Qulog/TeleoR

version 0.7, 29 February 2020

Peter Robinson (pjr@itee.uq.edu.au)
Keith Clark

This manual is for Qulog/TeleoR (version 0.7, 29 February 2020).

Table of Contents

Qulog

1 Introduction

QuLog is a higher-order logic/functional/string processing language with an imperative rule language sitting on top, defining actions. QuLog's action rules are used to program multi-threaded communicating agent behaviour. Its declarative subset is used for the agent's belief store. The language is flexibly typed and allows a combination of compile time and run-time type checking.

It is a fully integrated language in that function calls can appear as or inside arguments to relation calls, and relational queries can be used as guards of function rules. It has sets as a separate data type from lists with set `<->` list convertors. Both can be created using Trm::Query comprehension expressions.

Sets are manipulated using union, intersection and difference operators. Lists are manipulated as in Prolog but also using non-deterministic pattern matching. Similar pattern matching is used for string processing as a precursor to DCG parsing. An 'in' primitive can be used to access elements of sets, lists and characters in strings.

QuLog supports type safe meta-level programming to complement its type safe higher order programming. However there are no lambda expressions in QuLog. All code has to be named and defined in the top level sequence of type definitions, type declarations and relation, function and action defining rules. At this time QuLog has no module system, so each consulted file must use different type and code names for its definitions.

As mentioned above, using its action rules and action primitives multi-threaded message communicating agent applications can be created with the agents communicating using the companion Pedro publish/subscribe and destination addressed communications server. Such an agent application can also receive and send MQTT notifications routed via an MQTT publish/subscribe server.

Debugging is done by putting a `watch` on any number of relations, functions and actions. This invisibly transforms their code to display each call, the input and output bindings of the unification or match of the call with each rule that can be used, and optiionally the instantiated body of the rule before it is used. An `unwatch` command reverses the code transformation.

This manual assumes familiarity with logic programming and with higher functional programming in a typed language. A tutorial introduction to the QuLog declarative subset is given in,

`doc/tutorial/QuLog.pdf`.

A formal systax using extended BNF grammar rules is given as an Appendix of this manual. `examples/introduction/qlexamples.qlg` is an example QuLog program that can be consulted and queried.

The `teleor` extension of the QuLog interpreter allows program files to be consulted containing TeleoR procedures as well as QuLog rules. This extension includes a generic agent shell that can be launched to execute calls to TeleoR procedures as tasks. It can be configured by including specially named QuLog action procedures and relations in your program file, as explained in `doc/tutorial/toolsRM.pdf`.

To support use of TeleoR robotic agent programs with robots and simulations that use ROS, there is an example Python program in the ROS directory that will act as an interface

between our Pedro inter-agent and inter-process communications server and an invocation of ROS. This program and information on how to modify it for a particular ROS architecture, are in the `examples/ROS` directory of the QuLog distribution.

QuLog has some predefined constructor types. These usually have a name such as `write_type__` and their constructors and atom values often end with underscore, for example the constructor `q_` and the atom `nl_` of the `write_type__` system type. We recommend you do not use a trailing underscore in any of your own type definitions. If you do use a system reserved name the compiler will reject your definition giving an error message.

Every data type of QuLog has a place in a lattice of types. At the bottom of the lattice is the system type `bottom` with one data value `bottom_`, meaning *undefined*. At the top is the system type `term`.

For a particular program the lattice of system and program associated types is finite.

2 Getting Started

This section describes how to set up the required environment variables and briefly describes how to run the interpreter. At the moment it is not possible to generate an executable QuLog application that can be launched independently of the interpreter. This will be possible. However, the intepreter can be used to launch a multi-threaded message communicating application that can be left to its own devices.

2.1 Environment Variables

The root directory of the QuLog tree contains the files `PROFILE_CMDS` that can be used to define the required environment variables.

2.2 Data Areas

Because Qulog is currently implemented in QuProlog it has the same data areas as QuProlog and the sizes of these areas can be modified in the same way as for QuProlog - see the QuProlog manual.

2.3 Using the Qulog Interpreter

`qulog` is the name of the Qulog interpreter. From a `Unix` shell, Qulog is started by typing:

```
qulog
```

or

```
qulog -A name
```

where `name` is a name for this QuLog process that will be registered with a Pedro server running on the same host. You need to use this option if you want to be able to receive and/or send messages to other processes that have similarly registered a different name with this Pedro server.

If the Pedro server is running on a different host identified by domain or IP address `Host`, launch QuLog using

```
qulog -N Host -A name
```

For example

```
qulog -N leo.itee.uq.edu.au  -A keith_agent
```

When the interpreter is ready it will prompt you with

```
| ??
```

At this point, a relation query or an action command followed by a FULLSTOP NEWLINE can be entered. The interpreter will check that the query or command is syntactically and type correct and that modes of use are correct. It will either display an error message or print out a response to the query or command.

A CONTROL-D will exit the interpreter whenever you get the prompt.

CONTROL-C will interrupt an evaluation and allow you to abort the interpreter (enter e in response to the interrupt prompt), or to terminate the current query and any forked action threads (enter r in response to the interrupt prompt), giving you the `| ??` query

prompt again. There are other response options, displayed if you enter ? in response to the interrupt prompt.

If you enter a relation query then either 'no' will be displayed to indicate there are no solutions to the query or bindings for variables of the query with their minimal types will be displayed separated by lines of fullstops. If you entered a command any output from the command will be displayed followed by 'success' or 'fail' depending upon whether the command ultimately succeeded or failed.

When there are multiple solutions to a relation query the first five (if there are that many) are displayed separated by lines containing ...

Example:

```
| ?? X in [4,0,3,4].

X = 4 : nat
...
X = 0 : nat
...
X = 3 : nat
...
X = 4 : nat

| ??  % New prompt indicates all sols have been given
```

If there are five or more solutions the interpreter waits for input from the user before displaying more. The two usual responses are:

NEWLINE - no more solutions are required; or

..NEWLINE - asking for up to 5 more solutions.

Example, showing a second use of 'in':

```
| ?? S in "Apple".

S = "A" : string
...
S = "p" : string
...
S = "p" : string
...
S = "l" : string
...
S = "e" : string
..      % Request for more answers if there are any
no more solutions
% Above displayed only after .. input and there are no more answers

| ?? X in {6,2,3,0,3,7,4}.
```

```
% {6,2,3,0,3,7,4} is a set so second 3 ignored, third use of 'in'

X = 0 : nat    % Answers displayed in value order
...
X = 2 : nat
...
X = 3 : nat
...
X = 4 : nat
...
X = 6 : nat
..        % Request to display up to 5 more answers
X = 7 : nat

| ??      % Prompt for next query indicating no more answers
```

If you feel that the interpreter is giving back too many, or too few answers for a particular problem you can control this in two ways. The first is to prefix the query by the number of solutions you would like displayed at a time, followed by a `of`, followed by the query. Also, instead of simply using a `..` to ask for more solutions you can change the number of solutions to be displayed for this query to positive integer `k` by entering `..k`.

Example:

```
| ?? 1 of X in [1,2,1,4,2].  % Answers 1 at a time

X = 1 : nat
.. 2                % Switch to sols 2 at a time
X = 2 : nat
...
X = 1 : nat
..                  % Request for the next 2 sols
X = 4 : nat
...
X = 2 : nat
..                  % Request for the next 2 sols
no more solutions
| ?
```

You can also change the default number of solutions that are displayed for any query to a positive number `n`, say 3, using the command:

```
| ?? set_num_answers 3.

success
```

Sometimes you might have a relation query that contains many variables but you might only be interested in the bindings for some of the variables. This can be accomplished by listing the variables for which you want to see the answer bindings, separated from the query by a `?`.

Example:

```
| ?? L1, L2 :: append(L1, L3, [1,2,3,4,5,6]) & append(L4, L2, L3)
     & 2 = #L4.
     % Expressions such as  #L4, length of L4, can be used in = tests
L1 = [] : list(Ty1)
     % A type variable Ty1 as [] is empty list of any type
L2 = [3, 4, 5, 6] : list(nat)
     % (nat) is type expression for a list of nats (non-neg ints)
...
L1 = [1] : list(nat)
L2 = [4, 5, 6] : list(nat)
...
L1 = [1, 2] : list(nat)
L2 = [5, 6] : list(nat)
...
L1 = [1, 2, 3] : list(nat)
L2 = [6] : list(nat)
...
L1 = [1, 2, 3, 4] : list(nat)
L2 = [] : list(Ty1)
```

The two ideas above can be combined as in the following example.

```
| ?? 2 of L1, L2 :: append(L1, L3, [1,2,3,4,5,6]) & append(L4, L2, L3) &
     2 = #L4.

L1 = [] : list(Ty1)
L2 = [3, 4, 5, 6] : list(nat)
...
L1 = [1] : [nat]
L2 = [4, 5, 6] : list(nat)
```

Equivalently you can express the above query using an existential quantification on `L3, L3`.

```
| ?? 2 of exists L3, L4 append(L1, L3, [1,2,3,4,5,6]) &
        append(L4, L2, L3) & 2 = #L4.
```

If you want to evaluate an expression in the interpreter, one way of doing this is to simply create a unification involving a variable and the expression of interest. Remember that expressions are evaluated before unification.

Example:

```
| ?? X = 2+sin(pi()/4).

X = 2.70711 : num

| ?? X = cos.

X = cos : atom, (num) -> num
```

The second expression is just the name of a primitive function and the value is that function. Note that `cos` is both an atom and the name of a function.

Instead of getting each solution of a query displayed (as in the previous append example) you can use a list comprehension expression with unwanted variables existentially quantified as below.

```
| ?? Lst = [(L1,L2) :: exists L3, L4 (append(L1, L3, [1,2,3,4,5,6]) &
           append(L4, L2, L3) & 2 = #L4)].

Lst = [([], [3, 4, 5, 6]), ([1], [4, 5, 6]), ([1, 2], [5, 6]),
           ([1, 2, 3], [6]),
           ([1, 2, 3, 4], [])] : list((list(nat), list(nat)))


% Value type is a list of pairs of lists of nats
```

We can re-express the last list expression query more succinctly using the list concatenation operator `<>` for splitting of a list using the special non-determinsitic match operator `=?` that requires its left hand side to be, or to evaluate to a ground term. `<>` may also be used for concatenating complete lists or ground or non-ground terms.

```
| ?? Lst = [(L1,L2) :: exists L4 [1,2,3,4,5,6] =? L1 <> L4::(2=#L4) <> L2].
```

Using this non-deterministic list pattern matching we do not need the `L3` variable, and the constraint that `L4` must contain two elements becomes a constraint

```
2=#L4
```

expressed inside the `<>` pattern expression attached to the variable `L4`, preceded by a `::`.

If you have constructed a program file `prog1.qlg` of QuLog type definitions, type declarations for relations, function and actions and their rules, you can bring all those into the interpreter using the command

```
| ?? consult prog1.

Consulting prog1...
... prog1 consulted
success
```

You may get syntax and mode errors signalled in which case none of the program file is consulted. There will be at least one QuLog examples file with the QuLog distribution that you installed. You can consult and query one of these files. For example, there may be a file `qlexamples.qlg` in `qulog/examples/introduction`. If you launch QuLog from inside this directory you can load all its definitions using:

```
| ?? consult qlexamples.
```

You can see all the relation and function rules you currently have in the interpreter using:

```
| ?? show.
```

or specific ones using:

```
| ?? show child_of, person, fact, new_child.
```

Notice the variable names of the consulted file will be used.
You can see all the type definitions and declarations using:

```
| ?? types.
```

You can see all the system type definitions and the type declarations for the primitive relations, functions and actions using:

```
| ?? stypes.
```

A displayed type declaration may be accompanied by a brief description of the primitive. You can also show the type declarations for specific relations by giving their names, separated by commas, after either the `types` or `stypes` command.

