Program Assertions

(Preliminary Version)
The CIAO System Documentation Series
Technical Report CLIP 4/97.1
Draft printed on: 15 September 1997
Version 0.5#5 (1997/9/12)

The CLIP Group
clip@dia.fi.upm.es
http://www.clip.dia.fi.upm.es/
Facultad de Informática
Universidad Politécnica de Madrid

Summary

This library provides modules which define a series of program assertions . These assertions can be used to document predicates and modules, from giving specifications (e.g., preconditions and post-conditions) for predicates and other program elements to adding machine-readable textual comments, suitable for automatic documentation generation.

This documentation corresponds to version 0.5#5 (1997/9/12).

1 Program Assertions (`assertions`)

This library provides modules which define a series of program assertions . These assertions can be used to document code elements (predicates, functions, properties, types, etc.), as well as whole modules, libraries or applications. The facilities provided by the library allow:

- Giving specifications (e.g., preconditions and post-conditions) for predicates and other program elements. This is supported by the `assertions/code` module.
- Adding machine-readable textual comments. This is supported mainly by the `assertions/comments` module, although some support for adding machine-readable comments is also provided by `assertions/code` .

All assertions are currently placed as directives in the source code, i.e., preceded by ":-".

1.1 Applications

The information given in these assertions can be used for a variety of purposes. For example, the specifications can be used to add run-time checks to the program in order to detect run-time violations of such specifications (see the documentation for the `rtchecks` library). Also, such information (or parts of it) can often be checked by the static analyzer , in some cases eliminating any need for run-time checks and in some other cases detecting violations of the specifications statically. In addition, these assertions can be used to provide information to the static analyzer, in order to guide it and also to control specialization (see the documentation for the `analyzer` library and `plai`). Finally, it is possible to generate quite reasonable program documentation automatically from the information provided in the assertions (see the documentation for the `autodoc` library and `pl2texi` script).

1.2 Some Attention Points

- **Formatting commands within text strings:** many of the predicates defined in these modules include arguments intended for providing textual information. This includes titles, descriptions, comments, etc. The type of this argument is a character string. In order for the automatic generation of documentation to work correctly, this character string should adhere to certain conventions. See the description of the `docstring/1` type/grammar for details.
- **Referring to variables:** In order for the automatic documentation system to work correctly, variable names (for example, when referring to arguments in the head patterns of *pred* declarations) must be surrounded by an `@var` command. For example, `@var{VariableName}` should be used for referring to the variable "VariableName", which will appear then formatted as follows: *VariableName*. See the description of the `docstring/1` type/grammar for details.

1.3 More Info

The facilities provided by the library are documented in the description of its component modules. This documentation is intended to provide information only at a "reference manual" level. For a more tutorial introduction to the subject and some more examples please see the document "An Assertion Language for Debugging of Constraint Logic Programs (Technical Report CLIP2/97.1)". The assertion language implemented in this library is modeled after this design document, although, due to implementation issues, it may differ in some details. The purpose of this manual is to document precisely what the implementation of the library supports at any given point in time.

1.4 Known bugs and planned improvements (`assertions`)

- The assertion design and the documentation are still in some flux.
- This documentation is still in a preliminary stage.

1.5 Change Log (`assertions`)

Version 0*5#5 (1997/9/12)
Original library split into two parts: formal (code) assertions and text-related assertions. (Manuel Hermenegildo)

2 Code-Related Assertions (code)

This module is part of the `assertions` library. It defines the basic code-related assertions, i.e., those intended to be used mainly by compilation-related tools, such as the static analyzer or the run-time test generator.

