Let's Write Some List Predicates

1. member(X, List).
2. append(List1, List2, Result).
3. swapFirstTwo(List1, List2).
4. length(List).

List Membership

Definition of member...

\[ \text{member}(X, [a, b]) \]
\[ \text{member}(X, [a, c]) \]
\[ \text{member}(X, [a, b, c]) \]

Trace of Member

[trace] \text{member}(c, [a, b, c, d]).
  Call (5) \text{member}(c, [a, b, c, d]) \rightarrow \text{creep}
  Call (6) \text{member}(d, [a, b, c, d]) \rightarrow \text{creep}
  \quad \text{Answer: No}
  \text{exit (1) \text{member}(c, [a, b, c, d]) \rightarrow \text{creep}}
  \text{exit (1) \text{member}(d, [a, b, c, d]) \rightarrow \text{creep}}
  \text{exit (1) \text{member}(c, [a, b, c, d]) \rightarrow \text{creep}}
  \text{exit (1) \text{member}(d, [a, b, c, d]) \rightarrow \text{creep}}

Append (cont.)

\[ \text{append}(X, Y, Z) \]
\[ Z = [0, 1, 2] \]
\[ X = [3, 4] \]
\[ Y = [5] \]

Generate lists:
\[ \text{append}(X, Y, Z) \]
\[ X = [3, 4] \]
\[ Z = [0, 1, 2] \]
\[ X = [3, 4, 5] \]

Trace:
[trace] \text{append}(X, Y, Z).
  Call (7) \text{append}(X, Y, Z) \rightarrow \text{creep}
  Call (8) \text{append}(X, Y, Z) \rightarrow \text{creep}
  Call (9) \text{append}(X, Y, Z) \rightarrow \text{creep}
  \quad \text{Answer: Yes}
  \text{exit (1) \text{append}(X, Y, Z) \rightarrow \text{creep}}

Computing the Length of a List

Definition of length...

\[ \text{length}([a, b, c], L) \]
\[ L = 3 \]

Trace of Length:

[trace] \text{length}([a, b, c], L).
  Call (1) \text{length}([a, b, c], 0) \rightarrow \text{creep}
  Call (2) \text{length}([a, b, c], 1) \rightarrow \text{creep}
  Call (3) \text{length}([a, b, c], 2) \rightarrow \text{creep}
  Call (4) \text{length}([a, b, c], 3) \rightarrow \text{creep}
  \quad \text{Answer: 3}
  \text{exit (1) \text{length}([a, b, c], L) \rightarrow \text{creep}}

Trace of Length (cont)

But this does work
\[ n \geq 1 \]
\[ n^2 + n \leq n \times (n+1) \]

Trace of Length (cont)

[trace] \text{length}([a, b, c, d], X).
  Call (1) \text{length}([a, b, c, d], 0) \rightarrow \text{creep}
  Call (2) \text{length}([a, b, c, d], 1) \rightarrow \text{creep}
  Call (3) \text{length}([a, b, c, d], 2) \rightarrow \text{creep}
  Call (4) \text{length}([a, b, c, d], 3) \rightarrow \text{creep}
  \quad \text{Answer: 4}
  \text{exit (1) \text{length}([a, b, c, d], X) \rightarrow \text{creep}}

Try some other traces
Accessing More Than One Initial Element

Definition of `sum_first`...

\[ \text{sum_first}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \land \text{Y} \neq \text{nil} \land \text{Z} \neq \text{nil} \\
\text{sum_first}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

1. `sum_first((\text{X}, \text{Y}), \text{Z})`.
2. `sum_first((\text{X}, \text{nil}), \text{Y})`.
3. `sum_first((\text{X}, \text{Y}), \text{nil})`.
4. `sum_first((\text{X}, \text{Y}), \text{Z})`.

Lists of a Specified Length

Definition of `list_of_length`...

\[ \text{list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

Lists of a Specified Length

New definition of `list_of_length`...

\[ \text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

\[ \text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) = \begin{cases} \text{true} & \text{if X} \neq \text{nil} \\
\text{working_list_of_length}(\text{X}, \text{Y}, \text{Z}) & \text{false} \end{cases} \]

Arithmetic in Prolog

What is the result of these queries:

1. `X = 3 * Y, Y = 7, X = 14`.
2. `X = Y + 3, Y = 10, X = 13`.