It's also possible to consult files from within a program (as in Prolog) using a line in the program like

```
?- consult prog1
?- consult prog2
```

Constraints on using consult

For this discussion we say the file consulted at the interpreter level is the *primary* file and any files consulted in the primary file or any file consulted from within those programs are *secondary* files.

Whenever programs are consulted, type and mode checking is applied and if, at the interpreter level, a file is re-consulted then we need to check that changes to programs have not introduced type and mode errors. In principle, this means we should remove all user definitions and declarations (both from the primary and secondary files) and do a fresh consult of the primary file.

Typically, when a user is debugging a programming, only the primary file will be edited by the user and so it would be inefficient to remove all definitions and declarations form the secondary files and start again. In order to make this process more efficient we constrain the use of consult in two ways in order to make this easier. The first is that, in any interpreter session, the user can only consult one file with re-consulting that file allowed (i.e. only one primary consult). The second is that all consults appearing in a program file are at the start of the file (before declarations and definitions).

When any file is consulted, a fact containg the file name and a timestamp (last update time) is remembered. When the primary file is reconsulted the first step is to check if any secondary files has a more recent last updated time than the remembered fact (meaning the file has been updated). In this case we take the conservative approach and clear out all user definitions and declarations and do a "fresh consult".

If no secondary files have changed and the primary file has also not changed then the consult does nothing as nothing has changed.

Otherwise, all the definitions and declarations of the primary file are removed and the file is re-consulted. If it turns out that one of the changes to the primary file is to change what files are consulted (including the order of consults) then this could lead to potential type mode problems and so, again, we take a conservative approach and remove all user definitions and declarations and do a fresh consult.

By doing the above, we believe that there is no significant impact on the way users write and debug programs. If fact, the most typical debugging approach of editing the definitions and declarations of only the primary file will be quite efficient as only the old definitions and declarations for the primary file will be removed and the new code consulted (without re-consulting the secondary files).

2.4 Using the TeleoR Interpreter

This is invoked using the command `teleor` with exactly the same option flags as the QuLog interpreter. Its queries and commands are an extension of QuLog interpreter queries and commands. Its query prompt is `| ?~>`.

Using the `teleor` interpreter files can be consulted containing TeleoR procedure definitions. It also supports extra action primitives `start_agent` and `start_task` to launch a generic robotic agent shell interacting with a robot interface process or simulator, and to fork task threads within the agent executing TeleoR procedure calls.

If Pedro is being used for communication then the `teleor` interpreter needs to be called using the `-A` switch in order to connect to Pedro and give a name to the agent.

3 Syntax

This section informally describes the concrete syntax of Qulog. There is a formal extended BNF syntax in Appendix A [EBNF Grammar for Qulog], page 47.

The basic building block is an *expression*. An expression is a: *data constant* (aka *atomic* value), *variable*, *compound term*, *list*, *set*, *function call*, *list comprehension*, *set comprehension*.

We define each of these categories below. The last three are evaluable expressions.

A reader unfamiliar with logic programming might find it odd that a *variable* is considered a data value. However in both QuLog and Prolog variables are first class values and can be passed between calls and embedded in lists and other compound terms, but not in sets. An answer to a query that contains variables are all the instantiations of that answer where the variable is replaced by any value of its type. The ability to pass around terms that are or which contain variables is a powerful programming feature of QuLog and Prolog. It is not a feature of Datalog or Answer Set Logic Programming.

A term is a: *data constant*, *variable*, a *simple compound term* (see below) all arguments of which are terms, a *list of terms*, a *set* (which can only contain ground terms).

A *ground term* is a: *data constant*, a list or *simple compound term* term containing no variables, a *set*.

QuLog function call evaluation is strict. A function call argument is completely evaluated just before the function call in which it appears is evaluated. Functions always return ground term values.

A `ground expression` is an expression that contains no variables, or is such that all its variables are bound to ground values at the point that the expression is evaluated. All function calls must be ground at the time they are evaluated.

3.1 Data constants - the atomic term values

These are atoms (`atom` type), natural numbers (`nat` type), integers (`int` type), floating point numbers (`num` type) and strings (`string` type).

QuLog strings are not lists of byte codes as in Prolog. They are packed sequences of byte codes as in Python an are stored on the heap and are garbage collected when no longer referenced. Identity of strings is determined by character by character matching if they have the same length. Manipulation of strings - concatenation and sub-string extraction involves copying but is quite fast.

As in Prolog, QuLog atoms are stored in an atom table and are replaced by a pointer to its entry in the atom table. Identity of atoms is then identity of atoms table address, there is no character by character matching at runtime. Atoms are a suitable alternative to strings for character sequence values that will not be manipulated and are not transient values. For example use them for names of things in facts. The atom table entries are not garbage collected.

All data contants are sub-types of the system type `atomic`.

`nat` is a sub-type of `int`, which is a sub-type of `num`

3.1.1 Atoms

There are four syntactic forms for atoms.

1. A lower case letter followed by any sequence consisting of "_" and alphanumeric characters.
   For example:

   ```
   percy_smith_2
   semester_1
   ```

2. Any combination of the following set of graphic characters.

   ```
   |-/+*<=>#@$\^&~:.?
   ```

   For example:

   ```
   @<=
   ```

3. Any sequence of characters enclosed by "'" (single quote). Single quote can be included in the sequence by writing the quote twice. "\" indicates an escape sequence, where the escape characters are case insensitive. The possible escape characters are:

newline	Meaning: Continuation
^	Meaning: Same as d.
^character	Meaning: Control character.
dd	Meaning: A two digit octal number.
a	Meaning: Alarm (ASCII = 7).
b	Meaning: Backspace (ASCII = 8).
c	Meaning: Continuation.
d	Meaning: Delete (ASCII = 127).
e	Meaning: Escape (ASCII = 27).
f	Meaning: Formfeed (ASCII = 12).
n	Meaning: Newline (ASCII = 10).
odd	Meaning: A two digits octal number.
r	Meaning: Return (ASCII = 13)
s	Meaning: Space (ASCII = 32).
t	Meaning: Horizontal tab (ASCII = 9).
v	Meaning: Vertical tab (ASCII = 11).
xdd	Meaning: A two digit hexadecimal number.

   Here are a few examples of quoted atoms.

   ```
   'hi!'
   'they''re'
   '\n'
   ```

3.1.2 Numbers

The available range of integers is -(2^31-1) to 2^31-1 on a 32 bit machine and -(2^63-1) to 2^63-1 on a 64 bit machine. Integers can be represented in any of the following ways.

1. Any sequence of numeric characters. This method denotes the number in decimal, or base 10.

   For example:

   ```
   123
   ```

2. `Base'Number`, where `Base` ranges from 2 to 36 and `Number` can have any sequence of alphanumeric characters. Both upper and lower case alphabetic characters in `Number` are used to represent the appropriate digit when `Base` is greater than 10.
   For example, integer value 10 can be written as:

   ```
   2'1010
   16'A
   16'a
   ```

3. Binary numbers can also be represented in the form `0b` followed by binary digits. Similarly octal and hexadecimal numbers can be represented by `0o` or `0x` followed by digits.
   For example

   ```
   0b1011
   0o3170
   0x3afd
   ```

4. 0'`Character` gives the character code of `Character`.
   For example,

   ```
   0'A
   ```

   gives the ASCII character code 65.

A natural number is a non-negative integer.

`num` type numbers include double precision floating point numbers. They are represented using either a decimal point or scientific e notation. Examples:

```
27.8
1.896e4
```

3.1.3 Strings

Any sequence of characters enclosed by '"'

is considered as a string.

Note: Strings in Qulog are the same as Python strings and NOT Prolog strings.

Example:

```
| ?? Str = "Hello" ++ " " ++ "there".

Str = "Hello there" : string
```

3.2 Variables

Variables are available in three syntactic forms.

1. An upper case letter followed by any sequence consisting of `"_"` and alphanumeric characters.
   For example:

   ```
   List_nums  Head
   ```

2. `"_"`, followed by an upper case letter, and then any sequence consisting of `"_"` and alphanumeric characters.
   For example:

```
_Dictionary   _X_1
```

3. `"_"` alone denotes an anonymous variable. Repeated occurences of underscore in a query or rule denote different unnamed variables.

Variables beginning with an underscore should be used when there is just a single occurence of a variable in a rule. It suppresses the "single occurrence of variable" warning which is given otherwise, which is useful for picking up mis-typed variable names.

3.3 Compound Terms

A *simple compound term* is composed of an atom of the first two forms (an alphanumeric or graphic atom), called the functor, immediately followed (no spaces) by a sequence of zero or more expressions separated by commas, enclosed in a pair of `"("..")"` parenthesis. For example:

```
data(jack, 35)
tr(emp(),X/9,tr(L,7,R))
$$(5)
```

Simple compound terms are typically instances of a structured data type declared in the program where the functor is a constructor for the type. If not, the compound term has default type `term`, and a warning that it is not a constructor of a defined type is issued in case there has been a spelling error.

A *compound term* is a simple compound term, or a compound term immediately followed by a sequence of zero or more expressions separated by commas, enclosed in a pair of `"("..")"` parenthesis. For example:

```
curry(*)(4)
curryR(child_of)(mary)(P)
```

Compound terms that are not simple determine the functor of the compound term by a function call which is itself a compound term.

Note that, in Qulog, `a()` is a zero arity compound term that is different from the atom `a`.

3.4 Function calls

A *function call* is a either simple compound term where the functor is the name of a primitive or program defined function, or it is a non-simple compound term where the compound term that denotes the functor is a function call that returns a function value.

For certain binary primitive functions the functor name may be used as an infix operator and placed between the two arguments. This holds for the usual binary arithmetic operators `+`, `*` etc. for which function applications are written as expressions such as `6+9*X`.

The special zero argument functions `e` and `pi`, invoked as in expressions `e()` and `pi()`, evaluate to the numbers 'e' and 'pi'. More details are given in Section 4.4 [Arithmetic], page 36.

In the QuLog interpreter a function call, indeed any expression, can be given as a unification problem to be evaluated.

Examples:

```
| ?? X = 67.7/2.3.

X = 29.4348 : num

| ?? X = curryR(child_of)(tom).

X = curryR(child_of)(tom) : rel(?human)

% The denoted value is a relation over humans.
```

Function calls denote expressions that contain no function calls. That is they denote non-variable terms: atomic values, code names, simple compound terms all the arguments of which are non-variable terms, lists or sets of non-variable terms. The exceptions are certain code returning function calls which are only evaluated when the code value they denote is itself called. The above `curryR(child_of)(tom)` is an example. It denotes an unary relation but that relation is only used when the unary relation is called in a query such as:

```
| ?? P :: Rel=curryR(child_of)(tom) & Rel(P).

P = roger : man
```

3.5 Lists

A list is a comma separated sequence of expressions enclosed in "[".."]" brackets. This is a complete list. Or it is a comma separated sequence of terms ending with `,..` optionally followed by a variable, or ending with `|`) always followed by a variable, enclosed in "[".."]" brackets.

Example:

```
[3,2.7,X*Y,"hello"]
[3,4,..Tail]
[Head,..Tail]
[Head,..]    % shorthand for [Head,.. _]  with _ the anonymous variable
[Head|Tail]
```

The first example is a complete enumeration and the remaining examples are list patterns in which both `,..` and `|` can be read as "followed by". The fourth example is equivalent to `[Head,.. _]`. The last example is using the Prolog syntax for a list.

3.6 List comprehension expressions

Lists of ground terms can also be denoted by a list compehension expression. Examples are:

```
[X**2::X in L & not X in [0,1]]
% Squares of numbers other than 0,1 in nums list L
```

```
[C :: exists A (child_of(C,tom) & age_of(C,A) & A<18)]
% Non-adult children of tom in order found
```

The general form of a list comprehension is:

```
[Expression :: exists VarSequence Condition]
```

where the `exists VarSequence` is optional.

There are constraints on the variables that can be used in such a comprehension. Each variable in `Expression` must either appear in `Condition` and be such that it will be given a ground value by some call in the conjunction, or it must appear before the comprehension expression in a query or rule and will have been given a ground value. Every variable in `Condition` must either be underscore, appear in `Expression` or in `VarSequence`, or must appear before the comprehension expression in a query or rule and will have been given a ground value. This ensures that the value of a list comprehension is always a list of ground terms.

`VarSequence` is a single variable or a comma separated sequence of variables such as `X,Y,Z`

The syntax for `Condition` is given below.

3.7 Sets

A set is a comma separated sequence of ground expressions surrounded with `{` and `}` braces. If there are any duplcate ground terms when all the expressions have been evaluated all but one of the duplicates will be removed. If returned as the value of a expression entry to the interpreter, or as a binding of a variable in a relation query, it will be diplayed with its elements in term order as determined by the `@` primitive.

Example set expression entry:

```
| ?? X = {4,3,1,5-2,1}.

X = {1, 3, 4} : set(nat)
```

3.8 Set comprehension expressions

`set(nat)` is the type expression for a set of natural numbers. `list(nat)` is the type expression for a list of natural numbers.

Sets can also be denoted by a set compehension expression. Examples are:

```
{X**2::X in L & not X in [0,1]}

% Squares of numbers other than 0,1 in  L, duplicates removed.

{A :: exists C, P (child_of(C,P) & age_of(C,A) & A<18)}
```

```
% Set of all the ages of recorded children
```

The general form of a set comprehension is:

```
{Expression :: exists VarSequence Condition}
```

where the `exists VarSequence` is optional.

The constraints on the variables that can be used in a set comprehension are the same as those for a list comprehnesion expression.

`VarSequence` and `Condition` are as for list comprehensions.

3.9 Programs

A Qulog program comprises a sequence of:

- type definitions,
- type macros,
- type declarations,
- relation rules (aka clauses),
- action rules, and
- function rules.

A TeleoR program also includes TeleoR procedures.

They may appear in any order except that all the rules for a particular relation, action or function must be contiguous. A type declaration for a relation, action, function or procedure does not need to be given immediately before its code. The rules of a TeleoR procedure are all included inside `{...}` braces following the procedure head.

An important constraint is that each type definition, type declaration, relation, action and function rule *must* begin at the left end of a new line. If one needs to be continued over more than one line all but the first line must be indented from the left end by at least one space or tab. Fullstop terminators *may* be given at the end of each definition, declaration or rule but is not needed. It is the text starting at the extreme left end of a line after one or more newlines that terminates the previous program statement.

TeleoR procedures must also begin at the left end of a line but inside the `{...}` there are more relaxed layout constraints. Each TeleoR rule can start anywhere on a new line. The parser can use the rule syntax to determine that it is the start of a new rule. Again fullstop terminators may be given at the end of each rule but are ignored.

3.9.1 Type Definitions

A type definition is of one of the forms (with the second being a type macro)

`def` *type-name* `::=` *type-expression*

`def` *type-name* `==` *type-expression*

type-name is either an alphanumeric atom or a compound term whose arguments are distinct variables (each representing any type). A type definition with such a type name defines a parameterised type where the type variables stand for any type. Those type variables then appears in one or more of a disjunction of compound terms with other arguments that are type names. We give examples below.

Examples are given below.

3.9.1.1 Integer range type expression

This is an expression of the form `M..N` where `M < N` and both are integers.

Examples:

```
def digit ::= 0..9
def small_int ::= -10..10
```

As in the examples different range types may overlap but only when one is completely contained inside the other. To have overlapping sets of integers corresponding to different types, type union must be used (see below).

3.9.1.2 Enumeration of constants type expression

This is an expression of the form `C1 | C2 | ... | Ck` where each `Ci` is the same kind of constant, except that we can mix different types of numbers.

Examples:

```
def gender ::= male | female
def threeNums ::= 20 | 6.7 | -50
def article ::= "a" | "an" | "one" | "the" | "that" | "those"
```

Different type definitions using overlapping disjunctions of constants are allowed providing one is completely contained inside the other. So, as well as the `article` type we could define

```
def indef_article::= "a" | "an" | "one"
```

It is possible to have an enumerated type with just one element as in

```
def def_article::= "the"
```

A disjunction of integers can also overlap with a range type providing it either comprises a subset or a superset of the integers of the range type. These constraints ensure that each constant belongs to a unique minimal type. For example `"a"` would belong to the types `indef_artcle`, `article`, `string`, `atomic`, `term`, `top`.

To have partially overlapping disjunctions of constants corresponding to different types, type union expressions must be used to define each partially overlapping type (see below).

3.9.1.3 Parameterised type expression

This is an expression of the form `CT1 | CT2 | ... | CTk` where each `CTi` is a compound term with arguments that are type names, or a single type variable `T`, or a parameterised type name with argument the same type variable `T`. Such a type expression can only appear as the right hand side of a parameterised type definition with left hand side a compound term containing the type variable `T`.

Examples:

```
def tree(T) ::= empty() | tr(tree(T),T,tree(T))
def an_indexed(T)::= rec(int,T)
```

3.9.1.4 Type union expression

This is an expression of the form `Ty1 || Ty2 || ... || Tyk` where each `Tyi` a simple type name or a ground parameterised type name or a code type expression.

Examples:

```
def int_atom == int || atom
```

3.9.1.5 Code type expressions

There are four code type expressions in `Qulog/TeleoR`. These are: a function type, a relation type, and action type and a `TeleoR` procedure type.

3.9.1.6 Function type expression

This has the form `(TE1,TE2,...,TEk) -> TE` where each `TEi` and `TE` is any simple, or compound type name, or type union expression, or a code type expression. Functions must be called by giving ground arguments of the required types and will return a ground value of the specified value type.

Examples:

```
(set(T), set(T)) -> set(T)
(string) -> nat
```

The first example declares the type of a function that takes a pair of sets of some type and returns a set of the same type. The function `union` has this type.

The second example takes a `string` and returns a `nat`. The function `#` (when applied to a string) has this type.

3.9.1.7 Relation type expression

This has the form `rel(MTE1,MTE2,...,MTEk)` where each `MTEi` is a moded type where the type is any simple, or compound type name, or type union expression, or a code type expression.

The possible modes of a moded type are the prefixes `!`, `?` `??` and `@`.

The moded type `!`*Type* used as an argument of a relation means, when called, the supplied argument must be ground and of type *Type*.

The moded type `?`*Type* used as an argument of a relation means, when called, the supplied argument may be variable or a partial or ground and of type *Type*. If not ground in the call it will be ground to a term of type *Type* by the call.

The moded type `??`*Type* used as an argument of a relation means, when called, the supplied argument may be variable or a partial or ground and of type *Type*. It may not be ground on success of the call.

The moded type `@`*Type* used as an argument of a relation means, when called, the supplied argument may be variable or a partial or ground and of type *Type*. It will be left unchanged by the call.

If no mode is given it is taken to be `!`.

Modes can be used multiple times in structured types as long as inner modes are more liberal than outer modes. For example, the moded type

```
!list(?int)
```

means that the top-level list structure must be given (i.e. the number or elements are known at call time) but the elements of the list can be a mixture of integers and variables with the variables instantiated to integers by the call.

Examples:

```
rel(!list(T), !list(T), ?list(T))
rel(!string, ??term)
```

which can also be be given as

```
rel(list(T), list(T), ?list(T))
rel(string, ??term)
```

as `!` is the default mode for static relation arguments.

The first example is the type of a relation whose first two arguments must be given as ground lists of values of the same type `T` and whose third argument can either be a variable or partial or ground list of values of type `T` that will be a ground list of `T` values if the relation call succeeds. This is one of the types of the `append` primitive. The second example is the type of a relation that takes a string as its first argument and whose second argument might be instantiated to a possibly non-ground term. This is the type of the `string2term` primitive.

3.9.1.8 Action type expression

This has the form `act(MTE1,MTE2,...,MTEk)` where each `MTEi` is a moded type where the type is any simple, or compound type name, or type union expression, or a code type expression.

The modes are as for a relation type.

Examples:

```
act(??term, !handle)
act()
```

The first example is the type of an action that takes a term that may be a variable or contains variables and a ground handle. It is the type of the message send action `to`. The second example is the type of an action that takes no arguments. It is the type of `kill_agent`.

3.9.1.9 TeleoR procedure type expression

This has the form `tel(TE1,TE2,...,TEk)` where each `TEi` is any simple, or compound type name, or type union expression, or a code type expression. The types are implicitly `!` ground input moded.

Example:

```
tel(list(block), slot)
```

This is the type of a TR procedure that takes a list of blocks and a slot. This is the type of `makeTower` in one of the tower building programs in the examples directory where `block` and `slot` are user defined types.

3.9.2 Type Declarations

All functions, relations, actions and TeleoR procedures have type declarations of the forms

`fun` *Name*`(`*TypeTuple*`)->`*Type*

`rel` *Name*`(`*ModedTypeTuple*`)`

`act` *Name*`(`*ModedTypeTuple*`)`

`tel` *Name*`(`*TypeTuple*`)`

As an example, below is the type declaration for the builtin `append` relation (with the same semantics as the standard Prolog append relation).

```
rel append(!list(T), !list(T), ?list(T)),
    append(?list(T), ?list(T), !list(T)),
    append(!list(??T), !list(??T), ?list(??T)),
    append(?list(??T), ?list(??T), !list(??T)),
    append(??list(T), ??list(T), ??list(T))
```

The first type of `append` says that, if the first two arguments of the call on `append` are ground lists of a given type, then the third argument will be a ground list of the same type on exit from the call.

The second type says that, if the third argument is a ground list of a given type, then the first and second arguments will be ground lists of the same type on exit from the call.

The third type of `append` says that, if the first two arguments are lists of a known length (i.e. do not have a variable tail) but possibly containing non-ground elements, then the third argument will have a known length on exit from the call but that variables occurring in any of the arguments need not be ground.

The fourth type is the "append driven backwards" version of the third type.

The fifth type is the most general allowing variable length lists in all arguments. In this situation, nothing can be said about the modes on exit from the call.

Note that when we say, for example, the first two arguments are of the same type we mean that the type inference system can find a suitable type as in the example interpreter query below.

```
| ?? append([1,2], [a,b], X).

X = [1, 2, a, b] : list(atom || nat)
```

Here, the suitable (minimal) type for `T` is the union of two types.

Dynamic relations are fact defined relations that can be updated by actions. They are declared as follows.

`dyn` *NameTypeTuple*

The declaration

```
dyn age_is(human,age)
```

is essentially the declaration

```
rel age_is(?human, ?age)
```

together with an implicit declaration that `age_of` is defined only by facts that are action updatable.

Dynamic relations are updated by primitive actions `remember, forget` etc.

Global variables are used to store either integer or number values and are declared as follows.

