ACL

Cancer Research UK

Advanced Computation Laboratory,
Cancer Research UK
44 Lincoln's Inn Fields
London WC2A 3PX
Tel: 020 7269 3627 Fax: 020 7269 3186
.

CREDO-2002-009

FAQ on Tallis Expressions and Assertions

David Sutton

Creation date	06 December 2002
Last modification	06 September 2005
Last edited by	Ayelet Oettinger
Revision	1
Version	3.1
Circulation	unrestricted

1. What if I just want a list of the available operators?

They are listed in Appendix A.

2. What is an expression?

Expressions are text strings that may be evaluated in order to obtain information about the current state of a PRO*forma* guideline during its enactment. Expressions occur in the definitions of tasks, in particular as preconditions and wait conditions. *PROforma* applications may also use expressions to query the current state of a guideline.

An expression has a value that is dependent on the state of the guideline. However evaluating an expression (i.e. calculating its value) never changes the state of a guideline.

3. What is an assertion?

An assertion is a text string describing value(s) to be assigned to data item(s) in a guideline. Assertions occur in the definitions of tasks, in particular as postconditions.

4. What types of values may an expression evaluate to?

An expression may be truth valued (in which case it may be referred to as a *condition*) or it may evaluate to an integer, a real number, a text string, a set of integers, a set of real numbers, or a set of text strings. An expression may also evaluate to the special value `unknown`.

5. What forms may an expression take?

An expression may be either:

a. **An integer:** e.g. 12345.

b. **A real number:** a real number must contain the decimal point and may contain an exponent, preceded by one of the letters `e`, `E`, `d` or `D`[1]

Examples:

```
3.14159
```

```
.445
```

```
45.
```

[1] If *N* and M are numbers then then *N* e *M* means *N* times 10 to the power of *M*. The letters E, D, d are all synonyms for e.

```
46.0e76

46.0d76

46.0E76

46.0D76
```

c. **A quoted text string**: that is to say a string of characters enclosed in double quotes. Example:

```
“this is a quoted string”
```

Double quotes may appear in quoted text strings so long as they are escaped with a backslash, e.g `“call me \”Ishmael\” ”`.

d. **An atom**: i.e. any sequence of characters enclosed in single quotes or any sequence beginning with an alphabetic character and containing only alphanumeric characters and underscores. Examples:

```
this_is_an_atom

so_is23456_this

‘and this is an atom too’
```

Atoms are generally used to refer to data items and tasks by name.

If you want a single quote to appear inside an atom you need to escape it with a backslash e.g. `‘this atom\’s got one’`.

e. **An infix operator applied to two arguments:** an infix operator is one that comes in between its arguments. The general form of expressions of this kind is

Exp1 InfixOp Exp2

Where *Exp1* and *Exp 2* are expressions and *InfixOp* is one of PRO*forma*'s built-in infix operators. Examples:

```
2+2

2+(3+2)

2+’my data item’
```

PRO*forma*'s infix operators are listed in Appendix A.

f. **A prefix operator applied to the correct number of arguments:** a prefix operator is one that precedes its arguments. The general form for expressions of this kind is

PrefixOp(*Arg1*, ... , *ArgN*)

Where *Arg1*, ... , *ArgN* are expressions.

PRO*forma*'s prefix operators are listed in Appendix A.

Examples:

```
not(age = 3)

isknown(age)

netsupport('my decision', mycandidate)
```

g. **A set expression:** that is to say an expression of the form

[*Exp1*, … ,*ExpN*]

The value of this expression is the set containing the values of the expressions *Exp1*, … ,*ExpN*.

The set expression `[]` denotes the empty set.

Examples:

```
["It", "is", "a" , "truth" ]

[1, 2+3, 4-5, 6]

[3.14159]

[]
```

6. What is the value of an atom?

When an atom occurs as an argument to the `netsupport` or `result_of` operators it is interpreted as naming a decision or candidate. See section entitled What are the arguments of the `result_of result_set` and `netsupport` operators? for more details.

Otherwise an atom has a value that is calculated as follows:

- If the atom is the name of a data item then its value is the current value of that data item.

- Otherwise the value of an atom *A* is simply the string *A* with any enclosing quotes removed.

Example: if a guideline contains an integer data item `mydata` whose current value is 2 then the expression

```
mydata
```

evaluates to the integer 2, Whereas the expression

```
‘this is not the name of a data item’
```

Evaluates to the string

```
this is not the name of a data item
```

N.B. We advise you *not* to use atoms to represent text strings in the manner illustrated by the second example above. In such cases it is preferable to use a double quoted text string.

7. I want to put a text string into an expression. Should I enclose it in single quotes, double quotes, or no quotes at all?

