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 which is designed to support the lexicon in an NLP system. Its syntax and semantics are designed to support the types of inference required in computational lexicography. It was not a design intention of the language to support general logic programming, yet in this paper we show that the types of inference permitted in the language do support a general type of logical inference. Drawing an analogy with Prolog, both are declarative languages, and each has its own inference engine or theorem prover, which are quite different. DATR allows at least a subset of Prolog-definable logic programs to be encoded.

*I am indebted to Jo Calder, Roger Evans, Gerald Gazdar and Adam Kilgarriff for conversations about this work. Research for this paper is partially supported by an Overseas Research Scholarship (ORS/C17/3) from the Committee of Vice-Chancellors and Principals of the United Kingdom (CVCP), and a scholarship from Fonds pour la formation de chercheurs et l'aide á la recherche from the Government of Québec.

Contents

1 Introduction **3**

2 Nodes as function definitions; Paths as arguments **4**

2.1 Path-to-value conversion 4
2.2 The path as a single argument 5
2.3 Nodes as functions of more than one argument; Paths as argument lists 7
2.3.1 Extracting arguments 7
2.3.2 Argument list termination 8
2.4 Chopping 9

3 Control structures **10**

3.1 CASE statements 10
3.2 IF-THEN-ELSE Structures 11

4 Applications **13**

4.1 Decimal arithmetic 13
4.2 Backtracking 15
4.2.1 Tree traversal 16
4.2.2 Conditional traversal (backtracking) 17
4.3 Polyadic Functions 18
4.4 Loops 19
4.4.1 Definite iteration 19
4.4.2 Foreach iteration 20

5 Prolog in DATR **21**

DATR Examples **22**

args.dtr - Tools for manipulating paths as argument lists 22
arglogic.dtr - Generalised logical equality operators using argument 26
add10.dtr - Illustrate decimal arithmetic 27
dadd.dtr - Table of decimal digit addition 30
count.dtr - Count occurrences of a symbol in a list 32
sum.dtr - Add up a list of decimal integers 33
loops.dtr - Illustrate a couple of types of iteration 34
polyadic.dtr - Illustrate definition of polyadic functions 36
pascal.dtr - Compute Pascal's triangle 38
powerset.dtr - To illustrate theories which can take exponential time 41
powerset2.dtr - powerset done with general backtracking 44
powertest.dtr - Illustrate backtracking tree searching 46

1 Introduction

While Prolog and DATR are both declarative logic programming languages with built-in inference engines, the former is intended as a tool for a more general class of applications, while the latter is designed for the more narrow application of lexical representation for feature-based grammar formalisms. Assuming that the features of DATR are limited to those required for its intended domain of application, it is interesting to see that a rather general class of recursive functions can be stated declaratively in DATR.

Prolog explicitly supports computation in a second order logic, having the univ operator. DATR may also support such computation through a combination of evaluable paths and global inheritance, though we do not discuss these issues directly in this paper. With respect to first-order theories, DATR is rather *more* declarative than Prolog: most importantly, there is no mechanism to *assert* and *retract* clauses;[1] and there are no variables, even as formal parameters. A Prolog clause which uses variables as formal parameters is not directly representable in DATR. An example is

$p(X) \leftarrow q(X), r(X).$

which denotes that p is true of X if q and r are true of *the same* X.[2] DATR has no variables of this type. Nevertheless one can express the same intention. q computes some function to derive a value for X, and we might call this function f. (We can assume that q states f declaratively, and therefore f is also a declarative statement of the value which satisfies it.) Now q is true of $f()$ (the value of f), and r is also true of $f()$. So we can state the proposition expressed by p as[3]

$p(f()) \leftarrow q(f()), r(f()).$

Every occurrence of $f()$ denotes the same value. In a DATR theory, a path evaluated at a node always derives the same extensional value (if one exists). But what about the alternative values for X which Prolog might find during backtracking? DATR has a deterministic semantics, and backtracking never occurs. It would seem on the surface that even the set of functions representable in 'pure Prolog'[4] is a superset of those representable in DATR, but intuition may be misleading. Backtracking, say over a proof tree, implies some type of recursive descent through choice points, and unfolding the recursive invocations back up to some choice point, when a non-goal terminal is reached, to try some alternative branch. The same algorithm can be stated deterministically, by viewing 'descending' as moving forward in a downward direction, and 'backing up' as moving forward in an upward direction. One need only save at all times the information required to compute the choice points whose choices have not been exhausted, and their successors remaining to be searched. If the successors of a particular node are computed all at once, then the list of successors still unsearched must be stored while each is searched in turn (assuming sequential processing). This storage, and some execution control, one gets 'for free' in the form of the execution stack in a backtracking language. One can do it explicitly by maintaining lists of choice points and alternatives. DATR does not backtrack; but one can certainly implement backtracking algorithms in DATR. Although Prolog's built-in proof mechanism performs depth-first left-to-right searching, it is possible to express in this paradigm breadth-first searching algorithms (see any introductory textbook on Prolog programming, (*e.g.*, Sterling and Shapiro (1986, pp. 271–273))). Similarly, it is possible to express within DATR's existing semantics algorithms which perform other than it's 'natural' theorem-proving strategy. Both in Prolog and DATR there are penalties in time and space complexity, but not necessarily in program length.

In the following sections we develop a scheme by which a path can be interpreted as a single argument, and show how function composition can be modelled. We then generalise this to interpreting paths as an argument list, and again illustrate mechanisms to implement function

[1] These are extra-logical operations. They allow real variables whose values can be updated.
[2] If X is free when p is invoked, and q binds X then r must satisfy the same value for X as q does.
[3] This is not Prolog notation, since in Prolog $f()$ would be a term, not a function call.
[4] *I.e.*, first-order *assert*-, *cut*-, *write*- and *read*-free programs – those which do not use extra-logical operations.

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; so we partition *ATOM* into an alphabet `$terminal`, the set of symbols over which arguments may be constructed, and *ATOM* - `$terminal`, about which we have nothing more to say.

Converting a value to a path (or portion of a path) is built-in in the form of evaluable paths. Converting a path into the argument it represents involves what we call *path-to-value conversion*, an operation we define for each symbol of `$terminal`, and recursively for sequences over this alphabet.

While some members of *FEAT* are atoms, there are other symbols which appear in paths which are distinctly *not* members of *ATOM*. We define the set of such symbols as the difference $CONTROL = FEAT - ATOM$. Members of *CONTROL* do not have the property of being convertible into values.

We begin by defining path-to-value conversion for a sequence of symbols over `$terminal`.

2.1 Path-to-value conversion

In order to define path-to-value conversion, we need an enumeration of the atoms in the alphabet `$terminal`. We begin by defining function `Pv0`, which performs path-to-value conversion for the alphabet {`a, b, c, d`}.

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

Some theorems of `Pv0` are:

[5] In fact, in this paper we use the terms 'node' and 'function' synonymously, and the terms 'path' and 'argument list' synonymously, unless context indicates otherwise.

[6] To avoid confusion, we adopt the following terminological conventions: we refer to <p_1 p_2> as a *path extension* of <p_1>. Where <p_1> is the longest defined prefix of <p_1 p_2>, we refer to p_2 as a *default extension* of <p_1>. The 'extension of a path' (its value) we refer to as its *extensional value*, or simply its *value*.