`int` *Name* `:=` *IntValue*

or

`num` *Name* `:=` *NumValue*

The declaration

```
int count := 0
```

is like a combination of the declaration

```
dyn count(int)
```

and the definition

```
count(0)
```

with the implicit restriction that the `count` dynamic relation is always defined with exactly one fact.

Global variables are updated using primitive actions `:=, +:=, -:=`.

3.9.3 Complex Conjunctions

An interpreter relation query is a *Complex Conjunction*.

This has the form:

Cond1 `&` *Cond2* `& ... &` *Condn*, `n>=1` where each *condi* is one of:

- a *predication* *RelExp*`(`*Exp1*`,...,`*Expk*), `k>=0`, where each *Expki* is an expression - a term that may contain function calls and *RelExp* is an expression returning a k-ary relation *rel'* such that the values of the argument expressions will satisfy the mode and type constraints of *rel'* when it is called. Certain binary primitive relations, for example `=,=?,in`, can be written with tghe relation name between the arguments.
- a `not (exists` *VarSeq SConj* `)` *negated condition*, where *VarSeq* is a sequence of local variables of *SConj* which is a *Simple Conjunction*. This is the QuLog negation-as-failure operator.
- a `call Var`, `Var` a variable, *relation meta-call*.
- a predication, or a `(..)` bracketed *Simple Conjunction*, prefixed with `once` - find one solution only
- a universally quantified implication of the form

  `forall` *UVars* `(exists` *EVars1 SConj1* `=> exists` *EVars2 SConj2* `)`

  The *UVars* variables are *universally* quantified over the implication. The sequence of variables of *EVarsi* are existentially quantified over *SConji*, `i=1,2`. Each *SConji* is a *Simple Conjunction*.

A *Simple Conjunction* is a *Complex Conjunction* which does not contain any `forall`

In a negated condition the existentially quantified variables are local variables of *SConj*, variables that do not appear elsewhere in the query or rule body in which the negation appears. All other variable of *SimpleConj*, except underscore anonymous variables, must be given ground values before the condition is evaluated. Underscore variables are always implicitly existentially quantified just before the predication in which they appear. If there are no such local variables the `exists` *VarSeq* is absent.

The type of `call` is `call: rel(!relcall)`. `relcall` is the system type comprising all terms that denote type correct calls to primitive or program defined relations. For the `relcall` type the `!` mode has a special meaning. It does not mean that `Var` has to be a ground term. It means only that all input arguments of the type declaration of the relation named by the functor of its compound term value must be ground. This is often guaranteed by preceding the meta call with the type test `type(Var,!relcall)`.

In a `forall`, each variable in *UVars must* appear in both *SimpleConj1* and *SimpleConj2*. It must be a local variable of the `forall`. All other variables of the `forall`, except the

existentially quantified variables and underscore variables, must have ground values when the `forall` condition is evaluated. Each *EVarsi* contains variables local to *SimpleConji*. If there are no existentially quantified variable, for either the antecedent or consequent, the `exists` operator is dropped.

Suppose we have a collection of `child_of(C, P)`, `age_is` and `male` dynamic relation facts. The following query will return as an answer each parent who only has male children age over 17.

```
child_of(_, P) & forall C (child_of(C,P) =>
                           exists A (male(C) & age_is(C,A) & A>17))
```

Note that `P` will be given a value before the evaluation of the `forall`, as required.

3.9.4 Action Sequences

An *Action Sequence* can be entered as an interpreter command. It has the form

```
{} (the empty action sequence)

or

ActOrQuery1 ; ActOrQuery12 ; ... ; ActOrQueryn, n>=1
```

with each *ActOrQueryi* one of:

- an *action call*: *ActExp*(*Exp1*,...,*Expk*), k>=0, where each *Expi* is an expression - a term that may contain function calls - and *ActExp* is an expression returning a *k*-ary action *a'*.
- a relation call, or a `(...)` bracketed *SimpleConj*, both preceded by the operator `?` - find at most one solution to a relation query and throws an exception if no solution exists.
- an action meta-call: `do` *A*, *A* a variable
- an *iterated action* (an action `forall`):

  ```
  forall UVars exists EVars1 SimpleConj ~>
                                    exists EVars1 SimpleActSeq)
  ```

- a new thread creation: `fork` *ActCall* as *Name*
- a blocking query:

  ```
  wait Query watch Depends after Time do Act
  ```

  where *Query* is a simple conjunction, *Depends* is a list of `++` and `--` dynamic relation names, *Time* is a number of seconds and *Act* is a simple action sequence.
- a non-blocking *message send*: *Mess* `to` *Handle*
- a blocking *single message receive*: *Mess* `from` *HandlePtn*
- a blocking *alternative message receive*:

  ```
  receive {
      MessPtn1 from HandlePtn1 :: Test1  ~> Act1
      ...
      ...
      MessPtnk from HandlePtnk :: Test1k  ~> Actk
  ```

```
timeout T ~> DefaultAct
}
```

where k>=1 and each *Acti* and *DefaultAct* is a *Simple Action Sequence* and each *Test1i* is a simple conjunction. The `timeout` rule is optional.

A *Simple Action Sequence* is an *Action Sequence* which does not contain any `forall`, `receive` or `wait` actions.

3.9.4.1 Receive Action

When a blocking `receive` is executed by a QuLog thread the message buffer of the thread is searched from the oldest message to the most recent to see if there is a message term that is an instance of one of the message patterns *MessPtni* from a sender with handle that is an instance of *HandlePtni*. The receive patterns are tried in their given before/after order. If a message is found that satisfies the two constraints of the ith rule it is removed, the *Acti* simple action sequence is executed and the receive is exited. Should the end message buffer be reached with no acceptable message found the thread will suspend until a new message arrives. Each new message is tested for acceptability. If there is a `timeout` rule this will be fired when T seconds have elapsed since the thread started its search for an acceptable message. *DefaultAct* is executed and the `receive` is exited. If there is no `timeout` rule the receive action will block indefinitely waiting for an acceptable message.

3.9.4.2 Discarding messages

The action rules `handle_messages` defines a non-terminating message receive behaviour that will process `request` and `ask` messages of the allowd form from agents allowed to send such messages. It removes all messages not satisfying either test. There is no action when a messages is discarded.

```
act handle_messages()
handle_messages()
 receive {
   request(TaskCall) from Agent :: can_request(Agent,TaskCall) ~>
                          {fork TaskCall as TaskId;
                           remember(taskfor(Agent,TaskId,Task)}
   ask(Id,Ans,Query) from Agent :: can_ask(Agent,Query) ~>
                          {eval(Query); ans(Id,Ans) to Agent}
   _ from _ ~> {}
   };
   handle_messages()
```

An alternative is to send some reply when a message is discarded as in:

```
handle_messages()
receive {
   request(TaskCall) from Agent :: can_request(Agent,TaskCall) ~>
                 {fork TaskCall as TaskId;
                  remember(taskfor(Agent,TaskId,Task)}
   request(TaskCall) from Agent ~> cannot_request(TaskCall) to Agent
   ask(Id,Ans,Query) from Agent :: can_ask(Agent,Query) ~>
                 {eval(Query); ans(Id,Ans) to Agent}
```

```
ask(Id,Ans,Query) from Agent ~> cannot_ask(Query) to Agent
discard Mess from Agent ~> invalid_message(Mess) to Agent
};
handle_messages()
```

Here the second and fourth rules send suitable messages when the tests of the first and third rules have failed for `request` and `ask` messages, and the last messagesends and `invalid_message` response for all other messages.

3.9.5 Relation Definitions

A relation definition consists of a sequence of rules (clauses) of the form

Head `::` *Commit* `<=` *Body*

Head is simple compound term of the form *RelNm(Term1,...,Termk)*, where *RelNm* is an atom. *Commit* is a simple conjunction, and *Body* is a complex conjunction. Both the `::` *Commit* and `<=` *Body* parts of the rule are optional. The rules for each relation name *RelNm* have must be contiguous and have the same number of arguments in the *Head*.

When a goal *Goal* with the same functor and arity as *Head* is called, the rules of the relation are tried in order. If *Goal* unifies with the head of the rule then the *Commit* part is called. If that succeeds then this rule is committed to (i.e. no subsequent rules are tried on backtracking) and *Goal* succeeds if and only if *Body* succeeds. If *Body* is not present it is treated as being the goal `true`.

If *Commit* is not present then *Goal* succeeds if *Body* succeeds but, on backtracking, subsequent rules will be tried.

The rule has the same semantics as the Prolog rule

Head `:-` *Commit*`, !,` *Body*

Note, however, that cut (`!`) is not part of Qulog.

As examples, the definitions of the relations `greater` and `sum_list` are given below.

```
rel greater(!num, !num, ?num)
greater(A, B, C) :: A > B <= C = A
greater(A, B, C) :: B > A <= C = B

rel sum_list(!list(num), ?num)
sum_list([], 0)
sum_list([H,..T], N) <= sum_list(T, M) & N = H+M
```

Note that in `N = H+M`, `H+M` is evaluated before unification and that the second rule of `greater` could have been written as

```
greater(A, B, C) <= B > A & C = B
```

3.9.6 Action Definitions

An action definition consists of a sequence of rules of the form

Head `::` *Commit* `~>` *Actions*

where *Head* is a simple compound term, *Commit* is a simple conjunction, and *Actions* is a an action sequence.

Both the `::` *Commit* and `~>` *Actions* parts of the rule are optional. The heads of each rule of an action have the same functor and arity.

The semantics of action definitions is the nearly the same as for relation definitions. The first difference is that at least one of the elements of the *Actions* sequence is an action which typically has a side effect such as writing, reading, sending a message or updating the database. The second difference is that backtracking is only local to the rule and so has equivalent semantics to the Prolog rule

Head `:-` *Commit*`, !,` *Actions*

Also, for each action definition, a catchall clause is added to the end of the defined clauses which throws an exception. This means that, if a call to an action does not match any of the defined clauses then an exception will be thrown as opposed to failure occurring (as would happen in a relation definition). This is also true for builtin actions. All builtin actions are also deterministic (i.e. no choice points are created by calling a builtin action. A consequence of this is that user defined actions also have this property. That is all actions are deterministic and all actions throw an exception rather than failing if no matches are found.

As examples, the definitions of the actions `ask_query` and `handle_response` are given below.

```
act ask_query(atom,  ??term, list(??term), handle)
ask_query(QId,Ans,QList,Ag) ~>
    ask(QId,Ans,QList) to Ag; Reply from Ag;
    handle_response(QId,Ag,Reply,Ans)

act handle_response(atom,handle, ??term, ??term)
handle_response(QId,_,tell(QId,ans(Ans)),Ans) :: true ~> {}
handle_response(QId,Ag,tell(QId,Reply),_) ~>
     writeLine(['Agent ',Ag,' responded ',Reply,' to query ',QId]); fail
```

3.9.7 Function Definitions

A function definition consists of a sequence of rules of the form

Head `::` *Commit* `->` *Expression*

where *Head* is an atom or simple compound term, *Commit* is a conjunct of goals, and *Expression* is a term. The `::` *Commit* part of the rule is optional. The heads of each rule of a relation have the same functor and arity.

When the function is called, the rules are tried and the first rule whose *Head* unifies with the function call and where *Commit* is true then the result returned is the evaluation of *Expression*. Note that rules without an explicit *Commit* have an implicit `true` commit and so no backtracking occurs.

As with action definitions, a catchall clause is added to the end so that if a function call does not match any of the rules then an exception is thrown.

