goal "$X \mathbf{is} 4+3$" succeeds with $X$ bound to 7. On the other hand, the goal "$5 \mathbf{is} X+Y$" fails, because the built-in functions do not do arbitrary equation solving.

- There are built-in predicates that have side effects when executed. These include input-output predicates and the as_sert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The occur check is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

Prolog’s design represents a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed.

### 9.4.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-B.C.ASK algorithm from Figure 9.6, with the program as the knowledge base. We say “essentially” because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, our implementation had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure, a stack of choice points, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

Second, our simple implementation of FOL-B.C.ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding. At any point in time, every variable in the program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the trail. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to find clauses that match the goal. A compiled Prolog

Note that if the Peano axioms are provided, such goals can be solved by inference within a Prolog program.
procedure \texttt{APPEND}(aT, y, z, \texttt{continuation})

\textsc{trait} \textsc{GLOBAL-TRAIL-POINTER()}

\textbf{if} aT = [] \textbf{and} \textsc{UNIFY}(y, aT) \textbf{then} \textsc{CALL(continuation)}

\textsc{RESET-TRAIL(trail)}
a, x, z \textsc{NEW-VARIABLE()}, \textsc{NEW-VARIABLE()}, \textsc{NEW-VARIABLE()}

\textbf{if} \textsc{UNIFY(aT, [a \ y])} \textbf{and} \textsc{UNIFY(aT, [a \ a])} \textbf{then} \textsc{APPEND}(x, y, z, \texttt{continuation})

Figure 9.8 Pseudocode representing the result of compiling the Append predicate. The function \textsc{NEW-VARIABLE} returns a new variable, distinct from all other variables used so far. The procedure \texttt{CALL(continuation)} continues execution with the specified continuation.

Program, on the other hand, is an inference procedure for a specific set of clauses, so it \textit{knows} what clauses match the goal. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to open-code the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren \textit{et al.} (1977).)

The instruction sets of today’s computers give a poor match with Prolog’s semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language. The most popular intermediate language is the Warren Abstract Machine, or \textsc{WAM}, named after David H. D. Warren, one of the implementers of the first Prolog compiler. The \textsc{WAM} is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language. Other compilers translate Prolog into a high-level language such as \textsc{Lisp} or \textsc{C} and then use that language’s compiler to translate to machine language. For example, the definition of the Append predicate can be compiled into the code shown in Figure 9.8. Several points are worth mentioning:

\begin{itemize}
  \item Rather than having to search the knowledge base for Append clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.
  \item As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by \textsc{RESET-TRAIL} if the first clause fails. This will undo any bindings generated by the first call to \textsc{UNIFY}.
  \item The trickiest part is the use of \textit{continuations to implement} choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like \texttt{APPEND} when the goal succeeds, because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the \texttt{APPEND} code, if the first argument is empty and the second argument unifies with the third, then the \texttt{APPEND} predicate has succeeded. We then \texttt{CALL} the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to \texttt{APPEND} were at the top level, the continuation would print the bindings of the variables.
\end{itemize}