[7] The 'usual' way one writes a DATR theory is in fact the reverse: one has a lexical theory in which these sets are explicit, and the DATR representation follows.

[8] Every DATR theory is defined over finite sets *FEAT* and *ATOM*; so far we have only given them names. The set of inferences derivable from the theories we will define remains infinite, as it is for every DATR theory.

```
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` requires a non-nil argument, so should `Rest`. We will not pursue this further.

yielding the following theorems:[14]

```
Third: <a b c d a a ;> = c.
Rest2: <a b c d a a ;> = c d a a.
```

2.3 Nodes as functions of more than one argument; Paths as argument lists

In order to define nodes corresponding to functions of more than one argument, we need a mechanism for interpreting a single sequence of path constituents as a sequence of separate arguments, and of converting path constituents comprising a single argument into the value it represents.

A path interpreted as an argument list has the format $< Arg_1 \;\; ; \;\; Arg_2 \;\; ; \;\; \ldots \; ; \;\; Arg_n \;\; ;>$ where

Arg_i is a sequence of symbols over the alphabet `$terminal`; and

; is the argument terminator.

We will define nodes which when queried on a path evaluate to one or other of the arguments.[15] It will be convenient to define a path-to-value conversion operation which performs path-to-value conversion on an entire argument list. This simply requires converting terminators as well as symbols in `$terminal`, a modification of `Pv1` which we call `Pv`:

```
Pv: <>  == ()
    <;> == ; <>
    <$terminal> == $terminal <>.
```

Although `Pv` permits theorems such as the following:

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

note that the first one is not a well formed argument list as the last argument is not terminated. `Pv` is an underlying primitive – called primarily by the primitives we now define.

2.3.1 Extracting arguments

We begin with `Arg1`, which returns the first ;-terminated argument in the argument list. If ; is the first symbol, a nil list is returned. Again, the terminator itself is removed:

```
Arg1:
   <;> == ()
   <$terminal> == $terminal <>
   <> == 'invalid symbol'.
```

A typical theorem is `Arg1:<a b c ; b b ; d d ;> = a b c`. Next we define function `Pop_arg`, which returns the argument list minus the first argument. The argument list must contain at least one argument, and which must be terminated:

[14] It is crucial that arguments be terminated, as `Pv_to_;` (and hence `Rest`) map the terminator and any path extension of it to the nil list. When a path-to-value conversion is performed inside an evaluable path, the semantics of DATR cause the default extension to be appended to the constructed path. For example, if the constructed argument in `Rest2` were not terminated (*i.e.*, `Rest2:<> == Rest:<Rest>.`) then resulting theorem would have been `Rest2: <a b c d a a ;> = c d a a a b c d a a`.

[15] The default extension may add more arguments as the argument list is passed down, but a function which requires n arguments and accesses only the first n arguments will not be affected by this surfeit.

```
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:

Initial query	Derived value	Justification
`Arg2:<a a ; b b ; d d ;> =`	`Arg1:<Pop_arg:<a a ; b b ; d d ;>` `a a ; b b ; d d ;>`	`Arg2:<> == Arg1:<Pop_arg>`
`=`	`Arg1:<b b ; d d ;` `a a ; b b ; d d ;>`	`Pop_arg:<a a ; b b ; d d ;>` `= b b ; d d ;`
`=`	`b b`	`Arg1` definition

Derivation of `Arg2:<a a ; b b ; d d ;> = b b.`

Note the reoccurrence of the incoming path, which is effectively ignored. This duplication poses no problems for the primitives so far defined, but could have adverse effects if unexpected arguments are accessed, since it effectively adds arguments to the argument list. We avoid this problem by introducing an argument list terminator.

2.3.2 Argument list termination

Although `Arg2` is not 'confused' by the presence of 'extra' trailing arguments due to DATR's default extension inference rule[16], it is nonetheless convenient to include the notion of an argument list terminator, which we notate `!`. This simplifies some of the argument manipulation primitives and allows certain error trapping – for example in the case of a missing argument. Judicious insertion of `!` at the end of a constructed evaluable path will, upon instantiation, separate the nominal argument list from the default extension. Primitive operations treat `!` just as `Arg1` treats `;` – by mapping it and arbitrary extensions to an empty list.[17]

To illustrate, suppose we want to define the constant function A by A:<Arg_1>= B:<`a b c`>. That is, the value of A invoked on *any* argument is the value of B invoked on the constant argument `a b c ;`. Consider the following definition of A:

```
A: <> == B:<a b c ;>.
```

If `A` is called with argument `<w y z ;>`, `B` will be called with two arguments: `<a b c ; w y z ;>`. So the function defined by `A` is $A(Arg_1) = B($<`a b c`>$, Arg_1)$; that is, `B` is a function of two arguments, the first of which is always the (constant) list `(a b c)`. The presence of extraneous arguments is of no consequence if `B` is monadic. On the other hand, if `B` is polyadic then the result from `B` is unlikely to be correct, and the definition of `A` is definitely incorrect. For this reason we append `!` to the argument list, to delimit the nominal argument list from the default extension. Adopting the convention that everything following `!` be ignored, the correct definition of `A` is:

```
A: <> == B:<a b c ; !>.
```

[16]When an evaluable path is instantiated, the default extension is appended to the constructed value.

[17]The argument terminator plays the syntactic role of terminating an argument and of delimiting the argument from a default extension. Gibbon (1990) uses the mechanism of delimiting the default extension from the query path by an extra-descriptive atom, which is mapped to `()` to effectively discard the default extension.

There is nothing to prevent `B` from 'accessing' the path suffix beginning with `!`; however if `B` uses only the primitives we have defined so far, then it will 'see' only the argument `a b c`.

In the case of monadic functions, the presence of `!` is optional. If a function requires n (a fixed number) of arguments and the first n arguments are those required, then a non-`!`-terminated default extension will never be referenced. On the other hand, the insertion of `!` keeps the primitives 'honest'.[18]

In the case of polyadic functions, the argument list terminator is required to delimit a variable-length argument list from the default extension (as we shall see below).

In view of the distinction between argument lists and 'invisible' extensions, we refine `Pv` to its final form, where it produces the path-to-value conversion of an entire argument list, discarding the argument list terminator and the default extension:

```
Pv: <>  == ()
    <!> == ()    % remove trailing default extension.
    <;> == ; <>
    <$terminal> == $terminal <>.
```

In summary: the instantiation of an evaluable path <*Arglist*> is <*Arglist* {*de*}> where {*de*} is the default extension. The default extension is the entire path on which the node was queried (the *incoming* path), minus whatever prefix was matched on the LHS of `==`. If the constructed (or *outgoing*) argument list is terminated, *i.e.*, <*Arglist* `!`> then the outgoing path instantiates as <*Arglist* `!` {*de*}> where the constructed argument list does not include arguments due to appending of the default extension, and the default extension is invisible to the argument manipulation primitives.

2.4 Chopping

In the previous section we discussed how to distinguish the default extension from intended arguments, and gave a cursory definition of 'default extension'. When the incoming path (the path on which a node is queried) is part of an outgoing path (*i.e.*, the path constructed in a further inheritance specification) there is some control over the form of the incoming path used in the instantiation of the outgoing path: it can either include the prefix which matched the LHS of `==` (*i.e.*, be the entire incoming argument list) or can have the matched prefix removed. We call this prefix removal *chopping*. Chopping, which is implicit in the semantics of DATR, occurs often in the applications below. As it can be confusing, we hope this review of its effect will render them easier to understand.

To extract an argument from the argument list, an extractor primitive is invoked on a path; to form an appropriate path we use chopping to (a) remove atoms which are not members of `$terminal`, (which must not appear in paths on which the argument manipulation primitives operate); and (b) to shift arguments. We make extensive use of appropriate chopping in the example theories which comprise later sections of this paper.

Consider nodes F_i below, which return the path-to-value conversion of their argument list, passed to `Pv`:

```
F1:<a b> == Pv.
F2:<a b> == Pv:<>.
F3:<a b> == Pv:<c>.
F4:<a b c ;> == Pv:<>.
```

[18] In the event that an argument terminator (`;`) is erroneously omitted, the presence of `!` can provoke early feedback that something is amiss.

Evaluating path `<a b c ;>` at each $\mathtt{F}_i$ yields the following theorems:

```
F1: <a b c ;> = a b c ;.
F2: <a b c ;> = c ;.
F3: <a b c ;> = c c ;.
F4: <a b c ;> = .
```

At `F1`, the entire incoming path is passed on to `Pv`, including the matched prefix. At `F2`, the prefix which matched the LHS of `==` (`<a b>`) has been removed (or chopped) from the path. `F3` is an example of *chopping with replacement* – the matched prefix `<a b>` is chopped and replaced with `c`. At `F4` the matched prefix, which is chopped, includes the argument terminator, hence the argument list itself contains one fewer argument, and all of the arguments have been shifted – for example, `F4:<a b c ; b b b ; c c c ;> = Pv:<b b b ; c c c ;>`. Thus *Arg2* at `F4` is *Arg1* at `Pv`.

Chopping is vacuous when the matched prefix is the empty path. For example, definitions `F5a` and `F5b` are equivalent:

```
F5a: <> == Arg2.
F5b: <> == Arg2:<>.
```

3 Control structures

In this section we illustrate how various types of control structures can be simulated in DATR.

3.1 CASE statements

The CASE statement is the most obvious control structure to represent in DATR because it corresponds directly to the standard inference mechanism: the statement executed is the one with the longest leading prefix which matches the query path. If the defined prefixes are all of the same length, this amounts to selecting the one which matches the prefix of the path of that length.

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

Case1:<> == red Pv.
Case2:<> == blue Pv.
Case3:<> == green Pv.
```

Here each case is defined in terms of a different node. If node `J` is called on an argument prefixed with a *CONTROL* sequence (`red`, `blue`, or `green`), then the case definition will be invoked on the argument itself. Consider the first line of the derivations below which indicate that in each case the definition is computed on the original argument (with the *CONTROL* prefix chopped):

```
J:<red a b c d ;> = Case1:<a b c d ;> = ...
J:<blue a b c d ;> = Case2:<a b c d ;> = ...
J:<green a b c d ;> = Case3:<a b c d ;> = ...
```

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`
	=	`Case2:<a b d ;>`	`J:<case blue> == Case2:<>`
	=	`blue Pv:<a b d ;>`	`Case2:<> == blue Pv`
	=	`blue a b d ;`	`Pv:<a b d ;> = a b d ;`

3.2 IF-THEN-ELSE Structures

The IF statement is simply a CASE statement with two cases, and can be locally defined similarly. Node `G` in the last example computed some non-boolean function. Node `Cond`, below, is similar in form to node `G`, but computes a boolean function:

```
Cond:<> == true
  <a> == false
  <a b> == true
  <b> == false
  <c c> == false.

K: <> == <if Cond>
  <if true> == Case1:<>
  <if false> == Case2:<>.
```

Here the selectors are locally defined prefixes `<if true>` and `<if false>`. These selectors can be collected at a central node, which takes a boolean argument and returns the selector.

```
If: <true> == then
    <false> == else.
```

```
K: <> == <If:<Cond1>>
   <then> == Then_fn:<>
   <else> == Else_fn:<>.
```

Since **then** and **else** must be unique prefixes at node K, multiple IF-THEN-ELSE conditions require unique prefixes:

```
K2: <> == <1 If:<Cond1>>
   <1 then> == <2 If:<Cond2:<>>>
      <2 then> == Then_fn1:<>      % Cond1 & Cond2
      <2 else> == Then_fn2:<>      % Cond1 & Not Cond2
   <1 else> == <3 If:<Cond3:<>>>
      <3 then> == Then_fn3:<>      % Not Cond1 & Cond3
      <3 else> == Then_fn4:<>.     % Not Cond1 & Not Cond3
```

Here we assume that all of the $\texttt{Cond}_i$ nodes return boolean values of either **true** or **false**. Note that chopping at all statements is required to remove the *CONTROL* prefix.[19] For example, the path on which Then_fn3 is invoked on the original path on which K2 was invoked.

[19] Chopping is not performed at the default statement, where it would be vacuous.

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 and look again

      <0 ; 0 ; 0 ;> == 0 ; 0 ;
      <0 ; 0 ; 1 ;> == 0 ; 1 ;
      <0 ; 0 ; 2 ;> == 0 ; 2 ;

           ⋮       ⋮       ⋮

      <0 ; 0 ; 9 ;> == 0 ; 9 ;

      <1 ; 0 ; 0 ;> == 0 ; 1 ;
      <1 ; 0 ; 1 ;> == 0 ; 2 ;
      <1 ; 0 ; 2 ;> == 0 ; 3 ;

           ⋮       ⋮       ⋮

      <0 ; 1 ; 1 ;> == 0 ; 2 ;
      <0 ; 1 ; 2 ;> == 0 ; 3 ;
      <0 ; 1 ; 3 ;> == 0 ; 4 ;

           ⋮       ⋮       ⋮

      <0 ; 1 ; 9 ;> == 1 ; 0 ;

      <1 ; 1 ; 1 ;> == 0 ; 3 ;
      <1 ; 1 ; 2 ;> == 0 ; 4 ;

           ⋮       ⋮       ⋮
```

Accessing the table in `Dadd` is done through nodes `Digit` and `Carry`:

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

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

The digit resulting from the addition of two digits plus a previous carry is the value of query `Digit:<old_carry ; digit ; digit ;>`, while the new carry resulting from the same addition is the value of `Carry:<old_carry ; digit ; digit ;>`.

Some theorems of `Dadd`, `Digit` and `Carry` are:

```
Digit:  <0 ; 9 ; 3 ;> = 2.
Carry:  <0 ; 9 ; 3 ;> = 1.
```

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` is monadic, so any default extension that happens to be trailing will have no effect.

We need one more tool before defining addition: `First_dig` returns the first symbol in an argument, or `0` if the argument is the empty list. This corresponds to the case where one of the numbers being added is shorter than the other, and we want to 'pad' the shorter one with zeros to make their lengths equal. `First_dig:<`x_0 x_1 ... x_n `;>` evaluates to x_0, if the list is non empty, or 0 if it is:

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

We define $X + Y$ recursively as follows: Let X and Y be the sequences of digits $x_n x_{n-1} \ldots x_0$ and $y_m y_{m-1} \ldots y_0$, respectively, and let $k = max(n, m)$. Then $X + Y$ is the sequence of digits

$$c_{k+1}\ (c_k + x_k + y_k)\ (c_{k-1} + x_{k-1} + y_{k-1})\ \ldots\ (c_0 + x_0 + y_0)$$

where
x_i is the i-th digit of X, or 0 if $i > n$;
y_i is the i-th digit of Y, or 0 if $i > m$; and
c_i is the carry from column $(i - 1)$, and $c_0 = 0$.
Our definition of addition requires the numbers to be added be entered in reverse order, so the sum of $123 + 456$ is the value of query `RevAdd:<0 ; 3 2 1 ; 6 5 4 ;>`, where the leading zero is the initial carry, c_0.

We recursively compute c_{k+1}, the leftmost digit of the sum, and compute the digits one to the right as recursion unfolds.

```
RevAdd:
  <0 ; ; ;> == ()   % X and Y exhausted; leftmost digit is zero.
  <1 ; ; ;> == 1    % X and Y exhausted; leftmost digit is a carry.
  <> == RevAdd:<Carry:<Arg1 ; First_dig:<Arg2 ;> ; First_dig:<Arg3 ;> ; > ;
                Rest:<Arg2 ;> ;
                Rest:<Arg3 ;> ;
               >
        Digit:<Arg1 ; First_dig:<Arg2 ;> ; First_dig:<Arg3 ;> ; >.
```

Some theorems of this theory are:

```
RevAdd: <0 ; 0 ; 0 ;> = 0
        <0 ; 1 0 0 0 ; 1 ;> = 0 0 0 2
        <0 ; 2 ; 3 ;> = 5
        <0 ; 2 1 ; 3 7 ;> = 8 5
        <0 ; 9 9 ; 4 3 ;> = 1 3 3
        <0 ; 6 8 1 ; 0 2 3 ;> = 5 0 6
        <0 ; 1 ; 5 4 3 ;> = 3 4 6
        <0 ; 5 4 3 ; 1 ;> = 3 4 6
        <0 ; 1 5 2 ; 5 5 ;> = 3 0 6.
```

In order to permit X and Y to be input in their natural order, we introduce the primitive `Reverse` which returns the reverse of its (single) argument (discarding, as usual, the terminator and arbitrary extensions which follow it); *e.g.*, `Reverse:< 1 2 3 4 ;> = 4 3 2 1.`:

```
Reverse:
   <;> == ()
   <$terminal> == <> $terminal.
```

Now we define an addition interface, `Add`, which constructs an argument for `RevAdd` with the numbers reversed, and supplies the initial carry c_0:

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

Some theorems of `Add` are:

```
Add: <0 ; 0 ;> = 0
     <1 ; 1 ;> = 2
     <1 2 ; 7 3 ;> = 8 5
     <9 9 ; 3 4 ;> = 1 3 3
     <1 8 6 ; 3 2 0 ;> = 5 0 6.
```

4.2 Backtracking

As noted in the introduction, backtracking algorithms can be implemented deterministically.[20] The backtracking control built-in in backtracking languages, *i.e.*, automatic stacking and restoring of execution contexts, is also provided in DATR. First, note that in a list or evaluable path each inheritance specification is evaluated independently.

```
A: <> == C D.

B: <> == <C D>
   <a> == 1
   <b> == 2.

C: <> == a.

D: <> == b.
```

When `A:<x>` is evaluated, `C:<x>` and `D:<x>` are evaluated independently (and possibly in parallel). Any context change effected at `C` has no effect on evaluation at `D`, or on forward evaluation at `A`, if there is any. Similarly, evaluation of `B:<x>` involves independent evaluation of `C:<x>` and `D:<x>`, and further evaluation at `B` is unaffected by context changes at `C` or `D`.[21] This is completely equivalent to context saving and restoration upon procedure invocation in procedural languages (and Prolog as well).

We can exploit this context saving and restoration to implement backtracking. We begin by illustrating tree traversal.

[20] Prolog, as well, searches deterministically. Clauses are tried in their order of enumeration, and conjuncts and disjuncts are also evaluated in this order.

[21] Evans *et al.* (1991) take advantage of this fact to show how prioritised multiple inheritance could be reconstructed in DATR.

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 is the label on the node.

Expanded out, such an argument has the format: $< a_n\ a_{n-1}\ \ldots\ a_1\ ;\ \ b_0\ b_1\ \ldots\ b_m\ ;>$. Now, having incorporated the edge label from its parent into the label on the node itself, we can omit the edge labels and still categorise completely the tree. Using this notation, the powerset tree of order 3 is:

```
                                     <aaa; ;>
                       ____________________|____________________
                      |                                         |
                   <aa; 0;>                                 <aa; 1;>
             __________|__________                     __________|__________
            |                    |                    |                    |
         <a; 00;>             <a; 01;>             <a; 10;>             <a; 11;>
       _____|_____          _____|_____          _____|_____          _____|_____
      |          |         |          |         |          |         |          |
  <; 000;>   <; 001;>   <; 010;>   <; 011;>  <; 100;>   <; 101;>   <; 110;>   <; 111;>
```

A simple algorithm to traverse the powerset tree and collect the labels on the terminals is shown in `Powerset1`, assuming the definition `$terminal` = {1,0,a}.[22] `Powerset1:<a a a ; ;>` represents the root node of the powerset tree of order 3, above, where the first argument, *distance_to_leaf*, is a list of 3 `a`'s, and *node_label* is the empty list, the label on the root. The leftmost preterminal node (on the left branch) would be represented by `Powerset1:<a ; 0 0 ;>`, where the distance to a leaf is 1 and the node label is `0 0`. The definition of `Powerset1` is:

```
Powerset1:
            <;>   ==   Arg2                               % distance to leaf = 0;
                                                          % return label.
            <a>   ==   Powerset1:<Arg1:<> ; Arg2:<> 0 ;>  % descend 0-labelled edge;
                                                          % left branch.
                       ,
                       Powerset1:<Arg1:<> ; Arg2:<> 1 ;>. % descend 1-labelled edge;
                                                          % right branch.
```

[22] An interesting point about `Powerset1` is that the symbols `0` and `1` are members of `$terminal`, although they do not appear to appear as values. However, they *do* appear as values in `Pv`, by virtue of being enumerated as members of `$terminal`. An interesting question is whether the declaration of `$terminal` could be generated automatically by a compiler, and this case indicates that distinguishing `$terminal` from *CONTROL* is not straightforward.

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 suppose that we have some definition of goal satisfaction, we want to test terminals as *possible* solutions, and we are interested in the first terminal which satisfies the definition of goal.

`Powerset` does a recursive descent on the left branch followed by a recursive descent on the right branch, its only 'decision-making' being to terminate recursion at terminal nodes, or to recurse at non-terminal nodes. By inserting a test between the two recursive descents, we can decide whether a solution has been found in the left-subtree, or whether the right subtree must also be searched. Instead of simply collecting the labels of the leaves, now we will test them.

