

# The IDP system reference manual

Johan Wittocx

December 23, 2011

## 1 Installing And Running

[TODO: Broes?](#) commandline options...

## 2 Comments

Everything between `/*` and `*/` is a comment, as well as everything between `//` and the end of the line. If a comment block starts with `/**`, but not with `/***`, then the comment is added as a description to the first thing after that comment block that can have a description. Currently, only procedures can have a description.

## 3 Include statements

Everywhere in an IDP file, a statement

```
#include "path/to/file"
```

is replaced by the contents of the file `path/to/file`. A statement

```
#include <filename>
```

is replaced by the contents of the standard library file `filename`. Currently the following standard library files are available:

**mx** Contains some useful model expansion procedures.

**theory** Contains procedures to transform theories.

**domain** Contains procedures to create ranges.

## 4 Namespaces

A namespace with name `MySpace` is declared by

```
namespace MySpace {  
    // content of the namespace  
}
```

An object with name `MyName` declared in namespace `MySpace` can be referred to by `MySpace::MyName`. Inside `MySpace`, `MyName` can simply be referred to by `MyName`.

A Namespace can contain namespaces, vocabularies, theories, structures, procedures, options and using statements. A using statement is of one of the following forms

```
using namespace MySpace
using vocabulary MyVoc
```

where `MySpace` is the name of a namespace, and `MyVoc` the name of a vocabulary. Below such a using statement, objects `MyObj` declared in `MySpace`, respectively `MyVoc`, can be referred to by `MyObj`, instead of `MySpace::MyObj`, respectively `MyVoc::MyObj`.

Every object that is declared outside a namespace, is considered to be part of the global namespace. The name of the global namespace is `global_namespace`. In other words, every IDP file implicitly starts with `namespace global_namespace {` and ends with `}.`

## 5 Vocabularies

A vocabulary with name `MyVoc` is declared by

```
vocabulary MyVoc {
    // contents of the vocabulary
}
```

A vocabulary can contain symbol declarations, symbol pointers, and other vocabularies. Symbols are types (sorts), predicate and functions symbols.

### 5.1 Symbol declarations

A type with name `MyType` is declared by

```
type MyType
```

When declaring a type, it can be stated that this type is a subtype or supertype of a set of other types. The following declares `MyType` to be a subtype of the previously declared types `A1` and `A2`, and a supertype of the previously declared types `B1` and `B2`:

```
type MyType isa A1, A2 contains B1, B2
```

A predicate with name `MyPred` and types `T1,T2,T3` is declared by

```
MyPred(T1, T2, T3)
```

A predicate with arity zero can be declared by `MyPred()` or `MyPred`.

A function with name `MyFunc`, input types `T1,T2,T3` and output type `T` is declared by

```
MyFunc(T1, T2, T3) : T
```

A partial function is declared by

```
partial MyFunc(T1, T2, T3) : T
```

Constants of type `T` can be declared by `MyConst:T` or `MyConst():T`. Besides functions with an identifier as name, functions of arity two with names `+`, `-`, `*`, `/`, `%` and `^` can be declared, as well as unary functions with names `-` and `abs`.

## 5.2 Symbol pointers

To include a type, predicate, or function from a previously declared vocabulary  $V$  in another vocabulary  $W$ , write

```
/* Declaration of vocabulary V*/
vocabulary V{
  //...
  type A
  P(A)
  F(A,A):A
  //...
}

vocabulary W{
  extern type V::A
  extern V::P[A] //also possible: extern V::P/1
  extern V::F[A,A:A]    also possible: extern V::F/2:1
}
```

In the example, explicitly including type  $A$  of vocabulary  $V$  in  $W$  is not needed, since types of included predicates or functions are automatically included themselves. To include the whole vocabulary  $V$  in  $W$  at once, used

```
vocabulary W{
  extern vocabulary V
}
```

## 5.3 The standard vocabulary

The global namespace contains a fixed vocabulary `std`, which is defined as follows:

```
vocabulary std{
  type nat
  type int contains nat
  type float contains int
  type char
  type string contains char

  +(int,int) : int
  -(int,int) : int
  *(int,int) : int
  /(int,int) : int
  %(int,int) : int
  abs(int) : int
  -(int) : int

  +(float,float) : float
}
```

```

-(float,float) : float
*(float,float) : float
/(float,float) : float
^(float,float) : float
abs(float) : float
-(float) : float
}