Suppose that you wish to construct an expression that tests whether the value of a data item `drug` is equal to the text string `tylex`. There are three different expressions which might fit the bill:

```
drug = tylex

drug = ‘tylex’

drug = “tylex”
```

However these three expressions are equivalent only if there is no data item called `tylex`. If there were such a data item then the first two expressions would compare the value of the data item `drug` with the value of the data item `tylex`, which is not what is intended in this case.

Since it is difficult to remember the names of all the data items in a guideline and impossible to predict the names of data items that might be subsequently added to that guideline, a text string should *always* be enclosed in double quotes. In other words the preferred way of expressing the above condition would be

```
drug = “tylex”
```

8. What are the arguments of the `result_of` `result_set` and `netsupport` operators?

The result_of operator returns a string and takes one argument, which must be an atom and must be the name of a decision. If a single candidate has been committed for the decision then result_of returns the name of that decision as a string. Otherwise it returns `unknown`.

Example:

```
result_of(‘prescribing decision’)
```

The value of the above expression would depend on whether or not a single candidate of the decision `prescribing decision` had been committed to. If it this were the case then the expression would evaluate to the name of that candidate, otherwise it would evaluate to `unknown`.

The `result_set` operator is similar to `result_of` except that it returns a *set* of strings, these being the names of the candidates that have been committed to.

In general you should use `result_of` to test the result of a decision only if that decision's `choice_mode` is single. Otherwise you should use `result_set`.

Example: if `prescribing_dec` has choice mode `multiple` and `allergy_dec` has choice mode `single` and you wished a particular task to become active if the result of `prescribing_dec` includes "penicillin" and the result of `allergy_dec` is equal to yes, then that task might have the precondition:

```
(result_set(prescribing_dec) includes "penicillin")

and (result_of(allergy_dec) = "yes")
```

Note that the `includes` operator is used to test the result of the first decision and the = operator for the second.

The `netsupport` operator takes two arguments, both of which must be atoms.The first argument is the name of a decision, the second is the name of a candidate of that decision. Example:

```
netsupport('prescribing decision', paracetamol)
```

The value of the above expression is determined by evaluating the netsupport function of the `paracetamol` candidate as set out in the definition of the `'prescribing decision'` task.

9. What forms may an assertion take?

An assertion that a data item should have a particular value takes the form:

DataItemName = *Expression*

Where *DataItemName* is an atom which names a data item and *Expression* is an expression whose value will be assigned to the data item. Note that the = operator appearing in the above assertion *assigns* a value to a data item whereas an = operator in an expression *tests* for equality between its right and left hand sides.

Assertions may be combined together using the infix operator `and`.

Here are some examples of assertions:

```
bmi = weight/(height*height)

age = 10 and sex = "male" and name = "Arthur"
```

```
name = ‘first name’ # “ ” # ‘family name’
```

10. Is PRO*forma* case sensisitive?

For the most part no, however there are some exceptions. More specifically:

- Comparisons between text strings are not case sensitive. For instance, the expression

  ```
  “thisstring” = “ThisString”
  ```

  evaluates to true.

- Names of data items, tasks, and candidates are not case sensitive. For instance the expressions

  ```
  netsupport(mydecision, mycandidate)
  ```

  and

  ```
  netsupport(MyDecision, MyCandidate)
  ```

  are treated as identical.

- Names of all prefix operators, apart from `netsupport` and `result_of` are not case sensitive. For instance

  ```
  abs(2-3)
  ```

  and

  ```
  AbS(2-3)
  ```

  are treated as identical.

- The Exceptions are:

  PRO*forma*'s infix operators are case sensitive. For instance

  ```
  item1 InCludes item2
  ```

  would be rejected as syntactically incorrect.

  The prefix operators netsupport, and result_of are also case sensitive (although we do allow Netsupport as a synonym for netsupport). So for instance

  RESULT_OF(mydecision)

  Is syntactically incorrect but

  The reason why some operators are case sensitive and others are not is related to the way PRO*forma*'s syntax is defined - the case sensitive operators are case sensitive because they are defined as reserved words in PRO*forma*'s lexical grammar.

11. Can I add my own operators?

Not yet. However we intend in future versions that PRO*forma*'s set of built-in operators will be extensible by users.

12. Appendix A: PRO*forma*'s built-in operators

This appendix contains short descriptions of PRO*forma*'s built-in infix and prefix operators. Further information on these operators may be found in the PRO*forma* language specification which may be found at http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=212780

12.1 Infix Operators

The following infix operators are built in to PRO*forma*.

Operator	**Description**

#	Text concatenation operator.
	Example: the expression
	`“more ” # “beans”`
	evaluates to the text string
	`more beans`
	If either operand of `#` evaluates to `unknown` then the resulting expression also evaluates to `unknown`.