As examples, the definitions of the functions `factorial` and `tree2list` are given below (using the tree type given above).

```
fun factorial(nat)->nat
factorial(0)->1
factorial(N) ::  N1 = N-1 & type(N1,nat) -> N*factorial(N1)
```

```
fun tree2list(tree(T)) -> list(T)
tree2list(empty()) -> []
tree2list(tr(LT, V, RT)) -> tree2list(LT) <> [V] <> tree2list(RT)
```

Note that `type(N1,nat)` is necessary above in order that the type checker can type check the recursive call on `factorial`.

3.9.8 TR Program Definitions

A TR-program definition has the form

Head `{`

TR Rule

...

TR Rule

`}`

where *Head* is an atom or simple compound term and each *TR Rule* has one of the forms given below.

The most simple TR rule has the form

Guard `~>` *TR Action*

where *Guard* is a conjunct of goals and *TR Action* is of the form given below.

At the other extreme, the two most complete forms of a TR rule are

Guard `while` *While* `min_time` *Duration* `~>` *TR Action*

and

Guard `inhibit` *Inhibit* `min_time` *Duration* `~>` *TR Action*

where *While* and *Inhibit* are conjuncts of goals and *Duration* is a number.

The most simple rule above is a particular form of either of the most complete rule forms where *While* and *Inhibit* are `false` and both durations are `0`.

Other variations of the general form are:

Guard `min_time` *Duration* `~>` *TR Action*

which is the same as

Guard `inhibit false min_time` *Duration* `~>` *TR Action*

Semantically, when a TR program is executed, the guards are checked in order until a guard is found that is true (with a given instantiation of variables). The corresponding *TR Action* is then executed.

As with actions, if there are no matching guards, an exception is thrown.

In the case where the rule contains a while condition, while *Guard* (with the same instantiation of variables) is true or *While* is true or the duration of the while part hasn't expired (since the time this rule was chosen) then this rule will continue to be the chosen unless an earlier guard becomes true.

In the case where the rule contains an inhibit condition, while *Guard* (with the same instantiation of variables) is true or *Inhibit* is true or the duration of the inhibit part hasn't expired (since the time this rule was chosen) then this rule will continue to be the chosen unless an earlier guard becomes true and *Inhibit* becomes false.

At that point, the guards will be checked again from the beginning. If no earlier rule guards are true, the same rule will refire if the guard is true with a different instantiation of variables (and execute the corresponding actions using the new instantiation of variables) or will continue as long as the guard remains true with the same instantiation of variables or *While* is true or the while duration hasn't expired or *Inhibit* is true or the inhibit duration hasn't expired.

Note that executing *TR Action* will typically mean stopping durative actions from previous rule firings and starting new actions (both discreet and durative). As an optimization, instead of stopping a durative action with given arguments and starting the same action with different arguments, the system will generate a 'modify action'.

The forms of *TR Action* for each TR rule are given below.

3.9.8.1 TR call

A *TR Action* can be a call on a TR program (possibly a recursive call on the same program). When such a rule is fired this TR program is executed.

3.9.8.2 Set of discreet and durative actions

A *TR Action* can be a comma separated set of discreet and durative robotic actions to be executed in parallel. `()` is the 'do nothing' action.

3.9.8.3 Timed sequence

A *TR Action* can be a semi-colon separated sequence of the form

`[`*Action* `:` *Duration*`, ..., `*Action* `:` *Duration*`]`.

where the last action need not have a maximum duration.

Each action above can be either a call on a TR program or a set of discreet and durative robotic actions.

When called, the first action is called and then after that duration is up, the second action is called and so on until the last action in the sequence is called. After its duration has expired then the sequence is repeated from the start. This repetitive action continues whilst the rule with the timed sequence remains a fired rule. If the last duration is missing, that action persists whilst the rule with the timed sequence remains a fired rule. The sequence is execututed just once.

3.9.8.4 Attached Qulog actions

TR Actions can have Qulog actions attached as below.

TRAction `++` *Action*

When called, both *TRAction* and *Action* will be executed. The Qulog action is typically a modification to the belief store or a message send action.

3.9.8.5 Example TR program

As an example of TR Programs, consider the following TR program (from the `examples/introduction` directory of the release) controlling a robot whose objective is to find, approach, and pick up an object using grippers.

```
%% We assume that if the program receives a dead_centre percept
%% it will also receive a centre percept
def dir::= left | right | centre | dead_centre

percept see(num, dir), holding()

def durative ::= move(num) | turn(dir)

def discrete ::= grab() | release()


%% We interpret holding true and see(0, centre) not true to mean that
%% the grippers are closed but not actually holding an object

tel get_object()
get_object() {
    holding() & see(0, dead_centre)       ~> ()
    holding() & see(0, centre)            ~> ()
    not holding()  & see(0, dead_centre)  ~> grab()
    not holding()  & see(0, centre)       ~> grab()
    not holding()                         ~> get_to()
    true                                  ~> release()
    }

tel get_to()
get_to() {
    see(0, dead_centre)  ~> ()
    see(0, centre)       ~> ()
    see(0, Dir)          ~> turn(Dir)
    see(_, dead_centre)  ~> move(6)
    see(_, centre)       ~> move(6)
    see(_, Dir) inhibit not see(_, dead_centre)
                         ~> move(4) , turn(Dir)
    true                 ~> [turn(left):10, move(4):10]
    }
```

Consider an initial state where no objects are seen and holding is false. When `get_object` is executed then the forth rule is fired causing `get_to` to be executed. The last of rules of `get_to` will be chosen (a timed interval). This will first cause the robot to start turning for 10 seconds and then start moving for 10 seconds. This will be repeated until an object is spotted.

At some point, say `see(10, left)` becomes true. This causes the sixth rule of `get_to` to fire (with `Dir` instantiated to `10`). Assuming this object is not moved by the environment, then eventually, say `see(8, centre)` becomes true. It might seem that the fifth rule should now fire because its guard becomes true. However, the inhibit condition prevents higher rules from firing. Once, say, `see(8, dead_centre)` becomes true then rule four will fire.

By over-achieving the guard of the sixth rule the "fluttering" between the fifth and sixth rule (without the inhibit condition) is eliminated.

The while condition is necessary because it takes over from the guard and mantains rule six as the active rule when seeing to the left is no longer true but seeing to the centre becomes true.

On the other hand, if, before `see(8, dead_centre)` becomes true the environment moves the object so that `see(8, right)` becomes true then there would be a refiring of rule six and the robot will start turning to the right.

Note that, if before the object is seen dead centre, `see(0, centre)` becomes true then rule four of `get_object` will fire. The inhibit only has a local effect - affecting rule choices within its own TR program.

Eventually, without interference from the environment, either `see(0, dead_centre)` or `see(0, centre)` will become true. The third or fourth rule of `get_object` will now fire (stopping the execution of `get_to`), causing the robot to grip the object. Under normal circumstances holding will become true and then the first or second rule will fire causing the robot to stop.

It may seem that the robot's job is done now that it has achieved its goal. However, the TR program is still monitoring the state and say the environment now removes the object from the robot's grip. Rule six will fire, opening the grippers, and then, once holding is no longer true, rule five will fire and the robot will go back to searching for an object.

4 Built-Ins

4.1 Introduction

This section contains descriptions of the functions, relations and actions of the Qulog library. In the interpreter you can see all their names and types by entering the command

```
| ?? stypes.
```

Many of these are Qu-Prolog primitive relations 'lifted' to QuLog by giving them appropriate type/mode declarations. Other Qu-Prolog relations and actions can be 'lifted' to the Qulog level be giving them a QuLog type declaration in your QuLog program file.

For example, if the primitives described in Section 4.3.3 [List Processing], page 35 had not already been made available for use in QuLog, all you would have needed to do was include their type declarations as given in that section.

As another example, there is a Qu-Prolog primitive

```
between(From,To,N)
```

for generating or testing an integer value `N` between given integer values `From` and `To`.

To use this in a QuLog program, add the moded type declaration

```
rel between(int,int,?int)
```

to your program file. It tells the QuLog mode/type checker that the relation is a Qu-Prolog primitive (that will be checked), and that in every use the first two arguments should be given as integers but the third integer argument may be given or may be returned as value of an unbound variable.

4.2 Input/Output

Actions:

`writeL(List)`

Write the elements of `List`. If the atom `nl_` appears in `List` it is written as a newline.

You can also use `sp_(N)` where N is a positive integer to insert N spaces. Strings in `List` are displayed without the string quotes `".."` unless you write them with `q_("...")`. The quotes are then put around the string.

`writeL : act(![??term])`

Example:

```
| ?? writeL(["List of atoms ",[a,b], nl_,
           "Set of nats ", {2,1,4,1}, nl_]).

List of atoms [a, b]
Set of nats {1, 2, 4}
```

with the next output on the same line.

`nl_` causes a new line to be output as would the string `"\n"`. In fact any of the C string control characters, such as `"\t"`, `"\s"` for tab and space respectively, can be put into a string and will have the intended effect unless the string is wrapped inside a `q_` term. So we could have written the above query as:

```
| ?? writeL(["List of atoms ",[a,b],
          "\nSet of nats ", {2,1,4,1}, "\n"]).
```

Other control term we can put in the list argument of `writeL` are:

`sp_(n)`, n positive integer. It will display n spaces.

`uq_(Atom)`, where `Atom` is an atom that normally needs to be quoted. It will be displayed without the single quotes.

`q_(String)`, where `String` is string will be displayed with string quotes.

`wr_(Var)`, will not display `Var` as an underscore followed by a sequence of digits, as is normal, but will give it a name such as `A`, `B`, `C` when displayed and will give subsequent occurence of `Var` in the list to be output using the given name for `Var`.

The following query illustrates the the use of `uq_` and `wr_`.

```
| ?? writeL([uq_('Hello')," there\n",wr_(_895),sp_(2),
              _895,sp_(2),wr_(_678),nl_]).

Hello there
A  A  B

_895 = A : Ty1
_678 = B : Ty2

success
```

`writeLine(List)`

The same as `writeL` but with a trailing newline.

Example:

```
| ??writeLine(["A list ", [a, b], sp_(2), {2, 1, 4, 1},
               " followed by a set."]).

A list [a, b]  {1, 2, 4} followed by a set.
```

with the next output being at the beginning of the next line.

`readT(Term)`

Matches `Term` with the next term denoted by the next sequence of characters typed at the terminal followed by fullstop, return.

`readT :act(??term)`

Example:

```
| ?? readT(X).

f(A).
```

```
X = f(A) : term
```

Note that this read remembers the names of variables (the `A` above). A consequence of this, given the occurs check in unification, is that the following query throws an exception.

```
| ?? readT(A).

f(A).
Qulog exception - exception term: readT_unify_error(_95, f(_95))
```

For this reason you should use a fresh variable when reading a term.

4.3 Terms

4.3.1 Comparison of Terms

Two terms are compared according to the standard ordering, which is defined below. Items listed at the beginning come before the items listed at the end. For example, numbers are less than atoms in the standard ordering.

1. Variables, in age ordering (older variables come before younger variables).
2. Numbers, in numerical ordering.
3. Atoms, in character code (ASCII) ordering.
4. String, in standard string odering.
5. Compound terms are compared in the following order:
    1. Arity, in numerical ordering.
    2. Functor, in standard ordering.
    3. Arguments, in standard ordering, from left to right.
6. Sets, in dictionary order on elements.
7. Lists, in dictionary order on elements.

The above ordering is used when constructing sets.

The following relations use the above ordering to test terms.

`Term1 @> Term2`

Succeeds if `Term1` is greater than`Term1` in the above ordering.

`Term1 @>= Term2`

Succeeds if `Term1` is greater than or equal to `Term2` in the above ordering.

`Term1 @< Term2`

Succeeds if `Term1` is less than`Term1` in the above ordering.

`Term1 @=< Term2`

Succeeds if `Term1` is less than or equal to `Term2` in the above ordering.

4.3.2 Testing of Terms

These testing predicates are used to determine various properties of the data objects, or apply constraints to the data objects.

Relations:

`type(Term, Type)`

Succeed if `Term` is a non-variable of type `Type`.
`type : rel(??term, !typeE(_))`

Example:

```
| ?? type(a, atomic).

yes

| ?? type(a, int).

no
```

`ground(Term)`

Succeed if `Term` is ground.
`ground: rel(??term)`

`isa(Term, Type)`

Succeed if `Term` is of type `Type` and `Type` is a finite type.
`isa: (?term, !typeE(_))`

Example: For this example we assume the following type declarations.

```
name ::= "Alice" | "Bob" | "Carol"
status ::= good(name) | bad(name)
| ?? isa(X, status).

X = good("Alice")
...
X = good("Bob")
...
X = good("Carol")
...
X = bad("Alice")
...
X = bad("Bob")
..
X = bad("Carol")

| ?? isa(good("Bob"), status).

yes
```

```
| ?? isa(2, nat).

Error: nat is not a finite type in isa(2, nat) of
isa(2, nat)
```

`template(Term)`

Succeed if `Term` is atomic or a compound term with a ground functor.
`template :rel(??term)`

4.3.3 List Processing

`append(L1, L2, L3)`

Succeed if `L3` is the concatentation of `L1` and `L2`

```
rel append(!list(T), !list(T), ?list(T)),
    append(?list(T), ?list(T), !list(T)),
    append(!list(??T), !list(??T), ?list(??T)),
    append(?list(??T), ?list(??T), !list(??T)),
    append(??list(T), ??list(T), ??list(T))
```

`reverse(L1, L2)`

Succeed if list `L2` is the reverse of `L1`.

```
rel reverse(!list(T), ?list(T)),
    reverse(?list(T), !list(T)),
    reverse(!list(??T), ?list(??T)),
    reverse(?list(??T), !list(??T))
```

`sort(L1, L2, Order)`

Will match the second argument against the first argument sorted by the transitive order relation given as the third argument, without instantiating variables in the first argument. If the relation is asymmetric, duplicate terms will be removed.

```
rel sort(!list(!T), ?list(T), !rel(!T, !T)),
    sort(!list(@T), ?list(@T), !rel(@T, @T))
```

`member(X, L)`

Succeed if `X` is in `L`.

```
rel member(?T, !list(T)),
    member(??T, ??list(T))
```

`X in L`

Succeed if `X` is in complete list of terms `L`.

It almost has the same uses as `member` except that it it must be given a complete list of possibly non-ground terms.

As its type indicates, `in` can also be used to access single character substrings of a string, a tuple and the ground elements of a set.

```
rel ?T in !list(T),
    ??T in !list(??T),
    ?T in !tuple(T),
    ??T in !tuple(??T),
```

```
?T in !set(T),
?string in !string
```

4.4 Arithmetic

The following arithemetic functions are available.

`Num1 + Num2`

Returns the sum of `Num1` and `Num2`.
`fun nat + nat -> nat, int + int -> int, num + num -> num`

`Num1 - Num2`

Returns the difference of `Num1` and `Num2`.
`fun int - int -> int, num - num -> num`

`-Num1`

Returns the negative value of `Num`.
`fun -(int) -> int, -(num) -> num`

`Num1 * Num2`

Returns the product of `Num1` and `Num2`.
`fun nat * nat -> nat, int * int -> int, num * num -> num`

`Num1 // Num2`

Returns the integer division of `Num1` and `Num2`.
`fun //(nat, nat) -> nat, //(int, int) -> int`

`Num1 / Num2`

Returns the division of `Num1` and `Num2`.
`fun num / num -> num`

`Num1 mod Num2`

Returns the mod of `Num1` and `Num2`.
`fun int mod int -> int`

`Num1 ** Num2`

Returns `Num1` raised to the power `Num2`.
`fun nat ** nat -> nat, int ** int -> int, num ** num -> num`

`Int >> Nat`

Returns the bitwise right shift of Int with respect to Nat.
`>>: (int, nat) -> int`

`Int << Nat`

Returns the bitwise left shift of Int with respect to Nat.
`fun int << nat -> int`

`Int1 /\ Int2`

Returns the bitwise AND of Int1 and Int2.
`fun int /\ int -> int`

`Int1 \/ Int2`

Returns the bitwise OR of Int1 and Int2.
`fun int \/ int -> int`

`\ Int`

Returns the bitwise complement of Int.

```
fun \(int) -> int
```

`abs(Num)`

Returns the absolute value of Num.

```
fun abs(int) -> nat, abs(num) -> num
```

`sqrt(Num)`

Returns the square root of Num.

```
fun sqrt(num) -> num
```

`round(Num)`

Returns the nearest integer to Num.

```
fun round(num) -> int
```

`floor(Num)`

Returns the greatest integer less than Num.

```
fun floor(num) -> int
```

`ceiling(Num)`

Returns the smallest number greater than Num.

```
fun ceiling(num) -> int
```

`pi()`

Returns value 3.14159 for PI.

```
fun pi() -> num
```

`e()`

Returns 2.71828 value for E.

```
fun e() -> num
```

`sin(Num)`

Returns the sin of Num.

```
fun sin(num) -> num
```

`cos(Num)`

Returns the cos of Num.

```
fun cos(num) -> num
```

`tan(Num)`

Returns the tan of Num.

```
fun tan(num) -> num
```

`asin(Num)`

Returns the arcsin of Num.

```
fun asin(num) -> num
```

`acos(Num)`

Returns the arccos of Num.

```
fun acos(num) -> num
```

`atan(Num)`

Returns the arctan of Num.
`fun atan(num) -> num`

`atan2(Y, X)`

Returns the atan2 of Y and X. This returns an angle in the range `(-pi, pi]`.
`fun atan2(num,num) -> num`

4.5 Other Functions

`now()`

Returns the current time.
`fun now() -> num`

`exec_time()`

Returns the lapsed time in seconds since this qulog process was started
`fun exec_time() -> num`

`start_time()`

Returns the time at which this qulog process was started
`start_time: () -> num`

`random_num()`

Returns a random number in `[0,1)`.
`fun random_num() -> num`

`random_int(Lower, Upper)`

Returns a random integer in the interval `[Lower, Upper]`.
`fun random_int(int, int) -> num`

`S1 union S2`

Returns the union of sets `S1` and `S2`.
`fun union(set(T), set(T)) -> set(T)`

`S1 inter S2`

Returns the intersection of sets `S1` and `S2`.
`fun inter(set(T), set(T)) -> set(T)`

`S1 diff S2`

Returns the set difference of sets `S1` and `S2`.
`fun diff(set(T), set(T)) -> set(T)`

`L1 <> L2`

Returns the concatination of lists `L1` and `L2`.
`fun list(T) <> list(T) -> list(T)`

`S1 ++ S2`

Returns the concatination of strings `S1` and `S2`.
`fun string ++ string -> string`

`#L`

Returns the length of the list, set, or string `L`.
`fun #list(T) -> nat, #set(T) -> nat, #string -> nat`

`F@..Args`

Returns the compound term obtained by applying `F` to `Args`.
`fun (term @.. list(term)) -> term`

Example:

```
| ?? X = a @.. [1,2].

X = a(1, 2) : term
```

`@..` can also be used to split up a compound term as in the following example.

```
| ?? a(1,2) =? F@..Args.

F = a : atom
Args = [1, 2] : list(nat)
```

`$Name`

Here `Name` is an atom that must have been initialised with a statement
`int Name:=Integer`, e.g. `int count:=0` or
`num Name:=Number`, e.g. `num savings:=678.50`

in the program. It returns the current value associated with `Name`
which can be updated by primitive actions (see Section 4.7 [Other Actions], page 40).

4.6 Other Relations

`true()`

Always succeeds.
`rel true()`

`false()`

Always fails.
`rel false()`

`Term1 = Term2`

Succeeds if `Term1` and `Term2` unify.

Any function calls in each argument term are evaluated first using strict evaluation - expression arguments evaluated first - as with every relation call.
`rel =(??term, ??term)`

`Term1 =@ Term2`

Succeeds if `Term1` and `Term2` unify without instantiating variables in Term2 (one-sided unification).

`rel =@(??term, @term)`

`N1 > N2`

Succeeds if `N1` is numberically greater than `N2`.
`rel >(!num, !num)`

`N1 >= N2`

Succeeds if `N1` is numberically greater than or equal to `N2`.
`rel >=(!num, !num)`

`N1 < N2`

Succeeds if `N1` is numberically less than `N2`.
`rel <(!num, !num)`

`N1 =< N2`

Succeeds if `N1` is numberically less than or equal to `N2`.
`rel =<(!num, !num)`

`X in T`

Succeeds if `X` is a term and and `T` is a list, tuple, or set of terms and `X` is an element of `T` or if `X` and `T` are both strings and `X` is a single character string occurring in `T`.

```
rel ?T in !list(T),
    ??T in !list(??T),
    ?T in !tuple(T),
    ??T in !tuple(??T),
    ?T in !set(T),
    ?string in !string
```

`string2term(S, T)`

Succeeds if `T` is the term obtained by parsing the string `S` as a Qulog term.
`rel string2term(!string, ??term)`

`this_thread_name(Name)`

Succeeds if `Name` is the name of this thread
`current_thread : rel(?atom)`

`get_active_resources(ResourceInfo)`

Succeeds if `ResourceInfo` is the list of terms `res(atom,list(resource))` giving resources used by each running task in a multi-tasking agent. The atom is task name.
`rel get_active_resources(?term)`

`get_waiting_resources(ResourceInfo)`

Succeeds if `ResourceInfo` is the list of terms `res(atom,list(resource))` giving resources needed by each waiting task in a multi-tasking agent. The atom is task name.
`rel get_waiting_resources(?term)`

4.7 Other Actions

`fork_as(Action, Name)`

Fork a new Qulog thread, give it the name `Name`, and start the thread executing `Action`. If `Name` is a variable it will be instantiated to a name given by the system. If `Name` is given it must not be the name of an existing thread.
`act fork_as(actcall, ?atom)`

Alternative syntax: `fork Action as Name`

`fork_light_as(Action, Name)`

Fork a new Qulog thread, give it the name `Name`, and start the thread executing `Action`. If `Name` is a variable it will be instantiated to a name given by the system. If `Name` is given it must not be the name of an existing thread. The sizes of the various

memory areas is set to be very small. This produces a very light weight thread and is useful for threads that do minimal computation - for example a simple message handling thread.
`act fork_light_as(actcall, ?atom)`

Alternative syntax: `fork_light Action as Name`

`from(Term, Handle)`

This is the agent message receive action. It will succeed it there is a message term in that threads message buffer whose message term unifies with `Term` and whose agent handle unifies with `Handle`. If not the call will suspend and be repeatedly retried as new messages arrive until it succeeds. When it does succeed, the matched message will be removed from the message buffer.
`act from(term?, ?agent_handle)`

Alternative syntax: `Term from Handle`

`to(Term, Handle)`

This is the agent message send action. It sends `Term` as a message to the agent (of possibly a process on another machine) whose agent address is `Handle`.
`act to(??term, !agent_handle)`

Alternative syntax: `Term to Handle`

`from_thread(Term, Handle)`

This is the message receive action. It will succeed it there is a message term in that threads message buffer whose message term unifies with `Term` and whose handle unifies with `Handle`. If not the call will suspend and be repeatedly retried as new messages arrive until it succeeds. When it does succeed, the matched message will be removed from the message buffer.
`act from_thread(term?, ?handle)`

Alternative syntax: `Term from_thread Handle`

`to_thread(Term, Handle)`

This is the message send action. It sends `Term` as a message to the thread (of possibly another process on another machine) whose address is `Handle`.
`act to_thread(??term, !agent_handle)`

Alternative syntax: `Term to_thread Handle`

`thread_sleep(Secs)`

Causes the executing thread to suspend for `Secs` seconds.
`act thread_sleep(!num)`

Dynamic facts can be remembered and forgotten using the actions below. Each such action atomically modifies the collection of dynamic facts.