```

Every vocabulary implicitly contains all symbols of **std**. Also, every vocabulary contains for each of its types **A** the predicates  $=(\mathbf{A},\mathbf{A})$ ,  $<(\mathbf{A},\mathbf{A})$ , and  $>(\mathbf{A},\mathbf{A})$  and the functions  $\text{MIN}:\mathbf{A}$ ,  $\text{MAX}:\mathbf{A}$ ,  $\text{SUCC}(\mathbf{A}):\mathbf{A}$  and  $\text{PRED}(\mathbf{A}):\mathbf{A}$ . In every structure, the symbols of **std** have the following interpretation:

|   |                               |
|---|-------------------------------|
| <b>nat</b>                                    | all natural numbers           |
| <b>int</b>                                    | all integer numbers           |
| <b>float</b>                                  | all floating point numbers    |
| <b>char</b>                                   | all characters                |
| <b>string</b>                                 | all strings                   |
| $+(\text{int},\text{int}) : \text{int}$       | integer addition              |
| $-(\text{int},\text{int}) : \text{int}$       | integer subtraction           |
| $*(\text{int},\text{int}) : \text{int}$       | integer multiplication        |
| $/(\text{int},\text{int}) : \text{int}$       | integer division              |
| $\%(\text{int},\text{int}) : \text{int}$      | remainder                     |
| $\text{abs}(\text{int}) : \text{int}$         | absolute value                |
| $-(\text{int}) : \text{int}$                  | unary minus                   |
| $+(\text{float},\text{float}) : \text{float}$ | floating point addition       |
| $-(\text{float},\text{float}) : \text{float}$ | floating point subtraction    |
| $*(\text{float},\text{float}) : \text{float}$ | floating point multiplication |
| $/(\text{float},\text{float}) : \text{float}$ | floating point division       |
| $^(\text{float},\text{float}) : \text{float}$ | floating point exponentiation |
| $\text{abs}(\text{float}) : \text{float}$     | absolute value                |
| $-(\text{float}) : \text{float}$              | unary minus                   |

The predicate  $=/2$  is always interpreted by equality. The order  $<_{dom}$  on domain elements is defined by

- numers are smaller than non-numers;
- strings are smaller than compound domain elements (see below for a definitions of a compound domain element);
- $d_1 <_{dom} d_2$  if  $d_1$  and  $d_2$  are numbers and  $d_1 < d_2$ ;
- $d_1 <_{dom} d_2$  if  $d_1$  and  $d_2$  are strings that are not numbers and  $d_1$  is before  $d_2$  in the lexicographic ordering;
- $d_1 <_{dom} d_2$  is some total order on compound domain elements (which we do not specify).

Every structure contains the following fixed interpretations:

|             |   |
|-------------|---|
| $<(A, A)$   | the projection of $<_{dom}$ to the domain of $A$  |
| $>(A, A)$   | the projection of $>_{dom}$ to the domain of $A$  |
| $MIN:A$     | the $<_{dom}$ -least element in the domain of $A$   |
| $MAX:A$     | the $<_{dom}$ -greatest element in the domain of $A$  |
| $SUCC(A):A$ | the partial function that maps an element $a$ of the domain of $A$<br>to the $<_{dom}$ -least element of the domain of $A$ that is strictly larger than $a$     |
| $PRED(A):A$ | the partial function that maps an element $a$ of the domain of $A$<br>to the $<_{dom}$ -greatest element of the domain of $A$ that is strictly smaller than $a$ |

## 6 Theories

A theory with name `MyTheory` over a vocabulary `MyVoc` is declared by

```
theory MyTheory : MyVoc{
    // contents of the theory
}
```

A theory contains sentences, inductive definitions, and fixpoint definitions.

### 6.1 Sentences

#### 6.1.1 Terms

Before explaining the syntax for sentences, we need to introduce the concept of a term and a formula. We also give the syntax for terms and formulas in IDP.

A *term* is inductively defined as follows:

- a variable is a term;
- a constant is a term;
- if  $F$  is a function symbol with  $n$  input arguments and  $t_1, \dots, t_n$  are terms, then  $F(t_1, \dots, t_n)$  is a term.

In IDP, variables start with a letter and may contain letters, digits and underscores. When writing a term in IDP, the constant and function symbols occurring in that term should be declared before. The *type of a term* is defined as its return type (see section 5.1) in the case of constants and functions. The type of a variable is derived from its occurrences in formulas (see section 6.6). If a term occurs in an input position of a function, then the type of the term and the type of the input position must have a common ancestor type.

