DATR Paths as Arguments*

Lionel Moser
School of Cognitive & Computing Sciences
University of Sussex
Brighton, U.K.

August 15, 1992

Abstract

DATR is a lexical knowledge representation language

Contents

composition. The appendix contains a variety of DATR theories illustrating 'applications' of these techniques.

2 Nodes as function definitions; Paths as arguments

DATR nodes implement functions of a single argument, the path. We view the node as a *function* and the path as a sequence of symbols, called the *argument* or *argument list*.[5,6]

A path is a sequence of path *constituents* which are elements of a set *FEAT*. The extension of a path at a given node is a *value*, which is itself a sequence over a set *ATOM*. A DATR theory is a finite sequence of definitional statements, which implicitly define the sets *FEAT* and *ATOM*.[7] Because the theory is finite, both of these sets are also finite.[8] Intuitively, atoms are terminal symbols which appear as part of a value on the RHS of `==` in definitional statements. Path constituents appear inside path descriptors (angle brackets `<>`), but DATR allows values to be substituted directly into path descriptors (in an evaluable path), thus any atom may also be a member of *FEAT* even if it does not appear directly inside a path descriptor. It follows that $ATOM \subseteq FEAT$.

In viewing a path as an argument, and an argument as a sequence of symbols, we needn't require that every atom be a possible such symbol;

```
Pv0: <a> = a
     <b> = b
     <c> = c
     <d> = d
     <d a b c> = d.
```

Pv0 can be stated as a generalisation over some set of atoms, which we define as a DATR variable.[9] We define the variable $terminal and write Pv0 as follows:

```
#vars $terminal: a b c d.

Pv0: <$terminal> = $terminal.
```

The set of theorems remains the same, as the variable notation is merely a shorthand for the original definition.

Pv0 maps any sequence of symbols over $terminal into the value of the first one, which is fine for truncation. In order to convert a path sequence into a list of values, we give a recursive definition, in Pv1:[10]

```
Pv1:<> == ()
    <$terminal> == $terminal <>.
```

The recursive definition of Pv1 is: the value of the empty path is the empty list; and the value of any other path is a list whose first symbol is the value of the first symbol of the path, appended with the value of Pv1 of the rest of the path. This lets us derive the following theorems:

```
Pv1: <a> = a
     <b> = b
     <a b a c d> = a b a c d.
```

2.2 The path as a single argument

A path interpreted as an argument has the format $< x_1\ x_2\ \ldots\ x_n\ ;>$ where
x_i is a symbol over the alphabet $terminal; and
; is the argument terminator.
The argument is a list structure, and the terminator we have introduced is neither part of the argument nor a member of $terminal (the domain of the theory). Under the path-as-argument interpretation, the argument must be terminated, permitting a distinction to be drawn between a path representing an argument which is a nil list (<;>), and the empty path (<>), for which we do not define a semantics – we do not consider the empty path to be an 'argument'.[11]

The primitive operations on lists are familiar: adding to the list at the front and the end, and splitting it into its first element and the rest (*i.e.*, its head and tail). The first operation is trivial: in an evaluable path, symbols placed before an argument's value construct a new argument with these symbols at the front, while any symbols placed after its value are added at the end. Suppose that node A is to pass node B its argument with atom x appended to the end. First we define a path-to-value conversion which strips the terminator. We call this Pv_to_;:

[9] A DATR variable is not a 'variable' which takes on different values. Rather, it is a shorthand for listing a fixed set of values – a macro substitution. (Jenkins 1990)

[10] In this paper we are using a notation for lists differing slightly from that used in current definitions of DATR: we omit parentheses around non-nil lists. We do, however, use () to represent a nil list.

[11] This is analogous to the distinction in some programming languages between f() and f([]) – these being a function call with no arguments, and one argument (a nil list), respectively.

```
Pv_to_;:
   <;> == ()
   <$terminal> == $terminal <>
   <> == 'invalid symbol'.
```

To create a new argument with `x` appended we append `x` to the value of the original argument:

```
A1: <> == B:<Pv_to_; x ;>.
```

Similarly, to create an argument list with `x` inserted at the front of the original argument, we place it *before* the value of the list:

```
A2: <> == B:<x Pv_to_; ;>.
```

Of course inserting at the front of the argument could have been done directly, simply by defining `A: <> == B:<x>.`[12]

The last two extraction operations also require explicit path-to-value conversion on the appropriate atom or atoms. `First` returns the first atom in the argument; a nil list does not have a first element:

```
First:
   <$terminal> == $terminal
   <;> == 'nil list'
   <> == 'missing terminator'.
```

The tail of the argument we define as path-to-value conversion of all symbols following the first, if there is one, and nil if the list is nil.[13] If the list is not nil, the result is computed as `Pv_to_;` on list with the first element removed:

```
Rest:
   <;> == ()
   <$terminal> == Pv_to_;:<>
   <> == 'missing terminator'.
```

Some typical theorems of `First` and `Rest` are:

```
First: <a b c d ;> = a.

Rest: <;> = ()
      <a ;> = ()
      <a b ;> = b
      <a b c d ;> = b c d.
```

`First` and `Rest` can be applied to the results of each other's (or their own) evaluation – indeed, this is the primary reason for defining them as we have. For example, the third element of an argument (assuming there is one), and the tail of the tail could be extracted by `Third` and `Rest2`, respectively, with the following definitions:

```
Third: <> == First:<Rest:<Rest ;> ;>.
Rest2:<> == Rest:<Rest ;>.
```

[12] However when we interpret the path as a *list* of arguments, explicit recreation will become necessary.

[13] The semantics of `Rest` given here are probably in need of slight revision – as a matter of consistency, if `First`

yielding the following theorems:[14]

```
Third: <a b c d a a ;> = c.
Rest2: <a b c d a a ;> =
Pop_arg:
   <;> == Pv:<>
   <$terminal> == <>
   <> == 'invalid symbol'.
```

A typical theorem is `Pop_arg:<a b c ; b b ; d d ;> = b b ; d d ;`. We combine these two primitives to define `Arg2`, which evaluates to the second argument in an argument list (which may be the nil list):

```
Arg2: <> == Arg1:<Pop_arg>.
```

The derivation of theorem `Arg2:<a b c ; b b ; d d ;> = b b.` is:

There is nothing to prevent `B` from ‘accessing’ the path suffix beginning with `!`; however if `B` uses

CASE statements become more interesting when the selector is a value resulting from some function of the input. In the simple example below, `G` returns a case selector label, and `J` defines its output in terms of the selector provided by `G`:

```
G: <> == green
   <a> == red
   <a b> == blue
   <b> == blue
   <c c> == red.

J: <> == <case G>
   <case red> == Case1:<>
   <case blue> == Case2:<>
   <case green> == Case3:<>.
```

Some theorems of this theory are:

```
J: <> = green
   <a ;> = red a ;
   <a a d ;> = red a a d ;
   <a b d ;> = blue a b d ;
   <c a b ;> = green c a b ;.
```

The derivation of `J:<a b d ;> = blue a b d ;` is:

Initial query	Derived value	Justification
`J:<a b d ;> =`	`J:<case G:<a b d ;> a b d ;>`	`J:<> == <case G>`
`=`	`J:<case blue a b d ;>`	`G:<a b> == blue`
`=`		

K:

4 Applications

4.1 Decimal arithmetic

Having discussed how to interpret and manipulate paths as argument lists, we now give our first example, decimal arithmetic. We begin by defining a look-up table of decimal digit addition for every pair of decimal digits and a carry of 0 or 1. The set of possible carries is $C = \{0, 1\}$, and the set of digits is $D = \{0, 1, 2, \ldots, 9\}$. A complete table would have $C \times D \times D$ entries, but then every triple (c, d_0, d_1) would have a matching entry for (c, d_1, d_0). In order to minimise the size of the table, we store only one of each such pair, and use look-up failure as a flag indicating that the result is to be found by swapping the digits (and looking again). Node `Dadd` implements such a table, where `Dadd:<carry ; digit0 ; digit1 ;> = new_carry ; remainder ;`:

```
Dadd:
        <> == < Arg1 ; Arg3 ; Arg2 ;>     % swap digits and3Arg1
```

Notice that `Digit` and `Carry` do not append an argument list terminator to the result from `Dadd` before passing it to `Arg2`. This is because the result returned by `Dadd` is not simply a sequence of `$terminal`s, but rather an argument list (*i.e.*, the values are `;`-terminated) ready for processing by the argument extractors. `Arg2`

In order to permit X and Y to be input in their natural order, we introduce the primitive

4.2.1 Tree traversal

We define the powerset tree of order n to be the tree having 2^n leaves such that the label of an edge is 0 or 1, the label on a node is its parent's label appended with the label on the edge from its parent to itself, and every non-terminal node having two children, one with each possible edge label. The powerset tree of order 2, with the internal node labels omitted, is as shown below:

```
     __________|__________
     |    0         1    |
_____0_____          _____1_____
|  0   1  |          |  0   1  |
00        01         10        11
```

An argument list to represent a node in the tree is a pair $<$ *distance_to_leaf* ; *node_label* ;$>$, where

distance_to_leaf is a sequence of n `a`'s, where n is the distance to the leaves; and
node_label

The theorem

```
Powerset1:<a a a ; ;> =
      0 0 0 , 0 0 1 , 0 1 0 , 0 1 1 , 1 0 0 , 1 0 1 , 1 1 0 , 1 1 1.
```

enumerates the powerset of order 3, as shown in the diagram above (which is effectively a 'call tree' for the derivation of the theorem). The first recursive call appends edge label `0` thus descending the left branch; the second recursive call appends edge label `1` thus descending the right branch. In each recursive call the first argument of the constructed argument list is *distance_to_leaf* (`Arg1`) chopped, thus decrementing the distance to the leaves.

A shorter version of this algorithm can be written using a minor 'trick': `Pv` does a path-to-value conversion of an argument list, and is oblivious to the presence or absence of argument terminators (which are required by the argument extractors `Arg1`, `Arg2`, etc...). By leaving the last argument unterminated, and terminating instead the argument list, appending to the last argument does not involve reconstructing the argument list explicitly. `Powerset` takes advantage of this fact, and also uses the implicit node notation (where node self-reference does not require the node to be named explicitly).[23] Here the root node is represented by `Powerset:<a a a ;>`, while the same preterminal as above is represented as `Powerset:<a ; 0 0>`. The definition of `Powerset` is:

```
Powerset:
   <;> == Pv:<>
   <a> == <Pv:<> 0 !> , <Pv:<> 1 !>.
```

The initial invocation now requires the final argument to be non-`;`-terminated, which on the initial call is a nil list, anyway. Theorems now look like this:

```
Powerset:<a a a ;> =
      0 0 0 , 0 0 1 , 0 1 0 , 0 1 1 , 1 0 0 , 1 0 1 , 1 1 0 , 1 1 1.
```

4.2.2 Conditional traversal (backtracking)

`Powerset`, as presented above, does a complete traversal of a powerset tree and collects the labels from all of the terminals. If the tree is viewed as a search tree, and the terminals as solutions, this amounts to finding all possible solutions. Now su-15999.7(to)0.7(the)]mq31.200-0.56020Td1999.3(a)-113(th)10 visit a number of nodes growing exponentially with the depth of the tree before finding a solution. `Powertest:< a a a a ...  a ;>` represents the root node of the search tree, as before. The definition of `Powertest` is: [24]

```
Powertest:
    <;> == Test:<>                          % leaf node.
    <a> == <if Powertest:<Pv:<> 0 !> ! >    % search left subtree.
    <if !> == Powertest:<Pv:<> 1 !>         % left branch result = (),
                                            % so search right subtree.
    <if> == Pv:<>.                          % NPE: left branch result /= ().
```

We again make use of a non-`;`-terminated final argument (though the argument list is `!`-terminated).[25] The terminal nodes are recognised, as before, by an empty distance list. The control logic is as follows: When the recursive call to search the left subtree is made, a *CONTROL* prefix is added, and the default extension will be inserted following `!`. If the left branch does not contain a solution, then `Powertest` returns a nil list, and the instantiated evaluable path will be `<if () !>`, which matches the prefix `<if !>` since the empty list 'disappears'. What remains after chopping is the default extension from case `<a>`, which is the entire argument list. If a solution *was* found, the result from `Test` lies between '`<if`' and '`!>`', so `Pv`

A few typical theorems of `Argc` are:

```
Argc: <!> = 0
      <; !> = 1
      <1 2 ; !> = 1
      <1 2 ; 3 2 ; 1 2 ; 4 5 ; !> = 4
      <; ; ; ; ; ; ; ; ; ; !> = 1 0.
```

```
I:<>==Arg1.
N:<>==Arg2.

For: <> == <argc Argc ;>
   <argc 1 ;> == For:<0 ;>
   <argc 2 ;> == <If:<Equal:<>>>
   <then> == () % stop
   <else> == Body:<I:<> ; !>
             ;
             For:< Add:<1 ; I:<> ; > ; N:<> ; !>.
```

The initial invocation supplies just one argument, `N`, and the argument list must be `!`-terminated, as the argument list is immediately passed to `Argc` to count the number of argument terminators (`;`) preceding the argument list terminator. On the initial call this is case `<argc 1>`, where `For` introduces a new argument, `I`, with initial value `0`, and `N` become the second argument (carried over as the default extension).[28] Thus the argument list `<N ; !>` has been changed to `<I ; N ; !>`. On subsequent calls, `Argc` evaluates to `2`, and the argument names `I` and `N` can be used (on the initial call `N` was the first argument, but no direct access was required). Selector `<case 2>` passes the argument list to `Equal`, a monadic function which compares the first two arguments for equality.[29] If `I` = `N`, iteration stops (recursion terminates). Otherwise, Foreach:<x_1 ; x_2 ; $\ldots$; x_n ; !> = Body:<x_1> ; Body:<x_2> ; $\ldots$; Body:<x_n> ;

The value of `Foreach` on an argument list is a sequence consisting of the value of `Body` on the first argument, followed by the value of `Foreach` on the rest of the argument list:

```
Foreach:
   <!> == ()
   <> == Body:<Arg1 ; !> ; <Pop_arg:<> !>.
```

Some theorems, assuming the same `Body` as above, are:

```
Foreach: <!> = ()
         <2 7 ; !> = 5 4 ;
         <1 2 ; 7 ; 3 8 ; !> = 2 4 ; 1 4 ; 7 6 ;
         <1 ; 2 ; 3 ; !> = 2 ; 4 ; 6 ;.
```

5 Prolog in DA R

In the introduction we alluded to the possibility that any pure Prolog program could be translated into DATR. Indeed, this is our conjecture, and we suspect that the DATR tools required to do so consist of no more than those developed in Moser (1992) and in this paper. The obvious way to go about this is to prove a complexity result for DATR which places it in the same class as Prolog – a topic of continuing research.

References

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            args.dtr                                            %
% Purpose:         Tools for manipulating paths as argument lists.     %
% Authors:         Lionel Moser, December 1991.                        %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr                                            %
% Version:         4.00                                                %
%     Copyright (c) University of Sussex 1991.  All rights reserved.   %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preliminaries:
%    All of the node definitions assume that variable $terminal has already
% been set. $terminal is the set of atoms in a particular alphabet (or
% descriptive domain).
%
%   E.g.,
%      #vars $terminal: 0 1 2 3 4 5 6 7 8 9 & | ~.
%      #load 'args.dtr'.
%
% This file provides tools for manipulating a path as an argument list,
% where the symbols in the descriptive domain of the theory are the
% alphabet over which arguments' values are defined.
% Tools included in this kit support argument extraction, manipulation,
% and path-to-value conversion (Pv).
%
% A path interpreted as an argument list has the format
%      <Arg1 ; Arg2 ; ... Argn ; ! Default_extn>
%  where
%      Argi -  is a sequence of symbols over the alphabet $terminal;
%      ;    -  is the argument terminator;
%      !    -  is the argument list terminator.
%
%
% Miscellaneous
%
%   1. Many of these primitives contain identical definitions, which could
%      be collected at two or three nodes. Indeed, they are mostly variations
%      on Arg1 and Pv, with exceptions (ie, defaults with exceptions).
%      However for debugging reasons it is more convenient to have them
%      separate. Since the error message is defined for the default (<>),
%      it would come from the inherited-from primitive. In order to get a
%      message back from the inheriting primitive global inheritance would be
%      required.
%   2. No use is made of global inheritance, as this could interfere
%      with the application.

% PRIMITIVE DEFINITIONS
%
% Arg1 returns the first ;-terminated argument. The ; terminator is removed.
% (It's easy to replace if it's needed.)
% If ; is the first symbol, a nil list is returned.
Arg1: <> == '**** ERROR: (Arg1) Unknown symbol'
   <;> == ()
   <$terminal> == ($terminal <>).

% Arg2 returns the second ;-terminated argument. The ; terminator is removed.
% If ; is the first symbol of the second arg, a nil list is returned.
% At least two arguments must be present in the arg list.
Arg2: <> == Arg1:<Pop_arg>.

% Arg3 returns the third ;-terminated argument. The ; terminator is removed.
% If ; is the first symbol, a nil list is returned.
% At least three arguments must be present in the arg list.
Arg3: <> == Arg1:<Pop_arg:<Pop_arg>>.

% First returns the first symbol in the first argument.
% The argument must contain at least one symbol.
First: <> == '**** ERROR: (First) Invalid argument'
   <;> == '**** ERROR: (First) Nil list'
   <$terminal> == $terminal.

% Second returns the second symbol in the first argument.
% The argument must have at least two symbols.
Second: <> == '**** ERROR: (Second) Invalid argument'
   <;> == '**** ERROR: (Second) Nil list'
   <; ;> == '**** ERROR: (Second) List too short'
   <$terminal> == First:<>.
```

```
% Top is the same as First. It is a nicer notation when the argument
% is viewed as a stack. It could be defined in terms of First, but
% the default (<>) would then be a message from First instead of Top.
Top: <> == '**** ERROR: (Top) Invalid argument'
   <;> == '**** ERROR: (Top) Nil list'
   <$terminal> == $terminal.

% Pop_arg returns an argument list with the first argument removed.
% The argument must be ;-terminated.
Pop_arg:  <> == '**** ERROR: (Pop_arg) Invalid symbol'
   <!> == ()
   <;> == Arglist:<>
   <$terminal> == <>.

% Pv performs path-to-value conversion.
% All symbols are converted, including argument terminators, up to !.
% A nil argument will return a nil list.
Pv: <>  == ()      % Can't flag unknown symbols here.
   <!> == ()      % special case: remove trailing default extension.
   <;> == (; <>)
   <$terminal> == ($terminal <>).

% Arglist is a better name for Pv when it is the entire argument
% list that is being reconstructed.
Arglist:<> == Pv.

% Rest returns everything in the ;-terminated argument following
% the first symbol (a tail operator).
Rest: <> == '**** ERROR: (Rest) Unknown symbol'
   <;> == ()
   % First symbol
   <$terminal> == Pv_to_;:<>.

% Pop is the same as Rest. It's a better notation when the arg
% is viewed as a stack.
Pop: <> == '**** ERROR: (Pop) Unknown symbol'
   <;> == ()
   % First symbol
   <$terminal> == Pv_to_;:<>.

% Pv_to_; returns a path-to-value conversion, stopping at the end of
% the first argument.
Pv_to_;:<> == '**** ERROR: (Pv_to_;) Unknown symbol'
   <!>  == '**** ERROR: (Pv_to_;) Missing argument terminator'
   <;> == ()
   <$terminal> == ($terminal <>).
```

```
% Reverse returns the ;-terminated argument reversed, minus the terminator.
% Sample theorem:
%    Reverse:< 1 2 3 4 ;> = (4 3 2 1).

Reverse: <> == 'Error: (Reverse) Unknown symbol'
   <;> == ()
   <$terminal> == (<> $terminal).

% Remove_last removes the last symbol in a list. The list must contain at
% least one symbol. Remove_last ignores all but the first argument.
%
% Remove_last:<X0 X1 ... Xn-1 Xn ;> == (X0 X1 ... Xn-1).
%
% Sample theorems:
%    Remove_last: <1 2 ;> = (1 2 3)
%                 <3 2 1 4 ; 2 1 4 ;> = (3 2 1)
%                 <3 ;> = ().
%

Remove_last: <> == '**** ERROR: (Remove_last) Invalid argument'
   <;> == '**** ERROR: (Remove_last) Missing argument'
   <> == Reverse:<Rest:<Reverse ;> ;>.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:           arglogic.dtr                                         %
% Purpose:        Generalised logical equality operators using argument %
%                 manipulation.                                        %
% Authors:        Lionel Moser, December 1991.                         %
% Documentation:  HELP
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:             add10.dtr                                          %
% Purpose:          Illustrate decimal arithmetic in DATR.             %
% Authors:          Lionel Moser, September 1991.                      %
% Documentation:    HELP *datr                                         %
% Related Files:    lib datr; args.dtr, dadd.dtr                       %
% Version:          4.00                                               %
%      Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Base 10 addition
% ----------------
%
% This file implements decimal addition. It's purpose is to illustrate how
% the path can be used to represent an argument list, and how the arguments
% can be extracted and modified. The node definitions which perform the
% argument manipulation are in args.dtr. The Node which is called to perform
% addition is called Add.

% Tools for argument manipulation.
#vars $terminal: 0 1 2 3 4 5 6 7 8 9.
#load 'args.dtr'.

% Decimal digit addition table
#load 'dadd.dtr'.

% Add adds two decimal numbers X and Y.
% Add:<Xn Xn-1 ... X1 X0 ; Ym Ym-1 ... Y0> = (X + Y).
%    where X = Xn ... X0
%          Y = Ym ... Y0
% Add transforms the arguments into the appropriate form for RevAdd,
% which does the actual addition.
% Example theorem:
%   Add:<1 2 ; 7 3 ;> = (8 5).
%
% The arguments for RevAdd are: A carry (initially 0); the digits of X,
% in reverse order; and the digits of Y, in reverse order.

Add: <> == RevAdd:<0 ; Reverse:<Arg1 ;> ; Reverse:<Arg2:<> ;> ; ! >.

% RevAdd embodies a recursive definition of addition.
% RevAdd:<carry ; X0 X1 ... Xn ; Y0 Y1 ... Ym ;>
%    where
%       X = Xn Xn-1 ... X0
%       Y = Ym Ym-1 ... Y0
%
%    Note that the numerical arguments are in reverse order, ie.,
% the integer 123 is represented as (3 2 1). This is because
```

%

```
% Carry and Digit are for accessing the decimal digit addition table in
% 'dadd.dtr'. The table is accessed by node Dadd, which derives theorems
% of the form:
%    Dadd:< old_carry ; digit ; digit ;> = (new_carry ; remainder ;).
%
% Carry:<old_carry ; digit ; digit ;> == new_carry.
% Sample theorem:
%    Carry: <0 ; 9 ; 3 ;> = 1.

Carry: <> == Arg1:<Dadd>.

% Digit:<old_carry ; digit ; digit ;> == new_digit.
% Example theorem:
%    Digit: <0 ; 9 ; 3 ;> = 2.

Digit: <> == Arg2:<Dadd>.

% First (see args.dtr) returns the first item in a ;-terminated list,
% but its argument must be non-empty. In this application one of the
% integers may be shorter than the other (ie, n != m). We make up the
% "missing" digits with zeros.
% Sample Theorems:
%     First_dig:<X0 X1 ... Xn ;> == X0
%     First_dig:<;> == 0

First_dig:
   <;> == 0
   <>  == First.

%--- Some theorems ------------------
%
% Add: <0 ; 0 ;> = (0)
%      <1 ; 1 ;> = (2)
%      <0 0 0 1 ; 1 ;> = (0 0 0 2)
%      <2 ; 3 ;> = (5)
%      <1 2 ; 7 3 ;> = (8 5)
%      <1 2 ; 3 4 ;> = (4 6)
%      <9 9 ; 3 4 ;> = (1 3 3)
%      <1 8 6 ; 3 2 0 ;> = (5 0 6)
%      <1 ; 3 4 5 ;> = (3 4 6)
%      <3 4 5 ; 1 ;> = (3 4 6)
%      <2 5 1 ; 5 5 ;> = (3 0 6)
%      <9 9 9 9 9 3 ; 8 ;> = (1 0 0 0 0 0 1).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            dadd.dtr                                            %
% Purpose:         Store a table of decimal digit additions.           %
% Authors:         Lionel Moser, September 1991.                       %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr; args.dtr, add10.dtr                       %
% Version:         4.00                                                %
%     Copyright (c) University of Sussex 1991.  All rights.gu15H3920TjsdTd(183.5)Tj10.31990Td(%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:             count.dtr                                          %
% Purpose:          Count occurrences of a symbol in a list.           %
% Author:           Lionel Moser, September, 1991.                     %
% Documentation:    HELP *datr                                         %
% Related Files:    lib datr, add10.dtr, args.dtr                      %
% Version:          4.00                                               %
%      Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#vars  $terminal: a b c d e f 0 1 2 3 4 5 6 7 8 9.
#load 'args.dtr'.
#load 'add10.dtr'.