`forget_remember(Forget, Remember)`

Forget the dynamic facts that match the patterns in `Forget` and then remember the dynamic facts in `Remember`

Alternative syntax: `forget F remember R`

`remember(Dyn)`

Adds each dynfact in (`Dyn`) as a new last dynamic fact for its functor relation name `R`. `R` must have been declared as dynamic.
`act remember(dynfact)`

`remember_for(Dyn, Secs)`

The same as `remember` except that `Dyn` is forgotten after `Secs` seconds.
`act remember_for(dynfact, num)`

Alternative syntax: `remember Dyn for Secs`

`rememberA(Dyn)`

Adds its ground relcall argument (`Dyn`) as a new first dynamic fact for its functor relation name `R`. `R` must have been declared as a belief.
`act rememberA(dynfact)`

`rememberA_for(Dyn, Secs)`

The same as `rememberA` except that `Dyn` is forgotten after `Secs` seconds.
`act remember_for(dynfact, num)`

Alternative syntax: `rememberA Dyn for Secs`

`forget(DynPtn)`

Remove the first dynamic fact matching each entry of `DynPtn`. Note that `Dynptn` may contain variables within the arguments of each entry. `forget` always succeeds even if there are no matching facts.
`act forget(dyncall)`

`Name := Expression`

Here `Name` is an atom that must have been initialised with a statement

`int Name:=Integer`, e.g. `int count:=0` or

`num Name:=Number`, e.g. `num savings:=678.50`

in the program. These statements are shorthand for `dyn` declarations and a definition using one fact of a unary relation called `Name`. They are respectively expanded into:

```
dyn Name(int)
Name(Integer)

dyn Name(num)
Name(Number)
```

The action `Name := Expression` is the same as

```
forget Name(_) remember Name(Expression).
```

`Name` can be used as though it were a global variable. To access its value the operator `$` is applied. The expression `$Name` evaluates to the current `int` or `num` value stored in `Name`, i.e. in the current `Name` belief.

`act :=(!rel(?num), !num), :=(!rel(?int), !int)`

`Name +:= Expression`

As above, `Name` is an atom that must have been initialised with a statement

`int Name:=Integer` or `num Name:=Number`

in the program.

The action `Name +:= Expression` is the same as

```
forget Name(Val) remember Name(Val+Expression).
```

`act +:=(!rel(?num), !num), +:=(!rel(?int), !int)`

Example use

```
count +:= 1
```

for increasing value held in `count` by 1.

`Name -:= Expression`

As above, `Name` is an atom that must have been initialised with a statement

```
int Name:=Integer or num Name:=Number
```

in the program.

The action `Name -:= Expression` is the same as

```
forget Name(Val) remember Name(Val-Expression).
```

`act -:=(!rel(?num), !num), -:=(!rel(?int), !int)`

Example use

```
savings -:= 67.90
```

for decreasing the value held in `savings` by `67.90`.

4.8 TeleoR Specific Actions

The following actions are available when the system is running in TeleoR mode (after the command `teleor` has been executed in the interpreter).

`actions(Actions)`

An interpreter command that can be used in teleor mode when an agent has been started using `start_agent`. `Actions` is a list of actions that agent wants the robotic interface to perform. The given actions must have been declared as discrete or durative.

`act actions(list(discrete || durative))`

`kill_agent`

Kill the current agent started using `start_agent`.

`act kill_agent()`

`kill_task(Task)`

Kill the task with name `Task` started using `start_task`.

`act kill_task(atom)`

`start_agent(Handle, Convention)`

Start a new agent. `Handle` is the message address of the robot interface or simulation with which the agent will interact. `Convention` is the percepts update convention being used. This is one of: `all` if the robot sends all the percepts each time it sends percepts or `updates` if the robot only sends changes to percepts.

Percepts can be post-pocessed using one of the user defined actions `post_percepts_all_` or `post_percepts_updates_`. See below.

`act start_agent(handle, atom)`

`start_task(Name, TRCall)`

Start a new task (as a thread) whose name is `Name`. `TRCall` is the TeleoR call to be executed in the thread.

`act start_task(atom, trcall)`

`post_percepts_all_(Percepts)`

This is a user defined action that is executed when processing percepts after the system defined percepts process when using the `all` convension. Both the system defined handler and the user handler are processed atomically.

`post_percepts_all_(list(dyncall))`

`post_percepts_updates_(Percepts)`

This is a user defined action that is executed when processing percepts after the system defined percepts process when using the `updates` convension. Both the system defined handler and the user handler are processed atomically.

`post_percepts_updates_(list(percept_update_))`

5 Standard Operators

We use the Prolog notation for operator declarations even though, unlike Prolog, QuLog's syntax cannot be extended by adding application specific operators and the QuLog parser is not an operator precedence parser. `op` is not a system predicate of QuLog. Using Prolog `op` statements gives us a succint way of summarising the precedence relationship between the operators that is implicit in the formal syntax rules.

In each `op` statement the number is the 'binding' power of the operator, called the *precedece* of the operator. The higher the precedence, the higher up the parse tree, so the less binding the operator. For example, `+` has higher precedence than `*` , so `X+Y*Z` is really `X+(Y*Z)` and we have to use brackets if we want to have the expression `(X+Y)*Z`.

`fx` means the operator is prefix and cannot be followed immediately by an expression with top operator of the same precedence unless that expression is bracketed.

`xfx` means that the operator is an infix non-associative operator and must have expression arguments for which the top operator has lower precedence, or the expressions arguments are bracketed. So, `(X**Y)**Z` needs the brackets.

`xfy` means that the operator is an infix right associative operator, and `yfx` means that the operator is an infix left associative operator. More specically, a `xfy` operator can have an expression to the right with a top operator of equal or lower precedence and that expression being implicitly bracketed. For a `yfx` this implicit bracketing applies to the expression on the left hand side. So, `X*5 mod 6` is implicitly `(X*5) mod 6`.

Note that `?` has two precedences as an infix operator. One is for its use in an interpreter query after the required initial number of solutions and/or variable bindings have been given. For this use it must have higher precedence than `&`. The second use is in `<>` and `++` patterns when giving a single condition constraint on a sub-string or sub-list. For this use it must have lower precedence than `<>` and `++`.

The mode annotations are not in the table as prefix or infix operators as they will always be inside a bracketed sequence of annotated types.

We have not included `forall` or `exists` as they are both prefix operators taking two arguments, the sequence of variables that immediately follows and then some operator expression. This is an `=>` or `~>>` implication in the case of `forall` and possibly an `&` conjunction in the case of `exists`.

`start_task` and `start_agent` are just reserved words and are never followed by an operator expression.

In QuLog comma is not treated as an operator. It is just an expression separator.

```
op(1100, xfx, [ <=, ~>>, ->, ~>, =>])
op(1050, xfx, [ ::= ])
op(1030, xfx, [ |, || ])
op(1030, xfx, [ .. ])
op(1020, xfx, [ ::, ? ])
<op(1020, xfy, [ while, inhibit ])
op(1000, xfy, [ &, ; ])
op(900,  fx,  [not , once,  watch , watchC, unwatch, show,
```

```
              types, stypes, remember, forget, call, do, wait])
op(850, xfx, [ for, after ])
op(800, xfx, [ ? ])
op(700,  xfx, [= , \= , :=, +:=, -:=, =? , == , \== , @< ,   @=< ,
              @> , @>= , @.. , in , =:= , =\= , < , =< , > , >= ])
op(600, xfy, [ ++, <> ])
op(550, xfx, [ ? ])
op(500, yfx, [ + , - , /\ , \/, union, diff ])
op(450, yfx, [ to, from ])
op(400, yfx, [ * , / , // , rem , mod , inter, << , >> ])
op(200, xfx, [ ** ])
op(200, fy,  [ + , - , \ ])
op(100, xfx, [ .. ])
op(50, xfx, [ : ])
op(50, fx, [ $, # ])
```

