall([1,2,3], Y).
```

we get the desired result:

```
Y = [2,3,4]
```

Let's go over that example in greater detail, by showing the steps of resolution. As there are no backtracking steps in this example, we will simply show each step in order:

```
incall([1, 2, 3], Y)
incall2(X, dl(Z, Z), dl(Y, [])) [[1, 2, 3]/X]
incall2([1, 2, 3], dl(Z, Z), dl(Y, []))
X1 is X + 1, incall2(T, dl(AL, AH), R) [1/X] [[2, 3]/T] [Z/AL] [[X1|AH]/Z] [dl(Y, [])/R]
X1 is 1 + 1, incall2([2, 3], dl([X1|AH], AH), dl(Y, []))
incall2([2, 3], dl([X1|AH], AH), dl(Y, [])) [2/X1]
incall2([2, 3], dl([2|AH], AH), dl(Y, []))
X1 is X + 1, incall2(T, dl(AL, AH_), R) [2/X] [[3]/T] [[2|AH]/AL] [[X1|AH_]/AH] [dl(Y, [])/R]
X1 is 2 + 1, incall2([3], dl([2,X1|AH_], AH_), dl(Y, []))
incall2([3], dl([2,X1|AH_], AH_), dl(Y, [])) [3/X1]
incall2([3], dl([2,3|AH_], AH_), dl(Y, []))
X1 is X + 1, incall2(T, dl(AL, AH), R) [3/X] [[]/T] [[2,3|AH_]/AL] [[X1|AH]/AH_], [dl(Y, [])/R]
X1 is 3 + 1, incall2([], dl([2,3,X1|AH], AH), dl(Y, []))
incall2([], dl([2,3,X1|AH], AH), dl(Y, [])) [4/X1]
incall2([], dl([2,3,4|AH], AH), dl(Y, []))
 (unify dl([2,3,4|AH], AH) and dl(Y, []))
Y = [2,3,4]
```

In this way, we see that we may use difference lists to write programs with accumulators, in which the accumulators grow at the tail rather than at the head. For this reason, the result is not, as before, the reverse of what we wanted.

13 Miscellany

To load a source file into Prolog, we use the predicate `consult/1`. A call to `consult(X)` loads the file "`X.pl`" into the database. A convenient shorthand for `consult(X).` is simply `[X].`.

To access Prolog's help facility, we issue the query `?- help(T).`, where `T` is our desired help topic. To search the help files, we issue the query `?- apropos(T)`, where `T` is our desired search term. Comments in Prolog begin with `%` and extend to the end of the line. C-style comments, beginning with `/*` and ending with `*/`, are also available. To display data to the screen, we use Prolog's `display/1` predicate. For character data, we may also use `put/1`. To print a newline to the screen, we use `nl/0`. In this way, Prolog programs can be made to behave like more traditional programs. In fact, there are Prolog implementations with support for graphics!

14 Query Programming and Datalog

A few of the words we used in this module may be familiar to you if you've studied databases. In particular, we call the set of clauses a *database*, and call our predicates and structures *relations*. These terms are the same as are used in *relational databases*, and are, in fact, used in exactly the same way. While meaning only to discuss programming in predicate logic, we have accidentally stumbled into a larger paradigm: *query programming*.

Although logic programming and relational databases were developed independently, when their relationship was discovered, attempts were made to describe a subset of Prolog which fit the use case of databases. In particular, the goal was to find a large and useful subset of Prolog which is *not* Turing-complete, since it is desirable when simply asking questions about data to ensure that the questions will be answered. These subsets are broadly called *Datalog*. Datalog is syntactically a subset of Prolog, and semantically mostly a subset of Prolog, except that certain programs which must not halt in Prolog instead *must* halt in Datalog. All queries which halt in Prolog and are valid in Datalog yield the same set of results.

Exactly what restrictions Datalog has isn't precisely defined—Datalog isn't an independent programming language, but a family of non-Turing-complete subsets of Prolog—but at least the following restrictions are common[5]:

1. The cut operator, `fail`, and `call` are removed.
2. Queries are always run to completion and give all results, rather than interactively requesting further results, and the results are not guaranteed to be in any order (although it's not uncommon to explicitly sort them as a final step after querying).
3. Structures other than atoms are removed.
4. Fully recursive queries (e.g., a predicate `A(X)` reached while trying to satisfy the predicate `A(X)`) are rejected.
5. Every variable appearing in the head of a rule must appear in the argument of a non-negated predicate in the body of the rule.
6. Negation is *stratified*: If the rules for predicate x include a negation of predicate y, then there must be no path through the rules by which predicate y refers to predicate x.
7. Numeric operators are also stratified: If the rules for predicate x include a numeric operator over variable Z, then either the result must be guaranteed to be closer to zero, or its use in predicates must stratify as with negation. Most versions of Datalog only allow positive numbers and/or only allow integers to make the closer-to-zero verification easier.
8. Lists, if allowed at all, must be stratified like numbers, either by reducing towards the empty list or by stratified use of predicates.

In short, with the exception of rules 1 and 2, these rules guarantee that taking a step in resolving a query always yields a lesser query, in the sense that it has fewer steps until facts are reached. This renders the language non-Turing-complete, but still useful. All queries in Datalog terminate.

Rule 3 removes the ability to create recursive structures (by removing all structures), and thus removes that path to infinite recursion. Note that we can still express interesting structures, but in a different way. For instance, to express

```
owns(jean, book(greatExpectations, dickens)).
```

we may instead give both people and books some kind of identifier, and then express the ownership relation over those identifiers, like so:

```
person(24601, jean).
book(9564, greatExpectations, dickens).
owns(24601, 9564).
```

Since the number of atoms in any given database is finite, rule 3 and 4 together guarantee that a query with only positive predicates over atoms (i.e., no numbers or lists) will eventually terminate. Rule 5 guarantees that no predicates are themselves infinite, since variables must be resolved. Rules 6, 7, and 8 allow us to add negation, numbers, and lists, respectively, by guaranteeing that each approach a base case.

The implication of rules 1 and 2 are more interesting: Without the cut, and without any particular guarantee on the order of results, the exact order in which we resolve a query is no longer constrained. In Prolog, it was necessary to follow the search algorithm precisely, because the result was only correct if we did. Thus, we couldn't optimize a query[5]; only one implementation of the algorithm is correct. Moreover, some queries and databases which in Prolog would expand infinitely are allowed and must terminate in Datalog, and the restrictions to Datalog's structure guarantee that this is possible. In spite of being a simpler language, Datalog is much more complicated to implement, because we *are* allowed to optimize the query, and such optimizations are required to guarantee termination.

But, before we get to query optimization, let's look at the other side of this history: databases.

15 Relational Databases

Relational databases were created completely independently of Prolog [2], and have their own foundational logic, called the *relational algebra*. In addition to their own foundational logic, they have their own defining language, the *Structured Query Language*, or SQL, sometimes pronounced as a homophone with "sequel". SQL is less a programming language than an entire family of related programming languages, as each relational database engine has its own customized version of SQL; the core, however, is standardized. This course is not intended to compete with CS348 or CS448, so we will not describe SQL at all, and will focus on relational algebra only as it relates to Datalog. Datalog is strictly more powerful than the relational algebra[6], so we will describe the relational algebra in terms of Datalog clauses.

The data in relational algebra is sets of *relations*, which are precisely the kinds of relations we've defined, i.e., functions. In relational database terminology, relations which are only defined as facts are called *tables*, and relations which are defined (at least in part) as rules (Horn clauses with bodies) are called *views*. These are sometimes also called *extensional database predicates* and *intensional database predicates*, respectively. A single fact within a table is called a *row*, and may not contain variables, only atoms or primitive data. This restriction is also in Datalog, implied by Datalog rule 5.

Consider a database which keeps track of employees and their supervisory relationship. This database might have two tables: one which tracks employee information, such as the employee's identification number and name, and a second which tracks who supervises whom. For instance, consider the following database:

```
employee(34, dantes).
employee(5, villefort).
employee(1, napoleon).

supervisor(1, 5).
supervisor(5, 34).
```

[5] Technically, certain optimizations are allowed, but they're very limited.

[6] Note that as there are many variants of SQL, it is not generally true that Datalog is more powerful than (a particular) SQL.

The `employee/2` table states that the employee `dantes` has employee number 34, the employee `villefort` has employee number 5, and the employee `napoleon` has, of course, employee number 1. The `supervisor/2` table states that `napoleon` supervises `villefort` (`supervisor(1, 5)`), and `villefort` supervises `dantes` (`supervisor(5, 34)`).

> **Aside:** This database is not a useful summary of *The Count of Monte Cristo*.

A useful view may be the supervisory relationship by name, instead of by number. In relational algebra terms, this is the *join* of the two tables:

```
supervisor_by_name(X, Y) :- employee(N, X), employee(M, Y), supervisor(N, M).
```

In a relational database, each argument to a predicate is called a *field* of the relation, and has a name, and usually a restricted type. Joining two (or more) tables is creating a new relation which is defined when some fields are equal in the original relations. In this case, `supervisor_by_name/2` joins twice: once with the first field of `supervisor/2` and the first field of `employee/2`, and once with the second field of `supervisor/2` and the first field of `employee/2`. The *natural join* of two tables is simply the join in which the fields which must match are those with the same names, which is not meaningful to us as Datalog fields do not have names.

In addition, relational databases usually have *unique* and *key* fields. For our purposes, there is no distinction between the two concepts; a key field is simply a unique field. This simply restrict how relations are added to the table, preventing two relations from having the same value for that field. For instance, in our above table, nothing prevents us from adding this relation:

```
employee(34, faria).
```

Doing so, however, would yield strange results, since two employees now have the same employee number. It was presumably our intention that no two employees have the same employee number—this is, of course, the point of using such numbers instead of employee names—so a relational database engine would prevent this row from being added to the database. We can instead add `faria` like so:

```
employee(27, faria).
```

and with that row, we can additionally add a supervisor relation:

```
supervisor(5, 27).
```

Queries with variables are called *selections*, because they *select* values for the variables which satisfy the query. For instance, we can find every employee who is supervised by employee 5 (`villefort`) like so:

```
?- supervisor_by_name(villefort, X).
X = dantes
X = faria
```

In the original theory of relational algebra, recursive views and queries were not allowed. In practice, however, there are as many extensions of relational algebra (and of SQL) as there are relational database systems, and most of them allow recursive views in some form. For instance, many would allow us to define a view for whether a named employee is higher in the supervisory hierarchy than another. We can split this into its two components: the relation of the name to the employee number, and the recursive relation of two employee numbers via `supervisor`:

```
higher(X, Y) :- employee(NX, X), employee(NY, Y), nhigher(NX, NY).

nhigher(M, N) :- nhigher(P, N), supervisor(M, P).
nhigher(M, N) :- supervisor(M, N).
```

We can now query all employees under `napoleon` at any level of the hierarchy:

```
?- higher(napoleon, X).
X = dantes
X = faria
X = villefort
```

Note that in Prolog, this query would not complete, because we've structured its recursion before its base case.

Some relational database systems support Datalog directly, but more frequently, they support a modified version of SQL inspired by Datalog.

16 Query Optimization

Consider the query `?- nhigher(1, X).` in Prolog, shown partially here:

```
1 nhigher(1, X)
2  nhigher(P, N), supervisor(M, P) [1/M] [X/N]
3   nhigher(P, X), supervisor(1, P)
4    nhigher(P_, N), supervisor(M, P_) [P/M] [X/N], supervisor(1, P)
5     nhigher(P_, X), supervisor(P, P_), supervisor(1, P)
6      nhigher(P_2, N), supervisor(M, P_2) [P_/M] [X/N], supervisor(P, P_), supervisor(1, P)
7       nhigher(P_2, X), supervisor(P_, P_2), supervisor(P, P_), supervisor(1, P)
8        ...
```

This query will never resolve, because `nhigher(P, N)` is listed before `supervisor(M, P)` in a clause defining `nhigher/2`, so it will expand to an infinite series of `supervisor/2` relations. If we had simply put `supervisor(M, P)` before `nhigher(P, N)`, then Prolog would have resolved this easily. It is the role of a *query optimizer* to reorder predicates in Datalog, making such queries not just more optimal, but in cases like this one, possible.

Query optimization is an enormous field, and we will only touch on the basic concepts here. Generally speaking, at any time, a query optimizer is presented with a list of predicates it must resolve, and must choose one to resolve first. For instance, on line 6 of the above example, a query optimizer must choose whether to resolve `nhigher(P, N)` or `supervisor(M, P)` first. The structure of Datalog guarantees that either ordering is valid (except for halting), but one may be faster (in this case, infinitely faster).

> **Aside:** You will see query optimization discussed in very different terms than this section uses, simply because it was originally developed for SQL and relational databases, not Datalog. The concept is the same, but the details vary greatly.

Given a list of predicates to be resolved, a query optimizer orders them based first on the structure of the predicates, and second on a cost model. The cost model is completely specific to the specific query optimizer and database engine, so we won't discuss it here. Structurally, observe that `nhigher/2` consists of a recursive join over `supervisor/2`: we join its first argument to its second. *Join optimization* consists of choosing joins which *reduce* the size of our query before joins which *increase* the size of our query. The "size" in this case is the number of unrestricted variables or, if equal, the number of predicates in the resulting query. Consider the partially-evaluated query `nhigher(P, X), supervisor(1, P)`. Since `supervisor/2` is a table (it has only facts), if we resolve it, we will not introduce any new predicates, and will resolve the value of `P`. On the other hand, if we resolve `nhigher(P, X)` first, its second rule will introduce one new variable, as well as a new predicate. Thus, we choose to resolve `supervisor(1, P)` before `nhigher(P, X)`. When options are equal in these terms, it's necessary to use a heuristic cost model.

A similar model can also used to select from multiple resolutions of a predicate. For instance, we would choose to expand `nhigher(1, X)` to `supervisor(M, N) [1/M] [X/N]` before `nhigher(P, N), supervisor(M, P) [1/M] [X/N]`, because the latter introduces more variables and predicates. This is less important, since ultimately, in Datalog, every resolution must be explored, but SQL allows, for instance, requesting just one matching result, and this optimization can be useful for doing so.

Now, let's consider the same query as above, but with a reordering step to bring the best query by the above model to the front, which we will mark with `/* OPTIMIZATION */`:

```
nhigher(1, X)
 /* OPTIMIZATION: Choose supervisor(M, N) [1/M] [X/N] first */
 supervisor(M, N) [1/M] [X/N]
  supervisor(1, X)
   supervisor(1, 5)
    solution 1: X=5
 nhigher(P, N), supervisor(M, P) [1/M] [X/N]
  nhigher(P, X), supervisor(1, P)
   /* OPTIMIZATION: Choose supervisor(1, P) first */
   supervisor(1, P), nhigher(P, X)
    supervisor(1, 5), nhigher(5, X)
     nhigher(5, X)
      /* OPTIMIZATION: Choose supervisor(M, N) [5/M] [X/N] first */
      supervisor(M, N) [5/M] [X/N]
```

```
       supervisor(5, X)
        supervisor(5, 34)
         solution 2: X=34
        supervisor(5, 27)
         solution 3: X=27
      nhigher(P, N), supervisor(M, P) [5/M] [X/N]
       nhigher(P, X), supervisor(5, P)
        /* OPTIMIZATION: Choose supervisor(5, P) first */
        supervisor(5, P), nhigher(P, X)
         supervisor(5, 34), nhigher(34, X)
          nhigher(34, X)
           /* OPTIMIZATION: Choose supervisor(M, N) [34/M] [X/N] first */
           supervisor(M, N) [34/M] [X/N]
            supervisor(34, X)
             fail
           nhigher(P, N), supervisor(M, P) [34/M] [X/N]
            nhigher(P, X), supervisor(34, P)
             /* OPTIMIZATION: Choose supervisor(34, P) first */
             supervisor(34, P), nhigher(P, X)
              fail
         supervisor(5, 27), nhigher(34, X)
          /* Similar to the case of supervisor(5, 34), nhigher(34, X) */
```

This query is not just faster than the Prolog version, but infinitely faster than the Prolog version, in that it terminates, resolving to `X=5`, `X=34`, and `X=28`.

17 Everything In Between

Most implementations of relational databases actually implement extensions to SQL which render them Turing-complete, but typically not the Prolog cut. In some cases, this is simply by cheating and allowing a Turing-complete language to be embedded into SQL, but in other cases, it is by allowing less restrictive databases, for instance by removing the stratification requirements of numbers or lists, or allowing unbounded variables in the heads of Horn clauses.

The cut operator is rarely reintroduced, because the goal of such a system is usually a Turing-complete database in which query optimization is still allowed. Modern database systems have to contend with Turing-complete programs over which optimization can turn a non-halting computation into a halting computation! If this concept seems fascinating, however, you'll have to study it in CS448, not here.

Datalog has also found some use in machine learning, where certain ML structures, such as neural networks, fit within the constraints of Datalog [6].

18 Fin

In the next module, we will look at imperative programming languages, which account for most languages used in practice. Assignment 4 will focus on implementing query programming like you've seen in this module.

References

[1] Roman Barták. On-line guide to Prolog programming, 1998. http://kti.mff.cuni.cz/~bartak/prolog/.

[2] Edgar F Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[3] Jacques Cohen. Non-deterministic algorithms. *ACM Computing Surveys (CSUR)*, 11(2):79–94, 1979.

[4] Jacques Cohen. Describing Prolog by its interpretation and compilation. *Communications of the ACM*, 28(12):1311–1324, 1985.

[5] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. *Datalog and recursive query processing*. Now Publishers, 2013.

[6] Hongyuan Mei, Guanghui Qin, Minjie Xu, and Jason Eisner. Neural datalog through time: Informed temporal modeling via logical specification. In *International Conference on Machine Learning*, pages 6808–6819. PMLR, 2020.

Rights