% Count:<A ; X1 X2 ... Xn ;> == # of A's in X
Count:
   <$terminal ; ;> == (0)
   <$terminal ; $terminal> == Add:<1 ; Count:<$terminal ; Arg1:<>  ; !> ; ! >
   <$terminal ; > == Count:<$terminal ; Rest:<Arg1:<> ;> ; ! >.

% --- Some theorems ------
%
% Count: <a ; d e f b c ;> = 0
%        <a ; a b c ;> = 1
%        <b ; a b a b f b f b c a ;> = 3
%        <b ; a b a b f b c b b b d e f b b f b e b b e b ;> = 1 2.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%                                                                         %
% File:            sum.dtr                                                %
% Purpose:         Add up a list of decimal integers.                     %
% Authors:         Lionel Moser, December 1991.                           %
% Documentation:   HELP *datr                                             %
% Related Files:   lib datr; args.dtr, dadd.dtr                           %
% Version:         4.00                                                   %
%     Copyright (c) University of Sussex 1991.  All rights reserved.      %
%                                                                         %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

#vars $terminal: [] 0 1 2 3 4 5 6 7 8 9.
#load 'args.dtr'.
#load 'arglogic.dtr'.
#load 'add10.dtr'.


% Sum:< X1 ; X2 ; ... Xn ; [] ;> == sum of Xi's

Sum:
   <> == <If:<Equal:<Arg2 ; [] ;>>>
   <then> == Arg1:<>
   <else> == <Add:<Arg1:<> ; Arg2:<> ; !> ;
              Pop_arg:<Pop_arg:<> ! > !
             >.

% ---- Some theorems ----

% Sum: <1 ; 3 ; 5 ; [] ;> = 9
%      <1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; [] ;> = 2 8
%      <1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ; 1 0 ; [] ;> = 5 5
%      <1 ; 3 ; 5 ; 2 9 1 ; [] ;> = 3 0 0.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
% File:            loops.dtr                                          %
% Purpose:         Illustrate a couple of types of iteration.         %
% Authors:         Lionel Moser, September 1991.                      %
% Documentation:   HELP *datr                                         %
% Related Files:   lib datr, args.datr, arglogic.dtr, polyadic.dtr,   %
%                  add10.dtr, dadd.dtr.                               %
% Version:         4.00                                               %
%     Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% --------- Some theorems ----------
%
% For: <0 ; !> = ()
%      <1 ; !> = (0 ;)
%      <2 ; !> = (0 ; 2 ;)
%      <4 ; !> = (0 ; 2 ; 4 ; 6 ;)
%      <1 4 ; !> = (0 ; 2 ; 4 ; 6 ; 8 ; 1 0 ; 1 2 ; 1 4 ; 1 6 ; 1 8 ;
%                   2 0 ; 2 2 ; 2 4 ; 2 6 ;).


% Another type of iteration is the Foreach loop. Here the `argument' is an
% argument list. Foreach returns an argument list, whose elements are
% the result of evaluating Body on each argument in the original argument list.

% Foreach:<X1 ; X2 ; ... ; Xn ; !> == (Body:<X1> ; Body:<X2>; ...; Body:<Xn>).
Foreach:
   <!> == ()
   <> == (Body:<Arg1 ; !> ; <Pop_arg:<> !>).

% A more verbose version of the same algorithm, but is independent of
% what symbol is used as the argument delimiter, is:
Foreach1:
   <> == <If:<Equal:<0 ;  Argc ; >>>
   <then> == () % stop
   <else> == (Body:<Arg1:<> ; !> ; Foreach1:<Pop_arg:<> !>).

% --- Some theorems of both Foreach and Foreach1 ------
% Foreach: <!> = ()
%          <2 7 ; !> = (5 4 ;)
%          <1 2 ; 7 ; 3 8 ; !> = (2 4 ; 1 4 ; 7 6 ;)
%          <1 ; 2 ; 3 ; !> = (2 ; 4 ; 6 ;).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            polyadic.dtr                                        %
% Purpose:         Illustrate defintion of polyadic functions.         %
% Authors:         Lionel Moser, September 1991.                       %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr, args.dtr, add10.dtr, dadd.dtr.            %
% Version:         4.00                                                %
%     Copyright (c) University of Sussex 1991.  All rights reserved.   %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Polyadic functions have variable arity, that is, they take a varying
% number of arguments. I show how to count the number of incoming arguments,
% and how to compute different functions based on that number.

#vars $terminal: 0 1 2 3 4 5 6 7 8 9 undef.
#load 'args.dtr'.
#load 'add10.dtr'.

% Argc:<Arg1 ; Arg 2 ; ... Argn ; !> == n.
% Argc counts the number of argument delimiters (;) preceding the
% argument list terminator (!). The number of arguments may be zero,
% but the ! terminator is not optional.