+,-,*,/	Arithmetic operators on integers and reals. When neither operand evaluates to `unknown` these operators have their standard meanings. If either of the operands of an arithmetic operator evaluates to `unknown` then the resulting expression also evaluates to `unknown`. For example if the data item `mydata` has not yet been assigned a value then the expression `mydata + 2` evaluates to `unknown`.
>, <, <=, =>, >=, =>, =, !=	Comparison operators. The arguments of these operators may be real numbers, integers, or text strings.

	When neither operand evaluates to `unknown` these operators have their standard meanings.
	A real number may be compared with an integer but a text string may only be compared with another text string.
	The operators
	`<=`
	and
	`>=`
	are synonyms as are
	`>=`
	and
	`<=`.
	If either of the operands of a comparison evaluates to `unknown` then the resulting expression evaluates to `false`.
	For example, if the data item `mydata` has not yet been assigned a value then the expression
	`mydata > 2`
	evaluates to `false`.

and, or, AND, OR.	Boolean operators. When neither operand evaluates to `unknown` these operators have their standard meanings. The operands should both be truth valued. If either operand of either of these operators evaluates to `unknown` then the resulting expression evaluates to `false`. `AND`, `OR` are synonyms respectively for `and`, `or`.
includes, include, oneof	Set membership operators. The first operand of `includes` should be of type `setof_integer`, `setof_real`, or `setof_text` and the second operand should respectively be of type `integer`, `real`, or `text`. An expression of the form

	`Exp1 includes Exp2`
	evaluates to `true` if and only if *`Exp1`* evaluates to a set that includes the value of *`Exp2`*, and *`Exp2`* does not evaluate to `unknown`. If either *`Exp1`* or *`Exp2`* evaluates to `unknown` then `Exp1 includes Exp2` evaluates to `false`.
	`include` is a synonym for `includes` and
	`Exp1 oneof Exp2` is equivalent to
	`Exp2 includes Exp1`
	Examples:
	`[1,2,3] includes 1`
	`“sugar” oneof [“sugar”, “spice”]`
	`“tylex” oneof mysetof_drugs`

12.2 Prefix Operators

The following prefix operators are built-in to PRO*forma*:

Operator	**Description**
abs	Returns the absolute value of its argument, which must be an integer or real. Example: `abs(3.4 - 4.5)` evaluates to `1.1`
completed_time, discarded_time, in_progress_time	An expression of the form `completed_time(Exp)` is evaluated as follows • If *`Exp`* evaluates to the name of a task then `completed_time(Exp)` evaluates to the “engine time” at which that task last entered the `completed` state or to `unknown` if the task has not entered that state at any point during the current enactment of the guideline. • If *`Exp`* `does not` evaluate to the name of a task then `completed_time(Exp)` evaluates to

	`unknown`. By default the engine time is the same as the system time of the computer on which the PRO*forma* engine is running. `discarded_time` and `in_progress_time` are defined in an analogous manner except that they respectively return the engine time at which the task last entered the `discarded` or `in_progress` states. Example: If a task *T* has a wait_condition `now – completed_time("T2") > 1000` it will not start until 1 second (i.e. 1000 milliseconds) after the task named "T2" has completed.
count	If *`Exp`* evaluates to a set of integers, real numbers or text strings then `count(`*`Exp`*`)` evaluates to the number of members of that set. Examples: `count[]` evaluates to 0. `count["anno","domini"]` evaluates to 2.
diff	If *`Exp1`* and *`Exp2`* both evaluate to a set of integers, real numbers or text strings then `diff(`*`Exp1`*`,`*`Exp2`*`)` evaluates to the result of evaluating *`Exp1`* and then removing any elements that are also found in the value of *`Exp2`* . Example: `diff([1,2+2,5,6],[5,2,1])` evaluates to `[4,6]` Note that it is possible for *`Exp1`* and/or *`Exp2`* to evaluate to `unknown` or to a list containing `unknown`. The behaviour of the `diff` operator in such cases is described in the PRO*forma* language specification.
forever	The expression `forever()` Always evaluates to false. It is intended to be used as a `cycle_until` condition for tasks that cycle forever.
if	An expression of the form