To illustrate the searching technique we choose a particularly simple definition of goal satisfaction: a label will be a solution if it begins with the sequence `1 0 1`. The value of a satisfying terminal node could be any function of the label, but for simplicity we return the label itself. In the case of non-goal terminals, we return an empty list:

```
Test: <1 0 1> == Pv              % success
      <> == ().                  % failure
```

Notice that with this particular definition, and a search over the powerset tree, a goal state will always be a child of the right-subtree of the root. Therefore depth-first left-to-right searching will

[23] *I.e.*, `A:<path1> == <A:<path2>>.` is equivalent to `A:<path1> == <<path2>>.`

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` picks off the result (up to `!`), which may be an argument list, or, as in this case, a sequence of symbols over `$terminal`. Some theorems of `Powertest` are:

```
Powertest: <a ;> = ()
           <a a ;> = ()
           <a a a ;> = 1 0 1
           <a a a a ;> = 1 0 1 0
           <a a a a a a ;> = 1 0 1 0 0 0.
```

4.3 Polyadic Functions

We noted that functions need not be monadic and that polyadic function are possible. One way for a function to take a variable number of arguments is to use a mechanism for counting the number of arguments in the argument list.[26] Monadic functions only reference arguments which are anticipated, hence the argument list terminator (`!`) is optional. With polyadic functions the `!` terminator is obligatory to mark the end of the argument list. This reduces the problem of determining the number of arguments to counting the number of argument terminators in a single pseudo-argument, the argument list. This is a trivial exercise and a nice application for decimal arithmetic. `Argc` returns the number of arguments in its argument list.

```
Argc:
   <;> == Add:<1 ; <> ;>
   <$terminal> == <>
   <!> == 0
   <> == 'missing ! or invalid symbol'.
```

[24] This definition makes use of *negative path extension* (NPE) (Evans *et al.* 1991, Moser 1992). Briefly, `<if !>` will be matched when the search of the left subtree evaluated to the empty list, while `<if>` will be matched if the search evaluated to any other value.

[25] If `Test` required the missing terminator to be present, we would reconstruct it at the terminal node (case `<;>`) before passing it to `Test`.

[26] Another way to implement polyadic functions is to process the arguments on at a time, chopping them off as they are processed.

A few typical theorems of `Argc` are:

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

`F` is an example of a polyadic function which takes any number of arguments, given in the form `F:<`Arg_1 `;` Arg_2 `; ...` Arg_n `; !>`. Note that the argument list terminator is not optional:

```
F: <> == <case Argc ;>
   <case 0 ;> == '0 args'
   <case 1 ;> == '1 arg:'  Pv:<>
   <case 2 ;> == '2 args:' Pv:<>
   <case> ==   'the' Arg1:<> 'args are:' Pv:<Pop_arg:<> !>. % NPE
```

Some theorems of `F` are:

```
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 ; !> =
                     the 4 args are: 1 2 ; 3 4 ; 5 6 ; 7 8 ;
   <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 ;.
```

The default statement counts the number of arguments, and adds a prefix to the argument list: `<case>`, a *CONTROL* prefix[27] and a new argument (terminated by `;`) indicating the number of arguments in the original path. The first three `<case>` selector labels match precisely the added prefix, so chopping the prefix yields the original path, at which point the appropriate definition can follow. In the NPE case, the prefix inserted by `Argc` corresponds to `Arg1` (after chopping the *CONTROL prefix*), and all of the original arguments have had their argument index incremented.

4.4 Loops

4.4.1 Definite iteration

`For` is a polyadic node which simulates definite iteration (counted loops). We say that `For` simulates iteration because `For:<N ; !>` evaluates to the sequence of values returned by node `Body` invoked on the single argument i, i = 0, 1, ..., `N`-1; *i.e.*,

```
For:<N ; !> = Body:<0 ;> ; Body:<1 ;> ; ... Body:<N-1 ;> ;,
```

The initial call supplies just one argument – the number of iterations `N`.

In the definition of `For` we introduce a new technique of 'naming the arguments'. This is done by giving a 'new name' to the argument extraction primitive, by defining a node whose definition is precisely that of the argument extractor. The definition `N:<> == Arg2`, for example, permits the second argument to be referenced by the name `N`, because they have the same value for any path. The two named arguments here are `N`, the number of iterations, and `I`, the loop counter:

[27]Recall that *CONTROL* symbols are not convertible into values, so cannot be passed as members of the argument list to any argument manipulation primitive.

```
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, the value of the loop is a sequence consisting of the value of `Body:<I:<> ; !>` followed by the value of `For` with `I` incremented and `N` copied over.[30] Any node `Body` taking a single argument is suitable; the following uses addition to compute `X + X`, for a single argument `X`:

```
Body:<> == Add:<Arg1 ; Arg1 ;>.
```

Some theorems using this particular `Body` are:

```
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 ;.
```

Although `Body`, the body of the simulated loop, is invoked on the value of the loop control variable, in theory it could be passed the argument list or some function of the arguments. In the next section we illustrate how to invoke `Body` on successive elements of an argument list of arbitrary length, and in the appendix the computation of Pascal's triangle uses a variation on this theme to pass the body an entire argument list of arbitrary length.

4.4.2 Foreach iteration

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, *i.e.*:

[28] In modern programming methodology, the loop control variable has a scope local to the loop (*i.e.*, should be both undefined and inaccessible outside the loop). In the best case, the loop creates the variable itself with no intervention on the part of the programmer. This is one motivation for introducing `I` in this fashion.

[29] The node `Equal` is defined in Moser (1992). It takes two arguments, and returns `true` is they are equal, `false` otherwise.

[30] Note that the control prefixes `<then>` and `<else>` are chopped, and the constructed argument lists are terminated, to delimit them from the default extension.

```
Foreach:<$x_1$ ; $x_2$ ; ... ; $x_n$ ; !> = Body:<$x_1$> ; Body:<$x_2$> ; ... ; 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 DATR

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

[Evans and Gazdar 1990] Roger Evans and Gerald Gazdar, editors. *The DATR Papers, Volume I.* CSRP 139. School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1990.

[Evans *et al.* 1991] Roger Evans, Gerald Gazdar, and Lionel Moser. Prioritised multiple inheritance in DATR. In Ted Briscoe, Ann Copestake, and Valeria de Paiva, editors, *Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon.* Technical Report 238, The Computer Laboratory, Cambridge University, Cambridge, UK, 1991.

[Gibbon 1990] Daffyd Gibbon. register.dtr. *The DATR Papers, Volume I,* pages 133–134, 1990. Formal DATR example file illustrating binary shift operations.

[Jenkins 1990] Elizabeth A. Jenkins. Enhancements to the Sussex Prolog DATR implementation. *The DATR Papers, Volume I,* pages 41–61, 1990.

[Moser 1992] Lionel Moser. Lexical constraints in DATR. Technical Report CSRP 216, School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1992.

[Sterling and Shapiro 1986] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, London, 1986.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% 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 *datr                                               %
% Related Files:   lib datr; args.dtr                                       %
% Version:         4.00                                                     %
%     Copyright (c) University of Sussex 1991.  All rights reserved.        %
%                                                                           %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

% This file illustrates how a logical equality operator can be generalised
% to the case where the comparands are sequences of symbols over an alphabet
% $terminal. The technique is based on the path-as-argument machinery in
% args.dtr.

% $terminal must have been defined elsewhere.
% #load 'args.dtr'.

% Equal:<Arg1 ; Arg2 ;> == true/false
% Sample theorems:
%     Equal: <1 2 3 ; 1 2 3 ;> = true
%            <1 2 3 ; 1 2 3 4 ;> = false.

Equal:
   <; ;> == true % reached end of both args
   <;> == false   % Arg1 is shorter
              % If Arg2 is shorter, First returns error msg, Tequal fails
   <> == < If:< Tequal:<First First:<Arg2 ;>>> >    % First == First:<Arg1>
   <then> == Equal:<Rest:<Arg1:<> ; > ; Rest:<Arg2:<> ; > ; !>
   <else> == false.

% Tequal:<atom atom>  == true/false
% Terminal Equal
Tequal:  <> == '**** Error: (Tequal) Unknown symbol'
   <$terminal $terminal> == true
   <$terminal> == false.

% If:<condition> == then/else
If: <> == '**** ERROR: (If) Invalid argument'
   <true> == then
   <false> == else.

% -- Some theorems ----------

% Equal: <; ;> = true
%        <1 2 3 ; 1 2 3 ;> = true
%        <1 2 3 ; 1 2 3 4 ;> = false.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% 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
% it is easier to pull off prefixes from the head of a list.
% Arguments in general can be viewed as a stack or queue, which
% can be accessed by (a) pulling from the left side; (b) pushing
% onto the left side; and (c) appending on the right side. We use
% ; as an argument terminator. Argument extractors
% (First, Arg1, Arg2, Arg3, and Rest) expect terminators to be
% present, but always return the argument without them, so they
% are usually replaced.
%   The definition of Add is as follows. Let k = max(n,m).
% Then X + Y is computed as a sequence of digits (C[i]+X[i]+Y[i]):
%
% (X + Y) = (C[k]+X[k]+Y[k]) (C[k-1]+X[k-1]+Y[k-1]) ... (C[0]+X[0]+Y[0])
%
%  where
%      X[i] is the i-th digit of X;
%      Y[i] is the i-th digit of Y;
%      C[i] is the carry from column (i-1),
%      and  C0 = 0.
%
%  eg, (163 + 456) = ((C[2]+1+4) (C[1]+6+5) (C[0]+3+6)) =  619
%  using C[0] = 0, C[1] = 0, and C[2] = 1.
%
% We recursively pass down the carry until the leftmost digit
% of the result is reached, compute this digit, and compute the
% digits one the right as we unrecurse.

 RevAdd:
   <0 ; ; ;> == ()   % X and Y exhausted; leftmost digit is zero.
   <1 ; ; ;> == 1    % X and Y exhausted; leftmost digit is a carry.
   <> == (RevAdd:<Carry:<Arg1 ; First_dig:<Arg2 ;> ; First_dig:<Arg3 ;> ; > ;
                 Rest:<Arg2 ;> ;
                 Rest:<Arg3 ;> ;
                >
          Digit:<Arg1 ; First_dig:<Arg2 ;> ; First_dig:<Arg3 ;> ; >
         ).
```

```
% 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 reserved.        %
%                                                                           %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

% Dadd is for "decimal digit add". It is an addition table for two decimal
% digits plus a carry (0 or 1). This table would be quite small for a
% binary add - but the rest of the machinery would be the same.

% Dadd:<Carry ; Digit ; Digit ;> = (new_carry ; remainder ;).

Dadd:
      % We only store one copy of each sum, so if <c ; a ; b ;> is stored
      % then <c ; b ; a ;> is not stored. So if the sum isn't found, swap
      % a and b and look again.
      <> == < Arg1 ; Arg3 ; Arg2 ;>

      <0 ; 0 ; 0 ;> == (0 ; 0 ;)
      <0 ; 0 ; 1 ;> == (0 ; 1 ;)
      <0 ; 0 ; 2 ;> == (0 ; 2 ;)
      <0 ; 0 ; 3 ;> == (0 ; 3 ;)
      <0 ; 0 ; 4 ;> == (0 ; 4 ;)
      <0 ; 0 ; 5 ;> == (0 ; 5 ;)
      <0 ; 0 ; 6 ;> == (0 ; 6 ;)
      <0 ; 0 ; 7 ;> == (0 ; 7 ;)
      <0 ; 0 ; 8 ;> == (0 ; 8 ;)
      <0 ; 0 ; 9 ;> == (0 ; 9 ;)

      <1 ; 0 ; 0 ;> == (0 ; 0 ;)
      <1 ; 0 ; 1 ;> == (0 ; 2 ;)
      <1 ; 0 ; 2 ;> == (0 ; 3 ;)
      <1 ; 0 ; 3 ;> == (0 ; 4 ;)
      <1 ; 0 ; 4 ;> == (0 ; 5 ;)
      <1 ; 0 ; 5 ;> == (0 ; 6 ;)
      <1 ; 0 ; 6 ;> == (0 ; 7 ;)
      <1 ; 0 ; 7 ;> == (0 ; 8 ;)
      <1 ; 0 ; 8 ;> == (0 ; 9 ;)
      <1 ; 0 ; 9 ;> == (1 ; 0 ;)

      <0 ; 1 ; 1 ;> == (0 ; 2 ;)
      <0 ; 1 ; 2 ;> == (0 ; 3 ;)
      <0 ; 1 ; 3 ;> == (0 ; 4 ;)
      <0 ; 1 ; 4 ;> == (0 ; 5 ;)
      <0 ; 1 ; 5 ;> == (0 ; 6 ;)
      <0 ; 1 ; 6 ;> == (0 ; 7 ;)
      <0 ; 1 ; 7 ;> == (0 ; 8 ;)
      <0 ; 1 ; 8 ;> == (0 ; 9 ;)
      <0 ; 1 ; 9 ;> == (1 ; 0 ;)

      <1 ; 1 ; 1 ;> == (0 ; 3 ;)
      <1 ; 1 ; 2 ;> == (0 ; 4 ;)
      <1 ; 1 ; 3 ;> == (0 ; 5 ;)
      <1 ; 1 ; 4 ;> == (0 ; 6 ;)
      <1 ; 1 ; 5 ;> == (0 ; 7 ;)
      <1 ; 1 ; 6 ;> == (0 ; 8 ;)
      <1 ; 1 ; 7 ;> == (0 ; 9 ;)
      <1 ; 1 ; 8 ;> == (1 ; 0 ;)
      <1 ; 1 ; 9 ;> == (1 ; 1 ;)

      <0 ; 2 ; 2 ;> == (0 ; 4 ;)
      <0 ; 2 ; 3 ;> == (0 ; 5 ;)
      <0 ; 2 ; 4 ;> == (0 ; 6 ;)
      <0 ; 2 ; 5 ;> == (0 ; 7 ;)
      <0 ; 2 ; 6 ;> == (0 ; 8 ;)
      <0 ; 2 ; 7 ;> == (0 ; 9 ;)
      <0 ; 2 ; 8 ;> == (1 ; 0 ;)
      <0 ; 2 ; 9 ;> == (1 ; 1 ;)
```