Appendix A EBNF Grammar for Qulog

```
(* The EBNF grammar for qulog/teleor *)

(*
Notation used as in:
   https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

We assume the following non-terminals that group tokens. All other
tokens in the grammar are given as strings.

atom:  the allowed atoms of qulog
string:  double-quoted strings
var: the variables of qulog
int: integers
float: floating point numbers

*)

anynum = int | float;

(* ------ query ------ *)

query =
    (
       ( int, "of", var_seq, "::", basic_query ) |
       ( int, "of", basic_query ) |
       ( int, "of", "exists", var_seq, conditions ) |
       ( var_seq, "::", basic_query ) |
       ( "exists", var_seq, conditions ) |
       basic_query
     );

basic_query =
    ( "prolog" ) |
    ( "bs" ) |
    ( "logging", atom ) |
    ( "logging", "(", atom , ")" ) |
    ( "logging", atom, "@", atom ) |
    ( "logging", "(", atom, "@", atom , ")" ) |
    ( "types", {atom_seq} ) |
    ( "stypes", {atom_seq} ) |
    ( "show", {atom_seq} ) |
    ( "watch", {atom_seq} ) |
    ( "watch", "(", {atom_seq}, ")" ) |
    ( "watchR", {atom_seq} ) |
```

```
    ( "watchR", "(", {atom_seq}, ")" ) |
    ( "unwatch", {atom_seq} ) |
    ( "unwatch", "(", {atom_seq}, ")" ) |
    ( "unwatchR", {atom_seq} ) |
    ( "unwatchR", "(", {atom_seq}, ")" ) |
    ( "answers", int ) |
    ( "consult", atom ) |
    ( "consult", "(", atom, ")" ) |
    ( "pconsult", atom ) |
    ( "pconsult", "(", atom, ")" ) |
    ( "[", atom_seq, "]" ) |
    conditions |
    action_seq;



(* ------ program item ------ *)
(* Note: a program file is parsed one program item at a time *)

program_item =
    ( "?-", term ) |
    enum_definition |
    macro_definition |
    declaration |
    ( compound_term, rel_rule_body ) |
    ( compound_term, act_rule_body ) |
    ( compound_term, fun_rule_body ) |
    ( simple_compound, tel_procedure_body );

(* ------ type definitions ------ *)

enum_definition = "def", definition_head, "::=", enum_type {string};

macro_definition = "def", definition_head, "==", macro_type {string};

(* either atom type or polymorphic type with args *)
definition_head =  atom |  ( atom, bracketed_var_seq );

(* ------ types ------ *)

enum_type =
      ( int, "..", int ) |
      ( atom, "|", atom, {"|", atom} ) |
      ( string, "|", string , {"|", string} ) |
      ( anynum, "|", anynum, {"|", anynum} ) |
      ( simple_compound, {"|", simple_compound} ) |
      ( "(", enum_type, ")" ) ;
```

```
macro_type =
       ( atom_or_simple_compound,  "||",
          atom_or_simple_compound,  {"||", atom_or_simple_compound} ) |
      macro_type_expression;

macro_type_expression =
    simple_type_compound |
     ( "(", type_expression, ",", type_expression,
            {",", type_expression}, ")" ) |
     code_type_expression;

type_expression =
     code_type_expression |
     var |
     simple_type_compound |
     atom |
     ( "(", type_expression, ")" ) |
     ( "(", type_expression, ",", type_expression,
         {",", type_expression}, ")" );

code_type_expression =
     fun_type_expression |
     dyn_type_expression |
     rel_type_expression |
     act_type_expression |
     tel_type_expression;

fun_type_expression =
    ( "fun", bracketed_type_seq, "->",  type_expression ) ;

rel_type_expression =
    ( "rel", bracketed_annotated_type_seq);

dyn_type_expression =
    ( "dyn", bracketed_type_seq);

act_type_expression =
    ( "act", bracketed_annotated_type_seq);

tel_type_expression =
    ( "tel", bracketed_type_seq);

bracketed_type_seq =
    ( "(", ")" ) |
    ( "(", type_expression, { ",", type_expression}, ")" );
```

```
mode_annotation = "!" | "?" | "??" | "@";

bracketed_annotated_type_seq =
    ( "(", ")" ) |
    ( "(", annotated_type_expression,
        { ",", annotated_type_expression}, ")" );

annotated_type_expression =
    mode_annotation |
    ( mode_annotation , inner_annotated_type_expression ) |
    inner_annotated_type_expression;

inner_annotated_type_expression =
   type_expression |
   ( atom, bracketed_annotated_type_seq);

(* ------ declarations ------ *)

declaration =
    ( "fun", fun_declaration, {",", fun_declaration}, [ string ] ) |
    ( "act", annotated_declaration, {",", annotated_declaration}, [string]) |
    ( "rel", annotated_declaration, {",", annotated_declaration}, [string]) |
    ( "dyn", unannotated_declaration,
             {",", unannotated_declaration}, [ string ] ) |
    ( "rem", unannotated_declaration,
             {",", unannotated_declaration}, [ string ] ) |
    ( ( "tel" | "task_start" | "task_atomic" ), unannotated_declaration,
             {",", unannotated_declaration}, [ string ] ) |
    ( "percept", unannotated_declaration,
             {",", unannotated_declaration}, [ string ] ) |
    global_num_declaration |
    resources_declaration;

fun_declaration =
    unannotated_declaration, "->", type_expression;

unannotated_declaration =
    atom, (
        ( "(", ")" ) |
        ( "(", [var, ":"], type_expression,
           { ",",  [var, ":"], type_expression}, ")" )
        );

annotated_declaration =
    atom, (
        ( "(", ")" ) |
        ( "(", [var, ":"], annotated_type_expression, ["default", term],
```

```
           { ",",  [var, ":"],
                annotated_type_expression, ["default", term]}, ")" )
        );

global_num_declaration =
    ("int", atom, ":=", int) |
    ("num", atom, ":=", num);

resources_declaration =
    "resources", atom_seq;
(* make resources a user defined sys type - above not needed *)

(* ------ rule definitions ------ *)

rel_rule_body =
    ["::", conditions], ["<=", conditions];

act_rule_body =
    ["::", conditions], "~>", action_seq;

fun_rule_body =
    ["::", conditions], "->", term;

tel_procedure_body =
    "{", ">>>", tel_rule, { tel_rule }, "<<<", { tel_rule }, "}" |
    "{", tel_rule, { tel_rule }, "}";


(* ------ conjunction and conditions ------ *)

conditions = a_condition, { "&", a_condition };

a_condition =
    ( "forall", var_seq,
       "(", exists_conditions, "=>", exists_conditions, ")" ) |
    ( "not", exists_a_condition ) |
    ( "not", "(", exists_conditions, ")" ) |
    ( "once", a_condition ) |
    ( "once", "(", conditions, ")" ) |
    ( int, "of", (", var_type_seq, "::",
         exists_conditions__, ")", "query_at", pedro_handle ) |
    ( "(", int, "of", var_type_seq, "::",
         exists_conditions__, ")", ")" , "query_at", pedro_handle ) |
    ("(", var_type_seq, "::",
         exists_conditions__, ")", "query_at", pedro_handle ) |
    simple_condition;
```

```
exists_a_condition =
    ( "exists", var_seq, simple_condition ) |
    a_condition;

exists_conditions =
    ( "exists", var_seq, a_condition ) |  conditions;

simple_condition =
    ( "(", conditions, ")" ) |
    "true" | "false" |
    ( "type", "(", term, ",", annotated_type_expression, ")" ) |
    ( "listof", "(", term, ",", term, "::", exists_conditions, ")" ) |
    compound_term |
    ( term, "=?", qeqrhs) |
    ( term, test_op, term );

qeqrhs =
    ( qeqrhs_string_term, "++", qeqrhs_string_term,
                 {"++", qeqrhs_string_term} ) |
    ( qeqrhs_list_term, "<>", qeqrhs_list_term, {"<>", qeqrhs_list_term} );

qeqrhs_string_term =
    ( simple_string_term, "::", conditions ) |
    simple_string_term;

qeqrhs_list_term =
    ( simple_list_term, "::", conditions ) |
    simple_list_term;

(* ------ actions ------ *)

action_seq = action, {";", action};

action =
    ( "forall",  var_seq, "{",
       exists_conditions, "~>", action_seq, "}") |
    simple_action ;

simple_action =
    ( "{", "}" ) |
    ( "{", action_seq, "}" ) |
    ( atom, global_num_op, term) |
    compound_term |
    ( "atomic_action", simple_action ) |
    ( "case", "{", case_alt, { case_alt }, "}" ) |
    ( "wait", "(", conditions, ")" ) |
    ( "wait_case", "{", case_alt, { case_alt },
```

```
                    [ "timeout", term, "~>", action_seq ], "}" ) |
    ( "receive", "{", receive_alt, { receive_alt },
                    [ "timeout", term, "~>", action_seq ], "}" ) |
    ( "raise", term ) |
    ( "try", action_seq, "except", "{", except_alt, { except_alt }, "}" ) |
    ( "forget", forget_remember_term, ["remember", forget_remember_term] ) |
    ( "remember", forget_remember_term, ["for", term] ) |
    ( "rememberA", forget_remember_term, ["for", term] ) |
    ( "subscribe", term, "as", var ) |
    ( "?", "(", exists_conditions, ")" ) |
    ( "?", a_condition ) |
    ( term, "to", agent_handle ) |
    ( term, "to_thread", pedro_handle ) |
    ( term, "from", agent_handle, ["::", conditions ] ) |
    ( term, "from_thread", pedro_handle, ["::", conditions ] ) |
    ( "fork", action, "as", ( var | atom | (var, "/", atom ) ) ) |
    ( "fork_light", action, "as", ( var | atom | (var, "/", atom ) ) ) |
    ( "fork_sizes", action,
                    "with", fork_size_list,
                    "as", ( var | atom | (var, "/", atom ) ) );

case_alt =
   conditions, "~>", action_seq;


receive_alt =
    ( term, "from", agent_handle, [ "::", conditions ], "~>", action_seq ) |
    ( term, "from_thread", pedro_handle, [ "::", conditions ],
           "~>", action_seq ) |
    ( "query", term, "from_thread", pedro_handle, [ "::", conditions ],
           "~>", action_seq );

catch_alt =
    term, [ "::", conditions ], "~>", action_seq;

(* ------ TR rule ------ *)

tel_rule =
    tr_rule_LHS, "~>", tr_rule_RHS;

tr_rule_LHS =
    conditions,
    [( "while", while_inhibit) | ("inhibit", while_inhibit) |
    ("min_time", term)];


while_inhibit =
```

```
    conditions |
    ( "min_time", term ) |
    (conditions, "min_time", term );

tr_rule_RHS = tr_action, [ "++", action_seq ];

tr_action =
    ( "(", ")" ) |
    ( simple_tr_action, {",", simple_tr_action} ) |
    ( "(", simple_tr_action, {",", simple_tr_action}, ")" ) |
    ( "[", tr_timed_seq, "]" );

tr_timed_seq =
    simple_tr_action, ":", term, {",",  simple_tr_action, ":", term },
        [",", simple_tr_action];

simple_tr_action = var | compound_term;

(* ------ term ------ *)

term =
    simple_term |
    ( simple_term, "<>", simple_term, { "<>", simple_term } ) |
    ( simple_term, "++", simple_term, { "++", simple_term } ) |
    ( simple_term, set_op, simple_term, { set_op, simple_term } ) |
    ( simple_term, arith_op, simple_term, { arith_op, simple_term } );

simple_term =
    var |
    atom |
    anynum |
    string |
    compound_term |
    agent_handle | pedro_handle |
    ( "(", term, ")" ) |
    ( tuple ) |
    ( "$", atom ) |
    set_term |
    set_comprehension |
    list_term |
    list_comprehension |
    ( "-", simple_term ) |
    ( "#", simple_term );

set_comprehension =
   "{", simple_term, "::", exists_conditions, "}";
```

```
list_comprehension =
    ( "[", simple_term, "::", exists_conditions, "]" );



(* ------ auxilary rules -------*)

atom_seq = atom, {",", atom};

var_seq = ( var, {",", var} ) | ( "(", var, {",", var}, ")" );

atom_or_simple_compound = atom | simple_compound;

simple_compound = atom, bracketed_arg_seq;

simple_type_compound = atom, bracketed_type_arg_seq;


compound_term = ( atom | var ), bracketed_arg_seq, { bracketed_arg_seq };

(* compound means ultimate functor is an atom *)

bracketed_arg_seq  = ("(", { term, "," }, ")") | ("(", arg_seq, ")");

bracketed_type_arg_seq = ("(", type_expression, {",", type_expression}, ")");

tuple = ( "(", ")" ) | ("(", term, ",", arg_seq, ")");

bracketed_var_seq  = ("(", ")") | ("(", var_seq, ")");

arg_seq = term, {",", term};

test_op =
    "=" | "==" | "\=" | "=@" | ">" | "<" | ">=" | "=<" |
    "@>" | "@<" | "@>=" | "@=<" | "in";

global_num_op =
    ":=" | "+:=" | "-:=";

simple_string_term =
    string | var | ( var, "/", ( string | var ) ) | compound_term;

simple_list_term =
    var | compound_term | list_term | list_comprehension ;

set_term =
    ( "{", "}" ) |
```

```
    ( "{", arg_seq, "}" );

list_term =
    ( "[", "]" ) |
    "[", term, list_tail;

list_tail =
    "]" |
    ( "|", term, "]" ) |
    ( ",..", "]" ) |
    ( ",..", term, "]" ) |
    ( term, ",", list_tail );

forget_remember_term =
    ( var | simple_compound ) |
    ( (var | simple_compound ), {",",  (var | simple_compound )}  ) |
    ( "[",  (var | simple_compound ), {",",  (var | simple_compound )}, "]");

pedro_handle = (atom | var), [":", (atom | var)], ["@", (atom | var)];
agent_handle = (atom | var), ["@", (atom | var)];

fork_size_list =
    ( "[", simple_compound, { ",", simple_compound }, "]" ) |
    ( simple_compound, { ",", simple_compound } ) ;

set_op = "union" | "diff" | "inter";

arith_op =
    "**" | "*" | "+" | "-" | "/" | "//" | ">>" | "<<" |
    "mod" | "rem" | "/\\" | "\\/";
```

Qulog Index

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W