Argc: <> == '**** ERROR: (Argc) Missing ! or invalid symbol'
      <;> == Add:<1 ; <> ;>
      <$terminal> == <>
      <!> == 0.

% Sample theorems:
%   Argc: <!> = 0
%         <; !> = (1)
%         <1 2 ; !> = (1)
%         <1 2 ; 3 2 ; 1 2 ; 4 5 ; !> = (4)
%         <; ; ; ; ; ; ; ; ; ; !> = (1 0).


% F is a polyadic function.
% NOTE: The arg list terminator ! is not optional.
% F:<Arg1 ; Arg2 ; ... Argn ; !>
F: <> == <case Argc ;>
   <case 0 ;> == '0 args'
   <case 1 ;> == ('1 arg:'  Arglist:<>)
   <case 2 ;> == ('2 args:' Arglist:<>)
   <case> ==   ('the' Arg1:<> 'args are:' Arglist:<Pop_arg:<> !>).

% --- Some theorems -----
%  F: <!> = 0 args
%     <1 2 3 ; !> = (1 arg: 1 2 3 ;)
%      <1 2 3 ; 4 5 6 ; !> = (2 args: 1 2 3 ; 4 5 6 ;)
%      <1 2 ; 3 4 5 ; 6 7 ; !> =
%                         (the 3 args are: 1 2 ; 3 4 5 ; 6 7 ;)
%      <1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ; 1 0 ; 1 1 ; 1 2 ; !> =
%                         (the 1 2 args are: 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ;
%                          9 ; 1 0 ; 1 1 ; 1 2 ;).
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%                                                                           %
% File:            pascal.dtr                                               %
% Purpose:         Compute Pascal's triangle                                %
% Author:          Lionel Moser, December  1991                             %
% Documentation:   HELP *datr                                               %
% Related Files:   lib datr; args.dtr; arglogic.dtr; add10.dtr;             %
% Version:         2.00                                                     %
%     Copyright (c) University of Sussex 1991.  All rights reserved.        %
%                                                                           %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

#vars $terminal: [ ] 0 1 2 3 4 5 6 7 8 9.
#load 'args.dtr'.
#load 'arglogic.dtr'.
#load 'add10.dtr'.

% Pascal's triangle looks like this:
%
%                1
%              1   1
%            1   2    1
%          1   3   3   1
%        1   4   6   4   1
%      1   5  10   10  5   1

% Successor computes the (n+1)th line of the triangle as a function of the
% nth line.

Successor:
   <[ 1 ; > == ([ 1 ; <1 ;>)
   <> == (Add:<Arg1 ; Arg2 ; !> ; <Pop_arg ! >)
   <1 ; ]> == (1 ; ]).

% ---- Some theorems ----

% Successor: <[ 1 ; ]> = ([ 1 ; 1 ; ])
%            <[ 1 ; 1 ; ]> = ([ 1 ; 2 ; 1 ; ])
%            <[ 1 ; 2 ; 1 ; ]> = ([ 1 ; 3 ; 3 ; 1 ; ])
%            <[ 1 ; 3 ; 3 ; 1 ; ]> = ([ 1 ; 4 ; 6 ; 4 ; 1 ; ])
%            <[ 1 ; 4 ; 6 ; 4 ; 1 ; ]> = ([ 1 ; 5 ; 1 0 ; 1 0 ; 5 ; 1 ; ]).
```

```
% node.  The latter sequence (a search tree path) is used to prefix the
% values of the leaves of the subtree of which the current node is the root.
%    Arguments are separated by ; (the "argument delimiter"). The symbol !
% is the "argument list terminator". Default extensions which are appended
% following the argument list appear after the terminator and are ignored.

Powerset:
    <;> == Pv:<>
    <a> == (<Pv:<> 0 !> , <Pv:<> 1 !>).

% Pv: Path-to-value conversion.
Pv: <> == ()   % end of path, since all other symbols are handled explicitly,
               % except comma (","), which never occurs.
    <a> == (a <>)
    <0> == (0 <>)
    <1> == (1 <>)
    <;> == (; <>)
    <!> == ().     % discard trailing default extension.

% ---- Some theorems ---------------
%
% Powerset: <;> = ()
%          <a ;> = (0 , 1)
%          <a a ;> = (0 0 , 0 1 , 1 0 , 1 1)
%
%          <a a a ;> = (0 0 0 , 0 0 1 , 0 1 0 , 0 1 1 ,
%                       1 0 0 , 1 0 1 , 1 1 0 , 1 1 1)
%
%          <a a a a ;> = (0 0 0 0 , 0 0 0 1 , 0 0 1 0 , 0 0 1 1 ,
%                         0 1 0 0 , 0 1 0 1 , 0 1 1 0 , 0 1 1 1 ,
%                         1 0 0 0 , 1 0 0 1 , 1 0 1 0 , 1 0 1 1 ,
%                         1 1 0 0 , 1 1 0 1 , 1 1 1 0 , 1 1 1 1).
% --------------------------
% Computing the value of Powerset:<a a a> involves 15 calls to
% Powerset, no two of which are the same. They are listed below in
% their calling sequence. It can be seen that they are all distinct.
% (They are distinct in the prefix preceding the argument list
% terminator, but are not distinct in the suffix following it.)
%
% Powerset:<a a a ;>
% Powerset:<a a ; 0 ! a a ;>
% Powerset:<a ; 0 0 ! a ; 0 ! a a ;>
% Powerset:<; 0 0 0 ! ; 0 0 ! a ; 0 ! a a ;>
% Powerset:<; 0 0 1 ! ; 0 0 ! a ; 0 ! a a ;>
% Powerset:<a ; 0 1 ! a ; 0 ! a a ;>
% Powerset:<; 0 1 0 ! ; 0 1 ! a ; 0 ! a a ;>
% Powerset:<; 0 1 1 ! ; 0 1 ! a ; 0 ! a a ;>
% Powerset:<a a ; 1 ! a a ;>
% Powerset:<a ; 1 0 ! a ; 1 ! a a ;>
% Powerset:<; 1 0 0 ! ; 1 0 ! a ; 1 ! a a ;>
% Powerset:<; 1 0 1 ! ; 1 0 ! a ; 1 ! a a ;>
% Powerset:<a ; 1 1 ! a ; 1 ! a a ;>
% Powerset:<; 1 1 0 ! ; 1 1 ! a ; 1 ! a a ;>
% Powerset:<; 1 1 1 ! ; 1 1 ! a ; 1 ! a a ;>
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% File:           powerset2.dtr                                      %
% Purpose:        powerset done with general backtracking.           %
% Author:         Lionel Moser, September, 1991.                     %
% Documentation:  HELP *datr                                         %
% Related Files:  lib datr, powerset.dtr, args.dtr, etc.             %
% Version:        3.00                                               %
%     Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% powerset.dtr illustrates branching recursive descent, but it has the
% following decisions hardcoded: (1) branching factor of 2; (2) definition of
% successor; (3) the order in which the successors to a node are visited.
% This file illustrates a general backtracking algorithm which is independent
% of these problem-specific details.

#vars $terminal: a 0 1 !!.
#load 'args.dtr'.

% Powerset2 is a general algorithm: Successors generates the successors of a
% given node, and returns them in the order they are to be visited. Test
% defines the value of the terminal.
%
% Powerset2:<Distance !! Label ;> == (t1, t2, ..., t[2^n]),
%    where ti is a terminal of the powerset tree.
Powerset2:
   <!!> == Test:<>
   <a> == Foreach:<Successors !>.

% Modified Foreach so that a delimiter is added only when another value
% follows.
%
% Foreach:<X1 ; X2 ; ... ; Xn ; !> == (Body:<X1> , ..., Body:<Xn>).
Foreach:
   <!> == ()
   <> == (Body:<Arg1 ; !>  <continue Pop_arg:<> !>)
   <continue !> == ()
   <continue> == (, <Pv:<> !>).

Body:<> == Powerset2.

% We're only enumerating the leaves, so everyone's a winner, babe.
Test: <> == Arg1.            % success
```

```
% Successors:< Distance !! Label ;> == (Rest:<Distance> !! Label 0 ;
%                                        Rest:<Distance> !! Label 1 ;)
Successors:
   <a> == (Distance:<> !! Label 0 ; Distance:<> !! Label 1 ;).

% Virtual arguments, defined in terms of !!-delimitation.
% Distance:<distance !! label ;> == distance.
Distance:
   <!!> == ()
   <a> == (a <>).

% Label:<distance !! label ;> == label.
Label:
   <!!> == Arg1:<>
   <a> ==  <>.

% Note: It would be nice if the above could be written as
% Distance:
%    <!!> == ()
%    <$terminal> == ($terminal <>).
% But !! is a terminal, so the definition, under the current definition, would
% be non-functional.

% --- Some theorems --------
%
% Powerset2: <a !! ;> = (0 , 1)
%            <a a !! ;> = (0 0 , 0 1 , 1 0 , 1 1)
%
%            <a a a !! ;> = (0 0 0 , 0 0 1 , 0 1 0 , 0 1 1 ,
%                              1 0 0 , 1 0 1 , 1 1 0 , 1 1 1)
%
%             <a a a a !! ;> = (0 0 0 0 , 0 0 0 1 , 0 0 1 0 , 0 0 1 1 ,
%                                0 1 0 0 , 0 1 0 1 , 0 1 1 0 , 0 1 1 1 ,
%                                1 0 0 0 , 1 0 0 1 , 1 0 1 0 , 1 0 1 1 ,
%                                1 1 0 0 , 1 1 0 1 , 1 1 1 0 , 1 1 1 1).
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%                                                                             %
% File:             powertest.dtr                                             %
% Purpose:          powertest
% Pv: Path-to-value conversion.
Pv: <> == ()   % end of path, since all other symbols are handled explicitly
    <a> == (a <>)
    <0> == (0 <>)
    <1> == (1 <>)
    <;> == (; <>)
    <!> == ().     % discard trailing default extension.

% --- Some theorems ------
%  Powertest: <a ;>
```