```
<1 ; 2 ; 2 ;> == (0 ; 5 ;)
<1 ; 2 ; 3 ;> == (0 ; 6 ;)
<1 ; 2 ; 4 ;> == (0 ; 7 ;)
<1 ; 2 ; 5 ;> == (0 ; 8 ;)
<1 ; 2 ; 6 ;> == (0 ; 9 ;)
<1 ; 2 ; 7 ;> == (1 ; 0 ;)
<1 ; 2 ; 8 ;> == (1 ; 1 ;)
<1 ; 2 ; 9 ;> == (1 ; 2 ;)

<0 ; 3 ; 3 ;> == (0 ; 6 ;)
<0 ; 3 ; 4 ;> == (0 ; 7 ;)
<0 ; 3 ; 5 ;> == (0 ; 8 ;)
<0 ; 3 ; 6 ;> == (0 ; 9 ;)
<0 ; 3 ; 7 ;> == (1 ; 0 ;)
<0 ; 3 ; 8 ;> == (1 ; 1 ;)
<0 ; 3 ; 9 ;> == (1 ; 2 ;)

<1 ; 3 ; 3 ;> == (0 ; 7 ;)
<1 ; 3 ; 4 ;> == (0 ; 8 ;)
<1 ; 3 ; 5 ;> == (0 ; 9 ;)
<1 ; 3 ; 6 ;> == (1 ; 0 ;)
<1 ; 3 ; 7 ;> == (1 ; 1 ;)
<1 ; 3 ; 8 ;> == (1 ; 2 ;)
<1 ; 3 ; 9 ;> == (1 ; 3 ;)

<0 ; 4 ; 4 ;> == (0 ; 8 ;)
<0 ; 4 ; 5 ;> == (0 ; 9 ;)
<0 ; 4 ; 6 ;> == (1 ; 0 ;)
<0 ; 4 ; 7 ;> == (1 ; 1 ;)
<0 ; 4 ; 8 ;> == (1 ; 2 ;)
<0 ; 4 ; 9 ;> == (1 ; 3 ;)

<1 ; 4 ; 4 ;> == (0 ; 9 ;)
<1 ; 4 ; 5 ;> == (1 ; 0 ;)
<1 ; 4 ; 6 ;> == (1 ; 1 ;)
<1 ; 4 ; 7 ;> == (1 ; 2 ;)
<1 ; 4 ; 8 ;> == (1 ; 3 ;)
<1 ; 4 ; 9 ;> == (1 ; 4 ;)

<0 ; 5 ; 5 ;> == (1 ; 0 ;)
<0 ; 5 ; 6 ;> == (1 ; 1 ;)
<0 ; 5 ; 7 ;> == (1 ; 2 ;)
<0 ; 5 ; 8 ;> == (1 ; 3 ;)
<0 ; 5 ; 9 ;> == (1 ; 4 ;)

<1 ; 5 ; 5 ;> == (1 ; 1 ;)
<1 ; 5 ; 6 ;> == (1 ; 2 ;)
<1 ; 5 ; 7 ;> == (1 ; 3 ;)
<1 ; 5 ; 8 ;> == (1 ; 4 ;)
<1 ; 5 ; 9 ;> == (1 ; 5 ;)

<0 ; 6 ; 6 ;> == (1 ; 2 ;)
<0 ; 6 ; 7 ;> == (1 ; 3 ;)
<0 ; 6 ; 8 ;> == (1 ; 4 ;)
<0 ; 6 ; 9 ;> == (1 ; 5 ;)

<1 ; 6 ; 6 ;> == (1 ; 3 ;)
<1 ; 6 ; 7 ;> == (1 ; 4 ;)
<1 ; 6 ; 8 ;> == (1 ; 5 ;)
<1 ; 6 ; 9 ;> == (1 ; 6 ;)

<0 ; 7 ; 7 ;> == (1 ; 4 ;)
<0 ; 7 ; 8 ;> == (1 ; 5 ;)
<0 ; 7 ; 9 ;> == (1 ; 6 ;)

<1 ; 7 ; 7 ;> == (1 ; 5 ;)
<1 ; 7 ; 8 ;> == (1 ; 6 ;)
<1 ; 7 ; 9 ;> == (1 ; 7 ;)

<0 ; 8 ; 8 ;> == (1 ; 6 ;)
<0 ; 8 ; 9 ;> == (1 ; 7 ;)

<1 ; 8 ; 8 ;> == (1 ; 7 ;)
<1 ; 8 ; 9 ;> == (1 ; 8 ;)

<0 ; 9 ; 9 ;> == (1 ; 8 ;)

<1 ; 9 ; 9 ;> == (1 ; 9 ;).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% 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.       %
%                                                                           %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

#load 'add10.dtr'.
#load 'args.dtr'.
#load 'arglogic.dtr'.
#load 'polyadic.dtr'.

% 'For' simulates a counted loop. It is polyadic. The initial call supplies
% just one argument - the number of times to loop. On subsequent calls, For
% adds its own loop counter to the argument list.

% For:<N ; !>    initial call to loop N times.
% For:<I ; N ; !>  recursive call while looping
%
% The body of the simulated loop is the node Body. Body is passed the value of
% the loop control variable as an argument; in theory it could be passed the
% argument list or
% something else. For returns the results of Body:<I>, (I = 0,1,...,N-1) as
% an argument list. ie,
% For:<N ; !> == (Body:<0 ;> ; Body:<1 ;> ; ... Body:<N-1 ;> ; !).

% argument names
I:<>==Arg1.
N:<>==Arg2.    % Arg1 on initial call only.

For: <> == <argc Argc ;>
   <argc 1 ;> == % initial call
                 For:<0 ;>
   <argc 2 ;> == % looping: test before iterating.
                 <If:<Equal:<>>>       % pass entire arglist; Equal is
                                         % univalent and only uses 1st two.
   <then> == () % stop
   <else> == (Body:<I:<> ; !> ; For:<Add:<1 ; I:<> ; > ; N:<> ; !>).

% A simple body.
% Body:< X ;> == 2 * X.
Body:<> == Add:<Arg1 ; Arg1 ;>.
```

```
% --------- 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 ; ]).
```

```
Line6:<> ==
     Successor:<Successor:<Successor:<Successor:<Successor:<[ 1 ; ]>>>>>.

% ------- A theorem -----

% Line6: <> = ([ 1 ; 5 ; 1 0 ; 1 0 ; 5 ; 1 ; ]).


% To define a procedure which returns the first N lines of Pascal's
% triangle, we strip down the For loop to just look at the first two
% arguments, and pass an arbitrary number of following args as the
% arg list to Body. What For computes is:
% For:<I ; N ; {Body(0)} ; !> ==   ( Body:<{Body(0)}> ;
%                                    Body:<{Body(1)}> ;
%                                    Body:<{Body(2)}> ;
%                                    ....
%                                    Body:<{Body(N-1)}> ;
%                                  ).
%
% But it is defined recursively:
% For:<I ; N ; PrevBody  ; !> ==
%    if I = N:    ()
%    otherwise:   (Body:<PrevBody> ; For:<I+1 ; N ; Body:<PrevBody>).
%
% Argument names
I:<> == Arg1.
N:<> == Arg2.
PrevBody:<> == Pop_arg:<Pop_arg !>.     % args 3, 4, ..., !