To get an expression evaluated, use `X is expression` where `expression` is an arithmetic expression, and `X` is fully instantiated.

Examples:

1. `X is 10 * 7`.
2. `Y is 7, Z is 3 * Y, Y = 21`.

Let's Write Some Predicates with Arithmetic

1. `factorial(N, Ans)`.
2. `sum_list(List, Total)`.

Factorial

\[ \text{factorial}(0, 1) \]

\[ \text{factorial}(X, Y) = \begin{cases} 1 & \text{if } X = 0 \\
Y * \text{factorial}(X - 1) & \text{if } X > 0 \end{cases} \]

What are the preconditions for `factorial`?

Factorial with an Accumulator

\[ \text{factorial2}(N, X, Y) \]

\[ \text{factorial2}(N, X, Y) = \begin{cases} 1 & \text{if } X = 0 \\
Y + \text{factorial2}(N, X - 1, Y) & \text{if } X > 0 \end{cases} \]

What are the preconditions?

Beyond Horn Logic

So far, we have studied what is known as pure logic programming, in which all the rules are Horn.

For some applications, however, we need to go beyond this.

For instance, we often need

- Arithmetic
- Negation

Fortunately, these can easily be accommo-
dated by simple extensions to the logic-
programming framework.

Trace of Factorial

\[ \text{factorial}(3, 1) \]

\[ \text{factorial}(X, Y) = \begin{cases} 1 & \text{if } X = 0 \\
Y * \text{factorial}(X - 1) & \text{if } X > 0 \end{cases} \]

\[ \text{factorial}(X, Y) = \begin{cases} 1 & \text{if } X = 0 \\
Y * \text{factorial}(X - 1) & \text{if } X > 0 \end{cases} \]

\[ \text{factorial}(X, Y) = \begin{cases} 1 & \text{if } X = 0 \\
Y * \text{factorial}(X - 1) & \text{if } X > 0 \end{cases} \]
Trace of Factorial w/ an Accumulator

Sum of List

Arithmetic Predicates may not be Invertible

Negation as Failure

Assuming that something unprovable is false is called negation as failure.

(Based on a closed world assumption.)

The goal \( \neg G \) succeeds whenever the goal \( G \) fails.

\( \neg \) member(b, [a, b, c]),
Yes
\( \neg \) member(b, [a, b, c]),
No
\( \neg \) member(b, [a, c]),

Example: Disjoint Sets

Example: Disjoint Sets (cont.)

Proper use of Negation as Failure

\( \neg \) works properly only in the following cases:

1. When \( G \) is fully instantiated at the time Prolog processes the goal \( \neg \neg G \).
   (In this case, \( \neg \neg G \) is interpreted to mean "the goal \( G \) does not succeed."

2. When all variables in \( G \) are unique to \( G \), i.e., they don't appear elsewhere in the same clause.
   (In this case, \( \neg \neg G(X) \) is interpreted to mean "There is no value of \( X \) that will make \( G(X) \) succeed."
Safety

Consider the following rule:

(*) hates(tom,x) := not loves(tom,x).

This may NOT be what we want, for several reasons:

• The answer is infinite, since for any person p not mentioned in the database, we cannot infer loves(tom,p), so we must infer hates(tom,p).

Rule (*) is therefore said to be unsafe.

• The rule does not require X to be a person, e.g., since we cannot infer

  loves(tom, hammer)
  loves(tom, verme)
  loves(tom, green)
  loves(tom, abc)

We must infer that tom hates all these things.

Safety (Cont’d)

To avoid these problems, rules with negation should be guarded:

hates(tom,x) := vegetable(z), green(x), not loves(tom,x).

i.e., tom hates every green vegetable that he does not love.

Here, vegetable and green are called guard literals. They guard against safety problems by binding X to specific values in the database.

Data Structure: the “Structure”

Representing a parse tree

Simple grammar:

<op> := <cmd> <adjlist> <op>
<adjlist> := { adj }
<cmd> := the | a
adj := child | dog | professor
adj := small | friendly | noisy

Parse tree:

Representation as a Prolog structure: