function \textsc{FOL-BC-ASK}(KB, \text{query})\,\text{returns a generator of substitutions}
\begin{align*}
&\text{return } \textsc{FOL-BC-OR}(KB, \text{query}, 0)\end{align*}

\text{generator } \textsc{FOL-BC-OR}(KB, \text{goal}, 0)\text{ yields a substitution}
\begin{align*}
&\text{for each rule } (lhs, rhs) \text{ in } \textsc{Fetch-Rules-For-Goal}(KB, \text{goal}) \text{ do}
\end{align*}
\begin{align*}
&(lhs, rhs) \leftarrow \textsc{Standardize}\text{Variables}(lhs, rhs)
\end{align*}
\begin{align*}
&\text{for each } 0' \text{ in } \textsc{FOL-BC-AND}(KB, \text{the, UNIFY}(rhs, \text{goal}, 0)) \text{ do}
\end{align*}
\begin{align*}
&\text{yield } 9'
\end{align*}

\text{generator } \textsc{FOL-BC-AND}(KB, \text{goals}, 9)\text{ yields a substitution}
\begin{align*}
&\text{if } O = \text{failure} \text{ then return}
\end{align*}
\begin{align*}
&\text{else if } \text{LENGTH} (\text{goals}) = 0 \text{ then yield } 0
\end{align*}
\begin{align*}
&\text{else do}
\end{align*}
\begin{align*}
&\text{first, rest } \textsc{First}(\text{goals}), \textsc{Rest}(\text{goals})
\end{align*}
\begin{align*}
&\text{for each } 0' \text{ in } \textsc{FOL-BC-OR}(KB, \text{SUBST}(\emptyset, \text{first}) , 0) \text{ do}
\end{align*}
\begin{align*}
&\text{for each } 0'' \text{ in } \textsc{FOL-BC-AND}(KB, \text{rest, } 0') \text{ do}
\end{align*}
\begin{align*}
&\text{yield } 8''
\end{align*}

Figure 9.6 A simple backward-chaining algorithm for first order knowledge bases.

Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove \texttt{Criminal(West)}, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time \textsc{FOL-BC-ASK} gets to the last conjunct, originally \texttt{Hostile(\text{z})}, \texttt{z} is already bound to \texttt{Nano}. 

9.4.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski’s equation,

\[
\text{Algorithm} = \text{Logic} \pm \text{Control},
\]

Prolog is the most widely used logic programming language. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to; instead of \( A \land B = C \) in Prolog we have \( C : - A, B \).

For example:

\[
\text{criminal}(X), \text{american}(X), \text{weapon}(Y), \text{sells}(X,Y,Z), \text{hostile}(Z).
\]

The notation \([E \mid L]\) denotes a list whose first element is \( E \) and whose rest is \( L \). Here is a typical example:

\[
\text{criminal}(X), \text{american}(X), \text{weapon}(Y), \text{sells}(X,Y,Z), \text{hostile}(Z).
\]

The notation \([E \mid L]\) denotes a list whose first element is \( E \) and whose rest is \( L \). Here is a Prolog program for \( \text{append}(X,Y,Z) \), which succeeds if list \( Z \) is the result of appending lists \( X \) and \( Y \):

\[
\text{append}([1,Y,Y]), \text{append}([A|X],Y,[A|Z]) \quad \text{append}(X,Y,Z).
\]

In English, we can read these clauses as (1) appending an empty list with a list \( Y \) produces the same list \( Y \) and (2) \( [A \mid Z] \) is the result of appending \( [A \mid X] \) onto \( Y \), provided that \( Z \) is the result of appending \( X \) onto \( Y \). In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a relation that holds among three arguments, rather than a function computed from two arguments. For example, we can ask the query \( \text{append}(X,Y,[1,2]) \): what two lists can be appended to give \( [1,2] \)? We get back the solutions

\[
X=[1] \quad Y=[1,2];
X=[1] \quad Y=[2];
X=[1,2] \quad Y=[1]
\]

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.5).
- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the