For: <> == <If:<Equal>>       % (I = N?) pass entire arglist;
                              % Equal uses only 1st two.
   <then> == () % stop
   <else> == (Body:<PrevBody:<> !> ;
              For:<Add:<1 ; I:<> ; > ;
                   N:<> ;
                   Body:<PrevBody:<> !> ;
                   !
                  >).

% Wire up Successor as the body.
Body: <> == Successor.

% ---- Some theorems ----
%
%  For: <0 ; 1 ; [ 1 ; ] ;> = ([ 1 ; 1 ; ] ;).
%
%  For: <0 ; 2 ; [ 1 ; ] ;> = ([ 1 ; 1 ; ] ; [ 1 ; 2 ; 1 ; ] ;).
% Make a pretty interface, and insert top line (Body(0)) of the triangle.
% Pascal:<n ;> == first (n+1) lines of Pascal's triangle.

Body0:<> == ([ 1 ; ]).

Pascal:<> == (Body0 ; For:<0 ; Arg1 ; Body0 ; !> ).

% ---- Some theorems ----
%
%
%   Pascal: <0 ;> = (            [ 1 ; ] ;
%                   ).
%
%   Pascal: <1 ;> = (            [ 1 ; ] ;
%                              [ 1 ; 1 ; ] ;
%                   ).
%
%   Pascal: <2 ;> = (            [ 1 ; ] ;
%                              [ 1 ; 1 ; ] ;
%                            [ 1 ; 2 ; 1 ; ] ;
%                   ).
%
%   Pascal: <3 ;> = (            [ 1 ; ] ;
%                              [ 1 ; 1 ; ] ;
%                            [ 1 ; 2 ; 1 ; ] ;
%                          [ 1 ; 3 ; 3 ; 1 ; ] ;
%                   ).
%
%   Pascal: <6 ;> = (            [ 1 ; ] ;
%                              [ 1 ; 1 ; ] ;
%                            [ 1 ; 2 ; 1 ; ] ;
%                          [ 1 ; 3 ; 3 ; 1 ; ] ;
%                        [ 1 ; 4 ; 6 ; 4 ; 1 ; ] ;
%                      [ 1 ; 5 ; 1 0 ; 1 0 ; 5 ; 1 ; ] ;
%                    [ 1 ; 6 ; 1 5 ; 2 0 ; 1 5 ; 6 ; 1 ; ] ;
%                   ).
%
%   Pascal: <7 ;> = (            [ 1 ; ] ;
%                              [ 1 ; 1 ; ] ;
%                            [ 1 ; 2 ; 1 ; ] ;
%                          [ 1 ; 3 ; 3 ; 1 ; ] ;
%                        [ 1 ; 4 ; 6 ; 4 ; 1 ; ] ;
%                      [ 1 ; 5 ; 1 0 ; 1 0 ; 5 ; 1 ; ] ;
%                    [ 1 ; 6 ; 1 5 ; 2 0 ; 1 5 ; 6 ; 1 ; ] ;
%                  [ 1 ; 7 ; 2 1 ; 3 5 ; 3 5 ; 2 1 ; 7 ; 1 ; ] ;
%                   ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% File:           powerset.dtr                                      %
% Purpose:        To illustrate theories which can take exponential  %
%                 time to evaluate if the compiler isn't smart.      %
% Author:         Lionel Moser, August 31, 1991.                     %
% Documentation:  HELP *datr                                         %
% Related Files:  lib datr                                           %
% Version:        3.00                                               %
%      Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The powerset tree of order 2 looks like this:
%          o
%         / \
%       0/   \1
%       /     \
%      o       o
%    0/ \1   0/ \1
%    o   o   o   o
%   00  01   10  11
%
% The left-branching and right-branching edges are labelled with 0 and 1,
% respectively. The nodes (little o's) have labels which are the concatenation
% of all of the edge labels from the root - or, stated recursively, the label
% on a node is the label on its parent plus the label on the edge to its
% parent. The collection of labels of the leaves of this tree we call the
% powerset of 2; they consist of all binary strings of length 2.
%
% In this example I show that the power set of n can be generated, by
% generating all binary strings of length n. There are 2^n such strings.

% In order to generate these strings, (2^n)-1 calls to Powerset are made,
% with no two such calls deriving a value for the same path. (See below.)

% The initial query path is a list of n a's, where n is the length of
% the binary strings to be generated, terminated by ;.
%
%            <-- n a's -->
% Powerset:<a a a a a ... a ;> =
%                      (0 0 ... 0, 0 0 ... 1, 0 0 ... 1 0, ... , 1 1 ... 1 1).
%                        <-------  2^n binary strings of length n -------->
%
%    The technique used is to perform a complete depth-first left-to-right
% traversal of a binary tree, where an "edge label" represents a binary
% digit, which is appended to the digits accumulated from the root to the
% current node. A path at an arbitrary node is viewed as two arguments:
% a sequence of a's, representing the distance to the leaf nodes; and the
% sequence of binary digits labelling the edges from the root to the current
% 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 illustrates how to test values at leaf nodes,  %
%                   stop searching when a solution is found, and pass the    %
%                   solution back as recursion unfolds.                %
% Author:           Lionel Moser, September, 1991.                     %
% Documentation:    HELP *datr                                         %
% Related Files:    lib datr, powerset.dtr                             %
% Version:          3.00                                               %
%     Copyright (c) University of Sussex 1991.  All rights reserved.   %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% powerset.dtr illustrates full traversal of a binary search tree. This file
% illustrates partial traversal of a search tree with backtracking. The search
% tree we use is the powerset tree of order n, where n is the distance from
% the root to the leaves. We consider the labels on the terminal nodes of the
% search tree to be possible solutions to some problem. The algorithm below
% (powertest) performs a depth-first left-to-right traversal of the powerset
% tree. At each terminal node it tests the label as a possible solution: if it
% is, powertest unrecurses back up to the root, passing up the satisfying sol-
% ution. Otherwise it backs up and descends the next right branch. Searching
% thus stops when a satisfying leaf label is found, or the tree is exhausted.

% We illustrate using a simple test for a solution. A label will be a solution
% if it begins with the sequence (1 0 1). The result returned could be any
% function of the label, but for simplicity we return the label itself.

Test: <1 0 1> == Pv             % success
      <> == ().                 % failure

% The number of leaves visited before the solution is found (using this
% particular test) grows exponentially with n because the solution is always
% in the right-branch subtree of the root, and the searching is done
% depth-first left-to-right.
%
% Powertest:< a a a a ... a ;>
Powertest:
    <;> == Test:<>    % leaf node
    <a> == <if  Powertest:<Pv:<> 0 !> ! >    % note that <a> is chopped.
    <if !> == <continue>
    <if> == Pv:<>                            % negative path extension.
    <continue> == Powertest:<Pv:<> 1 !>.
```

```
% 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 ;> = ()
%             <a a ;> = ()
%             <a a a ;> = (1 0 1)
%             <a a a a ;> = (1 0 1 0)
%             <a a a a a a ;> = (1 0 1 0 0 0).
```