10 Prolog: Final Thoughts

Chapter Objectives

Prolog and declarative representations
- Facts
- Rules

The `append` example

Prolog referenced to automated reasoning systems
- Lack of *occurs check*
- No *unique names* or *closed world*

Prolog semantics
- Pattern-matching

Left-to-right depth-first search search

Backtracking on variable bindings

References offered for Prolog extensions

Chapter Contents

10.1 Prolog: Towards a Declarative Semantics

We have now finishing our nine-chapter presentation of Prolog. To summarize and conclude we describe again the design philosophy supporting this language paradigm, look at how this influenced the history of its development, summarize the main language idioms we used in building our AI applications programs, and mention several modern extensions of this declarative approach to programming.

Prolog was first designed and used at the University of Marseilles in the south of France in the early 1970s. The first Prolog interpreter was intended to analyze French using *metamorphosis grammars* (Colmerauer 1975). From Marseilles, the language development moved on to the University of Edinburgh in Scotland, where at the Artificial Intelligence Department, Fernando Pereira and David Warren (1980) created *definite clause grammars*. In fact, because of the declarative nature of Prolog and the flexibility of pattern-driven control, tasks in Natural Language Processing, NLP, (Luger 2009, Chapter 15) have always offered a major application domain (see Chapters 8 and 9). Veronica Dahl (1977), Dahl and McCord (1983), Michael McCord (1982, 1986), and John Sowa (Sowa 1984, Walker et al. 1987) have all contributed to this research.

Besides NLP, Prolog has supported many research tasks including the development of early expert systems (Bundy et al. 1979). Building AI representations such as semantic nets, frames, and objects has always been an important task for Prolog (see especially *Knowledge Systems and Prolog* by

Adrian Walker, Michael McCord, John Sowa, and Walter Wilson, 1987, and *Prolog: A Relational Language and Its Applications* by John Malpas 1987).

In the remainder of this chapter we discuss briefly declarative programming, how Prolog relates to theorem proving, and describe again the Prolog idioms presented in Part II.

In traditional computing languages such as FORTRAN, C, and Java the logic for the problem's specification and the control for executing the solution algorithm are inextricably mixed together. A program in these languages is simply a sequence of things *to be done* to achieve an answer. This is the accepted notion of *applicative* or *procedural* languages. Prolog, however, separates the logic or specification for a problem application from the execution or control of the use of that specification. In artificial intelligence programs, there are many reasons for this separation, as has been evident throughout Part II.

Prolog presents an alternative approach to computing. A program, as we have seen, consists of a set of specifications or declarations *of what is true* in a problem domain. The Prolog interpreter, taking a question from the user, determines whether it is true or false with respect to the set of specifications, If the query is true, Prolog will return a set of variable bindings (a model, see 10.2) under which the query is true.

As an example of the *declarative/nonprocedural* nature of Prolog, consider `append`:

```
append([ ], L, L).
append([X | T], L, [X | NL]) :- append(T, L, NL).
```

`append` is nonprocedural in that it defines a relationship between lists rather than a series of operations for joining two lists. Consequently, different queries will cause it to compute different aspects of this relationship. We can understand `append` by tracing its execution in joining two lists together. If the following call is made, the response is:

```
?- append([a, b, c], [d, e], Y).
Y = [a, b, c, d, e]
```

The execution of append is not tail recursive, in that the local variable values are accessed after the recursive call has succeeded. In this case, X is placed on the head of the list ([X | NL]) after the recursive call has finished. This requires that a record of each call be kept on the Prolog stack. For purposes of reference in the following trace:

```
1. is append([ ], L, L).
2. is append([X | T], L, [X | NL]) :-
          append(T, L, NL).

?- append([a, b, c], [d, e], Y).
   try match 1, fail [a, b, c] /= [ ]
   match 2, X is a, T is [b, c], L is [d, e],
        call append([b, c], [d, e], NL)
        try match 1, fail [b, c] /= [ ]
        match 2, X is b, T is [c], L is [d, e],
             call append([c], [d, e], NL)
```

```
            try match 1, fail [c]  [ ]
            match 2, X is c, T is [ ], L is [d, e],
                 call append([ ], [d, e], NL)
                match 1, L is [d, e]
                yes
            yes, N is [d, e], [X | NL] is [c, d, e]
   yes, NL is [c, d, e], [X | NL] is [b, c, d, e]
yes, NL is [b, c, d, e],
      [X | NL] is [a, b, c, d, e]
Y = [a, b, c, d, e], yes
```

In most Prolog programs, the parameters of the predicates seem to be intended as either "input" or "output"; most definitions assume that certain parameters be bound in the call and others unbound. This need not be so. In fact, there is no commitment at all to parameters being input or output! Prolog code is simply a set of specifications of what is true, a statement of the logic of the situation. Thus, `append` specifies a relationship between three lists, such that the third list is the catenation of the first onto the front of the second.

To demonstrate this we can give `append` a different set of goals:

```
?- append([a, b], [c], [a, b, c]).
Yes
?- append([a], [c], [a, b, c]).
No
?- append(X, [b, c], [a, b, c]).
X = [a]
?- append(X, Y, [a, b, c]).
X = [ ]
Y = [a, b, c]
;
X = [a]
Y = [b, c]
;
X = [a, b]
Y = [c]
;
X = [a, b, c]
Y = [ ]
;
no
```

In the last query, Prolog returns all the lists `X` and `Y` that, when appended together, give `[a,b,c]`, four pairs of lists in all. As mentioned above, `append` gives a statement of the logic of a relationship that exists among three lists. What the interpreter produces depends on the query.

The notion of solving a problem based on a set of specifications for relationships in a problem domain area, coupled with the action of a theorem prover, is exciting and important. As seen in Part II, it is a

valuable tool in areas as diverse as natural language understanding, databases, expert systems, and machine learning. How the Prolog interpreter works cannot be fully understood without the concepts of resolution theorem proving, especially the Horn clause refutation process, which is presented in Luger (2009, Section 14.2 and Section 14.3) where Prolog is presented as an instance of a resolution refutation system. In Section 10.2 we briefly summarize these issues.

10.2 Prolog and Automated Reasoning

Prolog's declarative semantics, with the interpreter determining the truth or falsity of queries has much of the feel of an automated reasoning system or *theorem prover* (Luger 2009, Chapter 14). In fact, Prolog is not a theorem prover, as it lacks several important features that would make it both *sound* (only producing mathematically correct responses) and *complete* (able to produce all correct responses consistent with a problem's specifications). Many of these features are not implemented in current versions of Prolog. In fact, most are omitted to make Prolog a more efficient programming tool, even when this omission costs Prolog any claim of mathematical soundness.

In this section we will list several of the key features of automated reasoning systems that Prolog lacks. First is the *occurs check*. Prolog does not determine whether any expression in the language contains a subset of itself. For example, the test whether `foo(X) = foo(foo(X))` will make most Prolog environments get seriously weird. It turns out that the systematic check of whether any Prolog expression contains a subset of itself is computationally costly and as a result is ignored.

A second limitation on Prolog is the order constraint. The Prolog inference system (interpreter) performs a left-to-right depth-first goal reduction on its specifications. This requires that the programmer order these specifications appropriately. For example, the termination of a recursive call must be presented before the recursive expression, otherwise the recursion will never terminate. The programmer can also organize goals in the order in which she wishes the interpreter to see them. This can help create an efficient program but does not support a truly declarative specification language where non-deterministic goal reduction is a critical component. Finally, the use of the cut, `!,` allows the programmer to further limit the set of models that the interpreter can compute. Again this limitation promotes efficiency but it is at the cost of a mathematically complete system.

A third limitation of Prolog is that there is no *unique name* constraint or *closed world* assumption. Unique names means that each atom in the prolog world must have one and only one "name" or value; otherwise there must exist a set of deterministic predicates that can reduce an atom to its unique (canonical) form. In mathematics, for example, 1, cannot be 1 + 0, 0 + 1, or 0 + 1 + 0, etc. There must be some predicate that can reduce all of these expressions to one canonical form for that atom.

Further, the closed world assumption, requires that all the atoms in a

domain must be specified; the interpreter cannot return `no` because some atom was ignored or misspelled. These requirements in a theorem proving environment address the *negation as failure* result that can be so frustrating to a Prolog programmer. Negation as failure describes the situation where the interpreter returns `no` and this indicates either that the query is false or that the program's specifications are incorrect. When a true theorem prover responds `no` then the query is false.

Even though the Prolog interpreter is not a theorem prover, the intelligent programmer can utilize many aspects of its declarative semantics to build a set of clean representational specifications; these are then addressed by an efficient interpreter. For more discussion of Prolog as theorem proving see Luger (2009, Section 14.3).

10.3 Prolog Idioms and Extensions

We now summarize several of the Prolog programming idioms we have used in Part II of this presentation. We consider idioms from three perspectives, from the lowest level of command and specification instructions, from a middle level of creating program language modules such as abstract data types, and from the most general level of using meta-predicates to make new interpreters within Prolog that are able to operate on specific sets of Prolog expressions.

From the lowest level of designing control and building specifications, we mention four different idioms that were used throughout Part II as critical components for constructing Prolog programs. The first idiom is *unification* or *pattern matching*. Unification offers a unique power for variable binding found only in high-level languages. It is particularly powerful for pattern matching across sets of predicate calculus specifications. Unification offers an efficient implementation of the `if/then/else` constructs of lower level languages: if the pattern matches, perform the associated action, otherwise consider the next pattern. It is also an important and simplifying tool for designing meta-interpreters, such as the production system (Section 4.2). Production rules can be ordered and presented as a set of patterns to be matched by unification) that will then trigger specific behaviors. An algorithm for unification can be found in Luger (2009, Section 2.3). It is interesting to note that unification, a constituent of Prolog, is explicitly programmed into Lisp (Chapter 15) and Java (Chapter 32 and 33) to support AI programming techniques in these languages.

A second idiom of Prolog is the use of *assignment*. Assignment is related to unification in that many variables, especially those in predicate calculus form, are assigned values through unification. However, when a variable is to have some value based on an explicit functional calculation, the `is` operator must be used. Understanding the specific roles of assignment, evaluation, and pattern matching is important for Prolog use.

The primary control construct for Prolog is *recursion*, the third idiom we mention. Recursion works with unification to evaluate patterns in much the same way as `for`, `repeat/until`, or `while` constructs are used in lower level languages. Since many of AIs problem solving tasks consist in

searching indeterminate sized trees or graphs, the naturalness of recursion makes it an important idiom: until specific criteria are met continue search over specifications. Of course the lower-level control constructs of `for`, `repeat`, etc., could be built into Prolog, but the idioms for these constructs is recursion coupled with unification.

Finally, at the predicate creation level of the program, the *ordering* of predicate specifications is important for Prolog. The issue is to utilize the built in depth-first left-to-right goal reduction of the Prolog interpreter. Understanding the action of the interpreter has important implications for using the order idiom. Along with order of specifications for efficient search, of course, is understanding and using wisely the predicate cut, !.

At the middle level of program design, where specifications are clustered to have systematic program level effects, we mention several idioms. These were grouped together in our presentation in Section 3.3 under the construct *abstract data types (ADTs)*. Abstract data types, such as *set*, *stack*, *queue*, and *priority queue* were developed in Chapter 3. The abstractions allow the program designer to use the higher-level constructs of queue, stack, etc. directly in problem solving. We then used these control abstract data types to design the search algorithms presented in Chapter 4. They were also later used in the machine learning and natural language chapters of Part II. For our Prolog chapters these idioms offer a natural way to express constructs related to graph search.

Finally, at that abstract level where the programmer is directly designing interpreters we described and used the *meta-predicate* idioms. Meta-predicates are built into the Prolog environment to assist the program designer with tools that manipulate other Prolog predicates, as described in Section 5.1. We demonstrated in Section 5.2 how the meta-predicate idioms can be used to enforce type constraints within a Prolog programming environment.

The most important use of meta-predicates, however, is to design meta-interpreters as we did in the remaining chapters (6 – 9) of Part II. Our meta-interpreters were collected sets of predicates that were used to interpret other sets of predicate specifications. Example meta-interpreters included a Prolog interpreter written in Prolog and a production system interpreter, Exshell, for building rule-based expert systems. The meta-interpreter is the most powerful use of our idioms, because at this level of abstraction and encapsulation our interpreters are implementing specific design patterns.

There are many additional software tools for declarative and logic programming available. An extension of Prolog's declarative semantics into a true resolution-based theorem-proving environment can be found in Otter (McCune and Wos 1997). Otter, originally produced at Argonne National Laboratories, is a complete automated reasoning system based on resolution refutation that addresses many of the shortcomings of Prolog mentioned in Section 10.2, e.g., the occurs check. A current version of Otter includes Isabelle, written in ML, (Paulson 1989), Tau (Halcomb and Schulz 2005), and Vampire (Robinson and Voronkov 2001). These automated reasoning systems are in the public domain and downloadable.

Ciao Prolog is a modern version of Prolog created in Spain (Mera et al. 2007, Hermenegildo et al. 2007). Ciao offers a complete Prolog system, but its novel modular design allows both restricting and extending the language. As a result, it allows working with fully declarative subsets of Prolog and also to extend these subsets both syntactically and semantically. Most importantly, these restrictions and extensions can be activated separately on each program module so that several extensions can coexist in the same application for different modules. Ciao also supports (through such extensions) programming with functions, higher-order (with predicate abstractions), constraints, and objects, as well as feature terms (records), persistence, several control rules (breadth-first search, iterative deepening), concurrency (threads/engines), a good base for distributed execution (agents), and parallel execution. Libraries also support WWW programming, sockets, external interfaces (C, Java, TclTk, relational databases, etc.).

Ehud Shapiro and his colleagues have researched the parallel execution of Prolog specifications. This is an important extension of the power to be gained by extending the built in depth-first search with backtracking traditional Prolog interpreter with parallel execution. For example, if a declarative goal has a number of or based goals to satisfy, these can be checked in parallel (Shapiro 1987).

Constraint logic programming is a declarative specification language where relations between variables can be stated in the form of constraints. Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties or requirements of the solution to be found. The constraints used in constraint programming are of various kinds, including constraint satisfaction problems. Constraints are often embedded within a programming language or provided via separate software libraries (O'Sullivan 2003, Krzysztof and Wallace 2007).

Recent research has also extended traditional logic programming by adding distributions to declarative specifications (Pless and Luger 2003, Chakrabarti et al. 2005, Sakhanenko et al. 2007). This is a natural extension, in that declarative specifications do not be need to be seen as deterministic, but may be more realistically cast as probabilistic.

There is ongoing interest in logic-based or pure declarative programming environments other than Prolog. *The Gödel Programming Language*, by Hill and Lloyd (1994), presents the Gödel language and Somogyi, Henderson, and Conway (1995) describe Mercury. Gödel and Mercury are two relatively new declarative logic-programming environments.

Finally, Prolog is a general-purpose language, and, because of space limitations, we have been unable to present a number of its important features and control constructs. We recommend that the interested reader pursue some of the many excellent texts available including *Programming in Prolog* (Clocksin and Mellish 2003), *Computing with Logic* (Maier and Warren 1988), *The Art of Prolog* (Sterling and Shapiro 1986), *The Craft of Prolog* (O'Keefe 1990), *Techniques of Prolog Programming* (VanLe 1993), *Mastering Prolog* (Lucas 1996), or *Advanced Prolog: Techniques and Examples* (Ross 1989), *Knowledge Systems through Prolog* (King 1991), and *Natural Language Processing in Prolog* (Gazdar and Mellish 1989).

In Part III we present the philosophy and idioms of functional programming, using the Lisp language. Part IV then presents object-oriented design and programming with Java, and Part V offers our summary. As the reader covers the different parts of this book it can be seen how the different languages are utilized to address many of the same problems, while the idioms of one programming paradigm may or may not be suitable to another.