	if(*Exp*1, *Exp*2, *Exp*3) evaluates to: - the result of evaluating *Exp*2 ,or - the result of evaluating *Exp*3, or - unknown respectively depending on whether *Exp*1 evaluates to true, to false, or to unknown.
intersect	If *Exp1* and *Exp2* both evaluate to a set of integers, real numbers or text strings then intersect(*Exp1*,*Exp2*) evaluates to the result of evaluating *Exp1* and then removing any elements that also appear in the value of *Exp2* . Example: intersect([1,2+2,5],[5,2,1]) evaluates to [1,5] Note that it is possible for *Exp1* and/or *Exp2* to evaluate to unknown or to a list containing unknown. The behaviour of the intersect operator in such cases is described in the PRO*forma* language specification.
is_completed	An expression of the form is_completed(*Exp*) evaluates to: - true if *Exp* evaluates to the name of a task and that task is currently in the completed state. - false if *Exp* evaluates to the name of a task that is not in the completed state. - unknown otherwise.
is_discarded	An expression of the form is_discarded(*Exp*) evaluates to: - true if *Exp* evaluates to the name of a task and that task is currently in the discarded state. - false if *Exp* evaluates to the name of a task that

	is not in the discarded state. - `unknown` otherwise.
is_dormant	An expression of the form `is_dormant(`*`Exp`*`)` evaluates to: - `true` if *`Exp`* evaluates to the name of a task and that task is currently in the `dormant` state. - `false` if *`Exp`* evaluates to the name of a task that is not in the dormant state. - `unknown` otherwise.
is_in_progress	An expression of the form `is_in_progress(`*`Exp`*`)` evaluates to: - `true` if *`Exp`* evaluates to the name of a task and that task is currently in the `in_progress` state. - `false` if *`Exp`* evaluates to the name of a task that is not in the in_progress state. - `unknown` otherwise.
isknown	An expression of the form `isknown(`*`Exp`*`)` evaluates to `true` if *`Exp`* evaluates to `unknown` and to `false` otherwise. The `isknown` operator may be applied to operands of any type.
ln, exp	Natural logarithm and exponent. Arguments must be integers or real numbers. Example: `ln(exp(1))` evaluates to `1.0`
max, min	If *`Exp`* evaluates to a set of integers, real numbers or text strings then `max(`*`Exp`*`)` evaluates to the maximum element in the set and `min(`*`Exp`*`)` evaluates to the minimum element

	in the set. When computing a maximum or minimum, numbers are compared in the usual way Example: `max([1,3,2])` evaluates to `3`. Text strings are compared lexicographically ignoring case. Example: `max(["bb","bbb","AAA"])` evaluates to `"bbb"`.
Netsupport, netsupport	These operators are explained in the section entitled "**Error! Reference source not found.** `Netsupport` is a synonym for `netsupport.`

not	An expression of the form
	`not(`*Exp*`)`
	evaluates to `true` if and only if *Exp* evaluates to `false`. It evaluates to `false` if *Exp* evaluates to `true` or to `unknown`.
	Note that the argument must be enclosed in parentheses, e.g.
	`not (drug = tylex)`
	is syntactically correct but
	`not drug = tylex`
	is incorrect.

now	The expression `now()` evaluates to the current engine time. By default the engine time is the same as the system time of the computer on which the PRO*forma* engine is running.
nth	If *Exp* evaluates to a set of integers, real numbers or text strings then `nth(`*n*`,`*Exp*`)` evaluates to the *n*th element in the set, or to `unknown` if the set does not have an *n*th element.

	Examples: `nth(2,["fee","fi","fo","fum"])` evaluates to `"fi"` `nth(4,[2,3,5])` evaluates to `unknown`.

random	The expression
	`random()`
	evaluates to a pseudo-random number that changes every time the state of the engine changes.
	Note however that two occurrences of `random()` that are evaluated without an intervening change in engine state will return the same value. For instance, if a task has the precondition
	`random() = random()`
	then this precondition will always evaluate to true.
	The reason for this behaviour is that it ensures that the value of any PRO*forma* expression is functionally dependent on the state of the engine at the time at which it is evaluated.

result_of, result_set	These operators are explained in ¶**Error! Reference source not found.**.
sin, cos, tan, asin, acos, atan.	Standard trigonometric operators. Angles are assumed to be defined in radians.
sum	If *Exp* evaluates to a set of integers or real numbers or text strings then `sum(`*Exp*`)` evaluates to the sum of the members of that set. Examples: `sum[]` evaluates to 0. `sum([1,1,2,3,5])` evaluates to 12.
union	If *Exp1* and *Exp2* both evaluate to a set of integers, real numbers or text strings then `union(`*Exp1*`,`*Exp2*`)` evaluates to the result of concatenating the values *Exp1* and *Exp2* . Unlike a set theoretic union the PRO*forma* union operator does not eliminate duplicates. Example:

	`union([1,2+2,5],[5,2,1])` evaluates to `[1,4,5,5,2,1]` Note that it is possible for *`Exp1`* and/or *`Exp2`* to evaluate to `unknown` or to a list containing `unknown`. The behaviour of the `union` operator in such cases is described in the PRO*forma* language specification.