2.1 Usage and interface (code)

- **Library Usage:**

  `:- use_module(library(code))`
- **Exports:**

  `(pred)/1`, `(func)/1`, `check/1`, `(prop)/1`, `(type)/1`, `p_assrt_body/1`, `head_pattern/1`, `complex_arg_property/1`, `property_conjunction/1`, `property_starterm/1`, `(calls)/1`, `(success)/1`, `(comp)/1`, `(true)/1`, `(false)/1`, `(entry)/1`, `(trust)/1`, `c_assrt_body/1`, `s_assrt_body/1`, `g_assrt_body/1`, `predfunctor/1`, `funcfunctor/1`, `propfunctor/1`, `typefunctor/1`, `n_assrt_body/1`.
- **Exported operator definitions:**

  `=>/2` [975,xfx], `?/1` [500,fx], `@/1` [500,fx], `pred/1` [1150,fy], `func/1` [1150,fy], `prop/1` [1150,fy], `type/1` [1150,fy], `calls/1` [1150,fy], `success/1` [1150,fy], `comp/1` [1150,fy], `true/1` [1150,fy], `false/1` [1150,fy], `entry/1` [1150,fy], `trust/1` [1150,fy].
- **Other modules used:**

  `library(assertions/prolog_dependent)`.

2.2 Documentation on exports (code)

- *Predicate* **(pred)/1:**

  `pred AssertionBody`
  - *Description:* This assertion provides information on a predicate. The body of the assertion (*Body*) contains properties or comments in the formats defined by `p_assrt_body/1` .

    More than one of these assertions may appear per predicate, in which case each one represents a possible "mode " of use (usage) of the predicate. The exact scope of the usage is defined by the properties given for calls in the body of each assertion (which should thus distinguish the different usages intended).

    Some examples of valid assertions:

    A basic assertion for `s/2`, stating that the second argument should be a var on input and the predicate will produce a ground list of integers without failing:

    ```
    :- pred s(A,B) : var(B) => (list(B,integer),ground(B)) +  no_fail.
    ```

    Same as above, but adding a textual comment (see `assertions/comments` for more details on the format of these comments and other possibilities for commenting code):

    ```
    :- pred s(A) :  var(A) => (list(A,integer),ground(A)) + no_fail
       ; "Not a very useful predicate.".
    ```

    Same as above, but using the compact argument notation:

```
:- pred s/2 : var * any => list(integer) * ground +  no_fail
   ; "Not a very useful predicate.".
```

Other examples:

```
:- pred p/3                  :  list * integer * var
                             => list * integer * list
   +  (iso,no_fail)
   ;  "This mode is good.".
:- pred p(Preds,Value,Assoc) :  list * var * var
                             => list * integer * list
   +  no_fail
   ;  "This mode is bad.".
:- pred p/3                  => list * integer * list
   +  (no_fail,no_fail)
   ; "This one just OK.".

:- pred q(A) : list(A) => (list(A),ground(A)) + no_fail ; "Foo".
:- pred q(A) ; "Not a bad use at all.".

:- pred q/2 : var * {ground,integer} => {ground,integer} * integer;"A test.'

:- pred p/1 : var => list.

:- pred t(+A,+B,?C,@D,-E) : list(A) => (list(A),ground(A)) + no_fail.
```

– *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **(func)/1:**

  `func AssertionBody`

  – *Description:* This assertion provides information on a function.

  – *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **check/1:**

  `check(PropertyConjunction)`

  – *Description:* This assertion provides information on a clause program point (position in the body of a clause). Calls to a `check/1` assertion can appear in the body of a clause in any place where a literal can normally appear. The property defined by *PropertyConjunction* should hold in all the run-time stores corresponding to that program point.

  – *Properties required at call time:* `property_conjunction(PropertyConjunction)`: *PropertyConjunction* is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument.

- *Predicate* **(prop)/1:**

  `prop AssertionBody`

  – *Description:* This assertion is similar to a pred assertion but it flags that the predicate being documented is also a "property ." Properties are standard predicates, but which are *deterministic* and *guaranteed to terminate for any possible instantiation state of their argument(s)* (and can thus be safely used as run-time checks).

The set of properties is thus a strict subset of the set of predicates. Note that properties can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

  - *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **(type)/1:**

  `type AssertionBody`

  - *Description:* This assertion is similar to a pred assertion but it flags that the predicate being documented is also a "type ." This allows for example checking whether it is in the class of types supported by the type checking and inference modules. Currently, types are properties whose definitions are *regular programs*.

    The set of types is a thus well defined subset of the set of properties. Note that types can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

  - *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Type* **p_assrt_body/1:**

  This predicate defines the different types of syntax admissible in the bodies of `pred/1` , `func/1` , etc. assertions. The following are admissible:

```
Pr : CP => AP + GP ; CO
Pr : CP => AP + GP
Pr : CP => AP ; CO
Pr : CP => AP
Pr : CP + GP ; CO
Pr : CP + GP
Pr : CP ; CO
Pr : CP
Pr => AP + GP ; CO
Pr => AP + GP
Pr => AP ; CO
Pr => AP
Pr + GP ; CO
Pr + GP
Pr ; CO
Pr
```

  where:

  - *Pr* is a head pattern (`head_pattern/1`) which describes the predicate/property/type and possibly gives some implicit call/answer information.
  - *CP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
  - *AP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
  - *GP* contains (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
  - *CO* is a comment string . This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `assrt_b_args/5` and its auxiliary types.

`p_assrt_body(X)`

- *Description:* X is a predicate assertion body.

- *Property* **head_pattern/1:**

  A head pattern can be a predicate name (functor/arity) (`predname/1`) or a term. Thus, both `p/3` and `p(A,B,C)` are valid head patterns. In the case in which the head pattern is a term, each argument of such a term can be:

  - A variable. This is useful in order to be able to refer to the corresponding argument positions by name within properties and in comments. Thus, `p(Input,Parameter,Output)` is a valid head pattern.
  - A ground term. In this case this term determines a property of the corresponding argument. The actual property referred to is that given by the term but with one more argument added at the beginning, which is a new variable which, in a rewriting of the head pattern, appears at the argument position occupied by the term. Unless otherwise stated (see below), the property built this way is understood to hold for both calls and answers. For example, the head pattern `p(Input,list(integer),Output)` is valid and equivalent for example to having the a head pattern `p(Input,A,Output)` and stating that the property `list(A,integer)` holds for the calls and successes of the predicate.
  - Finally, it can also be a variable or a ground term, as above, but preceded by a "mode ." This mode determines in a compact way certain call or answer properties. Acceptable modes are `+` (`nonvar/1` at call time), `-` (`var/1` at call time), `@` (`not_further_inst/1`), and `?` (no information).

  `head_pattern(Pr)`

  - *Description:* Pr is a head pattern.

- *Type* **complex_arg_property/1:**

  `complex_arg_property(Props)`

  - *Description:* $Props$ is a (possibly empty) complex argument property . Such properties can appear in two formats, which are defined by the types `property_conjunction/1` and `property_starterm/1` respectively. The two formats can be mixed provided they are not in the same field of an assertion. I.e., the following is a valid assertion:

    ```
    :- pred foo(X,Y) : nonvar * var => (ground(X),ground(Y)).
    ```

- *Type* **property_conjunction/1:**

  This type defines the first, unabridged format in which properties can be expressed in the bodies of assertions. It is essentially a conjunction of properties which refer to variables. The following is an example of a complex property in this format:

  - `(integer(X),list(Y,integer))`: X has the property `integer/1` and Y has the property `list/2` , with second argument `integer`.

  `property_conjunction(Props)`

  - *Description:* $Props$ is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument.

- *Type* **property_starterm/1:**

  This type defines the a second, compact format in which properties can be expressed in the bodies of assertions. A similar series of properties as in `property_conjunction/1` appears, but their arity is one less than the corresponding argument definitions. The number of properties is the same as the arity of the predicate being documented. They are connected by `*/2` instead of conjunction. The idea is that each element of the `*/2` term corresponds to a head argument position. Several properties can be assigned to each argument position by grouping them in curly brackets. The following is an example of a complex property in this format:

- `integer * list(integer)`: the first argument of the procedure (or function, or ...) has the property `integer/1` and the second one has the property `list/2` , with second argument `integer`.
- `{integer,var} * list(integer)`: the first argument of the procedure (or function, or ...) has the properties `integer/1` and `var/1` and the second one has the property `list/2` , with second argument `integer`.

`property_starterm(Props)`

- *Description: Props* is either a term or several terms separated by `*/2` . The main functor of each of those terms corresponds to that of the definition of a property, and the arity such be one less than in the definition of such property. All arguments of each such term are ground.

- *Predicate* **(calls)/1:**

  `calls AssertionBody`

  - *Description:* This assertion is similar to a `pred/1` assertion but it only provides information about the calls to a predicate.
  - *Properties required at call time:* `c_assrt_body(AssertionBody)`: *AssertionBody* is a call assertion body.

- *Predicate* **(success)/1:**

  `success AssertionBody`

  - *Description:* This assertion is similar to a `pred/1` assertion but it only provides information about the answers to a predicate.
  - *Properties required at call time:* `s_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **(comp)/1:**

  `comp AssertionBody`

  - *Description:* This assertion is similar to a `pred/1` assertion but it only provides information about the global execution properties of a predicate.
  - *Properties required at call time:* `g_assrt_body(AssertionBody)`: *AssertionBody* is a comp assertion body.

- *Predicate* **(true)/1:**

  `true AssertionBody`

  - *Description:* This assertion is identical syntactically to a `pred/1` assertion. However, the properties stated have been proved to hold by the analyzer. Thus, these assertions often represent the analyzer output .
  - *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **(false)/1:**

  `false AssertionBody`

  - *Description:* This assertion is identical syntactically to a `pred/1` assertion. However, the properties stated have been proved not to hold by the analyzer. Thus, these assertions often represent the analyzer output .
  - *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

- *Predicate* **(entry)/1:**

  `entry AssertionBody`

- *Description:* This assertion provides information about the calls to a predicate. It is identical syntactically to a `calls/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. As a result, if these assertions are erroneous they can introduce bugs in programs. Thus, `entry/1` assertions should be written with care.

  An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program. The main use is in providing information on the ways in which exported predicates of a module will be called from outside the module. This will greatly improve the precision of the analyzer, which otherwise has to assume that the arguments that exported predicates receive are any arbitrary term.
- *Properties required at call time:* `c_assrt_body(AssertionBody)`: *AssertionBody* is a call assertion body.

• *Predicate* **(trust)/1:**

`trust AssertionBody`

- *Description:* This assertion also provides information on a predicate. It is identical syntactically to a `pred/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. While the compiler may in some cases detect an inconsistency between a `trust/1` assertion and the program, in all other cases the information given in the assertion will be taken to be true. As a result, if these assertions are erroneous they can introduce bugs in programs. Thus, `trust/1` assertions should be written with care.

  An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program (either because the information is not present or because the analyzer being used is not precise enough). In particular, providing information on external predicates which may not be accessible at the time of compiling the module can greatly improve the precision of the analyzer. This can be easily done with trust assertion.
- *Properties required at call time:* `p_assrt_body(AssertionBody)`: *AssertionBody* is a predicate assertion body.

• *Type* **c_assrt_body/1:**

This predicate defines the different types of syntax admissible in the bodies of `call/1` , `entry/1` , etc. assertions. The following are admissible:

```
Pr : CP ; CO
Pr : CP
```

where:

- *CP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- *CO* is a comment string . This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

`c_assrt_body(X)`

- *Description:* *X* is a call assertion body.

• *Type* **s_assrt_body/1:**

This predicate defines the different types of syntax admissible in the bodies of `pred/1` , `func/1` , etc. assertions. The following are admissible:

```
Pr : CP => AP ; CO
Pr : CP => AP
Pr => AP ; CO
Pr => AP
```

where:

- *Pr* is a head pattern (`head_pattern/1`) which describes the predicate/property/type and possibly gives some implicit call/answer information.
- *CP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- *AP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
- *CO* is a comment string . This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

`s_assrt_body(X)`

– *Description:* X is a predicate assertion body.

- *Type* **g_assrt_body/1:**

This predicate defines the different types of syntax admissible in the bodies of `comp/1` assertions. The following are admissible:

```
Pr : CP + GP ; CO
Pr : CP + GP
Pr + GP ; CO
Pr + GP
```

where:

- *Pr* is a head pattern (`head_pattern/1`) which describes the predicate/property/type and possibly gives some implicit call/answer information.
- *CP* is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- *GP* contains (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- *CO* is a comment string . This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

`g_assrt_body(X)`

– *Description:* X is a comp assertion body.

- *Type* **predfunctor/1:**

`predfunctor(X)`

– *Description:* X is a the main functor of a predicate class assertion (includes `pred`, `prop`, `type`).

- *Type* **funcfunctor/1:**
  `funcfunctor(X)`
  - *Description:* X is a the main functor of a function class assertion (`func`).
- *Type* **propfunctor/1:**
  `propfunctor(X)`
  - *Description:* X is a the main functor of a property class assertion (includes `prop`, `type`).
- *Type* **typefunctor/1:**
  `typefunctor(X)`
  - *Description:* X is a the main functor of a type class assertion (includes `type`).

2.3 Change Log (`code`)

Version 0*5#6 (1997/9/15)
Added check/1. (Manuel Hermenegildo)

Version 0*5#5 (1997/9/12)
Original library split into two parts: formal assertions and text. (Manuel Hermenegildo)

Version 0*5#4 (1997/8/11)
Improved documentation. (Manuel Hermenegildo)

Version 0*5#3 (1997/8/11)
Added more precise definition of docstring syntax. (Manuel Hermenegildo)

3 Documentation-Oriented Assertions (comments)

This module is part of the assertions library. It defines some additional assertions, mainly for adding additional textual comments to a module or application. The intention is that these be used instead of (or in addition to) the usual comments included in the program text. The content of these assertions can be used for automatic documentation of the code.

Examples:

```
:- comment(title,"Documentation-Oriented Assertions").

:- comment(version(0*5#5,1997/9/12), "Original library split into two
   parts: formal assertions and text. (Manuel Hermenegildo)").

:- comment(module,"This module is part of the assertions
   library.").

:- comment(comment/2,"This pseudo-assertion provides one of the main
   means for adding machine readable comments
 to
   programs.").
```

3.1 Usage and interface (comments)

- **Library Usage:**
  `:- use_module(library(comments))`
- **Exports:**
  `comment/2`, `predname/1`, `version_descriptor/1`, `ymd_date/1`, `version_number/1`, `docstring/1`, `stringcommand/1`.
- **Other modules used:**
  `library(assertions/prolog_dependent)`.

3.2 Documentation on exports (comments)

- *Predicate* **comment/2:**
  This pseudo-assertion provides one of the main means for adding machine readable comments to programs. Such comments are meant to be complementary to the formal statements present in the actual assertions. These comments can be used to document the program in the same way as traditional comments, but, in addition, they can be used to generate printed or on-line documentation automatically (see the autodoc library and the pl2texi automatic documentation generator manual for more details).
  Usage 1: `comment(Type,Text)`

- *Description:* Provides the title of the module or application. When generating documentation for the module automatically, the text in *Text* will be used as the title of the corresponding chapter or manual. If not provided, the module or file name are used instead as title. There should be at most one of these assertions per module.
- *Properties required at call time:* `Type=title`: *Type* unifies with *title*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 2: `comment(Type,Text)`

- *Description:* Provides the author (s) of the module or application. If present, when generating documentation for the module automatically, the text in *Text* will be placed in the corresponding chapter or front page. There can be more than one of these assertions per module (normally, one per author).
- *Properties required at call time:* `Type=author`: *Type* unifies with *author*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 3: `comment(Type,Text)`

- *Description:* Provides subtitle lines. This can be, e.g., an explanation of the application to add to the title, the address of the author(s) of the application, etc. When generating documentation automatically, the text in *Text* will be used accordingly. Several of these assertions can appear per module, which is useful for, e.g., multiple line addresses.
- *Properties required at call time:* `Type=subtitle`: *Type* unifies with *subtitle*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 4: `comment(Type,Text)`

- *Description:* Provides a copyright text. This normally appears somewhere towards the beginning of a printed manual.
- *Properties required at call time:* `Type=copyright`: *Type* unifies with *copyright*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 5: `comment(Version,Text)`

- *Description:* Provides a means for keeping a log of changes . *Version* contains the version number and date corresponding to the change and *Text* an explanation of the change. Several of these assertions can appear per module. When generating documentation automatically, the text in the *Text* fields may appear as items in an itemized list of changes. Version comments should appear near the beginning of the the file and should appear in reverse chronological order (i.e., the topmost comment is the most recent one). The CIAO `emacs` mode helps tracking version numbers by prompting for version comments when files are saved.
- *Properties required at call time:* `version_descriptor(Version)`: *Version* is a version(Number,Date) structure denoting a complete version description. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 6: `comment(Type,Text)`

- *Description:* Provides an introductory comment to the module or application. When generating documentation automatically, the text in *Text* will be used as the introduction of the corresponding chapter or manual. There should be at most one of these assertions per module.
- *Properties required at call time:* `Type=module`: *Type* unifies with *module*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 7: `comment(Type,Text)`

- *Description:* Provides a very brief global explanation intended as an introduction for the manual of an application or library. The text in *Text* will be used as the abstract for the whole manual. There should be at most one of these assertions per module.
- *Properties required at call time:* `Type=summary`: *Type* unifies with *summary*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 8: `comment(PredName,Text)`

- *Description:* Provides an introductory comment for a given predicate, function, property, type, etc., denoted by *PredName*. When generating documentation for the module automatically, the text in *Text* will be used as the introduction of the corresponding predicate/function/... description. There should be at most one of these assertions per predicate, function, property, or type.
- *Properties upon exit:* `predname(PredName)`: *PredName* is a structure Name/Arity denoting a predicate name. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

Usage 9: `comment(Type,Text)`

- *Description:* Documents a known bug or planned improvement in the module or application. Several of these assertions can appear per module. When generating documentation automatically, the text in the *Text* fields will be used as items in an itemized list of module or application bugs.
- *Properties required at call time:* `Type=bug`: *Type* unifies with *bug*. `docstring(Text)`: *Text* is a character string possibly with embedded formatting commands.

- *Type* **predname/1:**

  `predname(PredName)`

  - *Description:* *PredName* is a structure Name/Arity denoting a predicate name.

- *Type* **version_descriptor/1:**

  `version_descriptor(Descriptor)`

  - *Description:* *Descriptor* is a version(Number,Date) structure denoting a complete version description.

- *Type* **ymd_date/1:**

  `ymd_date(Date)`

  - *Description:* *Date* is a Year/Month/Day structure denoting a date.

- *Type* **version_number/1:**

  `version_number(Version)`

  - *Description:* *Version* is a Major*Minor#Patch structure denoting a complete version number.

- *Type* **docstring/1:**

  Defines a type of character string which includes the embedding of certain formatting commands .

  - All printable characters are admissible except "`@`", "`{`," and "`}`". To produce these characters the following escape sequences should be used, respectively: `@@`, `@{`, and `@}`.
  - In order to allow better formatting of on-line and printed manuals, in addition to normal text, certain formatting commands can be used within these strings. The syntax of all these commands is `@`*command*`{`*body*`}`, where *command* is the command name and *body* is the (possibly empty) command body, or `@`*command* (followed by a space or `{}`). The set of commands currently admitted can be found in the documentation for the predicate `stringcommand/1` .

`docstring(CO)`

- *Description:* *CO* is a character string possibly with embedded formatting commands.

- *Property* **stringcommand/1:**

Defines a structure resulting from parsing a formatting command admissible in comment strings inside assertions.

In order to make it possible to produce documentation in a wide variety of formats, the command set is kept small. The names of the commands are intended to be reminiscent of the commands used in the LaTeX text formatting system, except that "`@`" is used instead of "`\`." Note that \ would need to be escaped in ISO-Prolog strings, which would make the source less readable (and in any case, many ideas in LaTeX were taken from `scribe`, where the escape character was indeed `@`!).

The following are the currently admissible commands:

- `@concept{`*text*`}`: *text* will be printed as is and also included in the concept index.
- `@cindex{`*text*`}`: *text* will included in the concept index (it is not printed).
- `@pred{`*predname*`}`: *predname* (which should be in functor/arity form) is the name of a predicate and will be printed in typewriter-like font, and also included in the predicate index.
- `@op{`*operatorname*`}`: *operatorname* (which should be in functor/arity form) is the name of an operator and will be printed in typewriter-like font, and also included in the operator index.
- `@lib{`*libname*`}`: *libname* is the name of a library and will be printed in fixed-width, typewriter-like font, and also included in the library index.
- `@begin{itemize}`: marks the beginning of an itemized list. Each item should be in a separate paragraph and preceded by an `@item{}` command.
- `@item{}`: marks the beginning of a new item in an itemized list.
- `@end{itemize}`: marks the end of an itemized list.
- `@begin{verbatim}`: marks the beginning of formatted text, such as a program example. A fixed-width, typewriter-like font is used.
- `@end{verbatim}`: marks the end of formatted text.
- `@section{`*text*`}`: starts a section whose title is *text*.
- `@subsection{`*text*`}`: starts a subsection whose title is *text*.
- `@today{}`: prints the current date.
- `@hfill{}`: introduces horizontal filling space (may be ignored in certain formats).
- `@copyright{}`: prints the copyright symbol in some appropriate form.
- `@bf{`*text*`}`: *text* will be formatted in bold or any strong font.
- `@em{`*text*`}`: *text* will be formatted in italics or any other emphasis font.
- `@tt{`*text*`}`: *text* will be formatted in a fixed-width, typewriter-like font.
- `@var{`*varname*`}`: *varname* is the name of a variable and will be formatted in an emphasized font. Note that when referring to variable names in a `pred/1` declaration, such names should be enclosed in `@var` commands for the automatic documentation system to work correctly.
- `@include{`*filename*`}`: the contents of *filename* will be included in-line, as if they were part of the string. This is useful for common pieces of documentation or storing in a separate file long explanations if they are perceived to clutter the source file.
- `@includeverbatim{`*filename*`}`: as above, but the contents of the file are included verbatim, i.e., commands within the file are not interpreted. This is useful for including code examples which may contain `@`.

- *Accents and special characters (to simplify things during development of the library "*" is printed in place of characters not supported by our current version of texinfo)*:

  @‘{o}: ò, @’{o}: ó, @^{o}: ô, @..{o}: ö, @~{o}: õ, @={o}: ō, @.{o}: *, @u{o}: ŏ, @v{o}: ǒ, @H{o}: ő, @t{oo}: **, @c{o}: *, @d{o}: *, @b{o}: *, @oe{}: œ, @OE{}: Œ, @ae{}: æ, @AE{}: Æ, @aa{}: å, @AA{}: Å, @o{}: ø, @O{}: Ø, @l{}: ł, @L{}: Ł, @ss{}: ß, @?{}: *, @!{}: *, @i{}: *, @j{}: *.

`stringcommand(CO)`

- *Description: CO* is a structure denoting a command that is admissible in strings inside assertions.

3.3 Change Log (comments)

Version 0*5#5 (1997/9/12)

Original library split into two parts: formal assertions and text. (Manuel Hermenegildo)

4 Some basic properties and types (basictypes)

This library contains a basic set of type and property definitions. They are intended for use both as type testing builtins within programs and as properties in assertions.

4.1 Usage and interface (basictypes)

- **Library Usage:**
  `:- use_module(library(basictypes))`
- **Exports:**
  `(=)/2`, `any/1`, `atom/1`, `filename/1`, `formatstring/1`, `ground/1`, `integer/1`, `iso/1`, `list/1`, `list/2`, `no_fail/1`, `predname/1`, `nonvar/1`, `not_further_inst/1`, `structofstrings/1`, `stream/1`, `string/1`, `(true)/0`, `term/1`, `var/1`.
- **Exported operator definitions:**
  `=/1` [500,fx].
- **Other modules used:**
  `library(assertions)`.

4.2 Documentation on exports (basictypes)

- *Property* **(=)/2:**
  `X=Y`
  - *Description:* X unifies with Y.
- *Type* **any/1:**
  This is equivalent to `term/1` . Included because this name is frequently used. `any(X)`
  - *Description:* No restrictions on X
- *Type* **atom/1:**
  `atom(X)`
  - *Description:* X is an atom.
- *Type* **filename/1:**
  `filename(X)`
  - *Description:* X is an atom describing the name of a file.
- *Type* **formatstring/1:**
  `formatstring(String)`
  - *Description:* *String* is a character string with formatting characters (as needed by `format/2`).
- *Property* **ground/1:**
  `ground(X)`
  - *Description:* X is ground.
- *Type* **integer/1:**
  `integer(X)`
  - *Description:* X is of type integer.

- *Property* **iso/1:**
  `iso(G)`
  - *Description: G complies with the ISO-Prolog standard.*
- *Type* **list/1:**
  `list(X)`
  - *Description:* X is of type list.
- *Type* **list/2:**
  `list(X,Y)`
  - *Description:* X is a list of Ys.
- *Property* **no_fail/1:**
  `no_fail(G)`
  - *Description:* Calls to G do not fail.
- *Type* **predname/1:**
  `predname(PredName)`
  - *Description: PredName* is a structure Name/Arity denoting a predicate name.
- *Property* **nonvar/1:**
  `nonvar(T)`
  - *Description:* T is a term which is not a free variable.
- *Property* **not_further_inst/1:**
  `not_further_inst(T)`
  - *Description:* T is not further instantiated w.r.t. the call.
- *Type* **structofstrings/1:**
  `structofstrings(S)`
  - *Description:* S is a structure of strings.
- *Type* **stream/1:**
  `stream(StreamId)`
  - *Description: StreamId* is a stream identifier.
- *Type* **string/1:**
  `string(String)`
  - *Description: String* is a character string.
- *Type* **term/1:**
  The most general type (includes all possible terms). `term(X)`
  - *Description:* X is any term.
- *Property* **var/1:**
  `var(X)`
  - *Description:* X is a free variable.

4.3 Known bugs and planned improvements (`basictypes`)

- The solution of putting all these predicates here is temporary. Most of these should be in the files where the corresponding builtins are defined in the system (e.g., builtins.pl, etc.). This file is useful while we finish adding declarations to the system library files, and also for running under other Prolog systems.

4.4 Change Log (basictypes)

Version 0*5#4 (1997/8/27)
Included type term/1 (Daniel Cabeza Gras)

Version 0*5#3 (1997/7/29)
Added unary '=' (quote in types).

5 Assertion processing library (lib)

This module defines some predicates which are useful for writing programs which process assertions (as defined in the assertions library). The exported predicates allow reading assertions in all acceptable syntactic forms and converting them to a normalized format.

5.1 Usage and interface (lib)

- **Library Usage:**
  `:- use_module(library(lib))`
- **Exports:**
  `get_directives/2`, `get_related_directives/2`, `normalize_assertions/2`.
- **Other modules used:**
  `library(assertions)`, `library(messages)`, `library(basictypes)`, `library(lists)`.

5.2 Documentation on exports (lib)

- *Predicate* **get_directives/2:**
  `get_directives(IS,Ds)`
  - *Description:* Reads all directives in *IS* and gathers them in *Ds*. Any assertions found are normalized before adding them to *Ds*. Will also work on scripts (the first two lines are then ignored).
  - *Properties required at call time:* `stream(IS)`: *IS* is a stream identifier.
  - *Properties upon exit:* `list(Ds)`: *Ds* is of type list.
- *Predicate* **get_related_directives/2:**
  `get_related_directives(Files,RelDirs)`
  - *Description:* *RelDirs* contains the directives in all the files related (by `use_module/1`) to the files in *Files*.
  - *Properties required at call time:* `list(Files)`: *Files* is of type list.
  - *Properties upon exit:* `list(RelDirs)`: *RelDirs* is of type list.
- *Predicate* **normalize_assertions/2:**
  `normalize_assertions(ID,OD)`
  - *Description:* The assertion-related directives in *ID* are in canonical format in *OD*.
  - *Properties required at call time:* `list(ID)`: *ID* is of type list.
  - *Properties upon exit:* `list(OD)`: *OD* is of type list.

5.3 Known bugs and planned improvements (lib)

- The “`@`” argument mode is supported in a roundabout way.

5.4 Change Log (`lib`)

Version 0*5#5 (1997/8/15)
Changed to ground representation. Vars are '$var(Name)', avoided dictionary.

Version 0*5#4 (1997/7/31)
Added syntax support for expressing properties directy in the head.

Version 0*5#3 (1997/8/11)
Added support for multi-argument properties in compact format.

Version 0*5#2 (1997/7/21)
Added handling of ISO standard-like modes.

Concept Index

Module/Object Definition Index

B

C

Predicate/Method Definition Index

(

C

G

N

Property Definition Index

(

G

H

I

N

S

V

Type Definition Index

Operator Definition Index

Table of Contents