#### 6.1.2 Formulas and Sentences

A *formula* is inductively defined by:

- **true** and **false** are formulas;
- if  $P$  is a predicate symbol with arity  $n$  and  $t_1, \dots, t_n$  are terms, then  $P(t_1, \dots, t_n)$  is a formula;
- if  $t_1$  and  $t_2$  are terms, then  $t_1 = t_2$  is a formula;

- if  $\varphi$  and  $\psi$  are formulas and  $x$  is a variable, then the following are formulas:  $\neg\varphi$ ,  $\varphi \wedge \psi$ ,  $\varphi \vee \psi$ ,  $\varphi \Rightarrow \psi$ ,  $\varphi \Leftarrow \psi$ ,  $\varphi \equiv \psi$ ,  $\forall x \varphi$ , and  $\exists x \varphi$ .

The following order of binding is used:  $\neg$  binds tightest, next  $\wedge$  and  $\vee$ , then  $\Rightarrow$  and  $\equiv$ , and finally  $\forall$  and  $\exists$ . Desambiguation can be done using brackets ‘(’ and ‘)’. E.g. the formula  $\forall x P(x) \wedge \neg Q(x) \Rightarrow R(x)$  is equivalent to the formula  $\forall x ((P(x) \wedge (\neg Q(x))) \Rightarrow R(x))$ .

As for terms, if term  $t$  occurs in predicate  $P$ , then the type of  $t$  and the type of the input position of  $P$  where it occurs must have a common ancestor type. For formulas of the form  $t_1 = t_2$ ,  $t_1$  and  $t_2$  must have a common ancestor type.

The *scope* of a quantification  $\forall x$  or  $\exists x$ , is the quantified formula. E.g., in  $\forall x \psi$ , the scope of  $\forall x$  is the formula  $\psi$ . An occurrence of a variable  $x$  that is not inside the scope of a quantification  $\forall x$  or  $\exists x$  is called *free*. A *sentence* is a formula containing no free occurrences of variables. If an IDP problem specification contains formulas that are not sentences, the system will implicitly quantify this variable universally and return a warning message, specifying which variables occur free. Each sentence in IDP should end with a dot ‘.’.

The IDP syntax of the different symbols in formulas are given in the table below. Also the informal meaning of the symbols is given.

| Logic         | IDP                      | Declarative reading |
|---------------|--------------------------|---------------------|
| $\wedge$      | $\&$                     | and                 |
| $\vee$        | $ $                      | or                  |
| $\neg$        | $\sim$                   | not                 |
| $\Rightarrow$ | $\Rightarrow$            | implies             |
| $\Leftarrow$  | $\Leftarrow$             | is implied by       |
| $\equiv$      | $\Leftarrow \Rightarrow$ | is equivalent to    |
| $\forall$     | $!$                      | for each            |
| $\exists$     | $?$                      | there exists        |
| $=$           | $=$                      | equals              |
| $\neq$        | $\sim =$                 | does not equal      |

Besides this, for every natural number  $n$ , IDP also supports the following quantifiers (with their respective meanings):

| IDP   | Declarative reading                          |
|-------|--|
| $?n$  | there exist $n$ different elements such that |
| $?<n$ | there exist less than $n$                    |
| $?=n$ | there exist at most $n$                      |
| $?=n$ | there exist exactly $n$                      |
| $?>n$ | there exist more than $n$                    |

A universally quantified formula  $\forall x P(x)$  becomes ‘ $! x : P(x)$ ’ in IDP syntax, and similarly for existentially quantified formulas. As a shorthand for the formula ‘ $! x : ! y : ! z : Q(x, y, z)$ ’, one can write ‘ $! x y z : Q(x, y, z)$ ’.

In IDP, every variable has a type. The informal meaning of a sentence of the form  $\forall x \psi$ , respectively  $\exists x \psi$ , where  $x$  has type  $T$  is then ‘for each object  $x$  of type  $T$ ,  $\psi$  must be true’, respectively ‘there exists at least one object  $x$  of type  $T$  such that  $\psi$  is true’. The type of a variable can be declared by the user, or derived by IDP (see section 6.6).

### 6.1.3 Definitions

A definition defines a concept, i.e. a predicate, in terms of other predicates. Formally, a definition is a set of rules of the form

$$\forall x_1, \dots, x_n \ P(t_1, \dots, t_m) \leftarrow \varphi$$

where  $P$  is a predicate symbol,  $t_1, \dots, t_m$  are terms that may contain the variables  $x_1, \dots, x_n$  and  $\varphi$  a formula that may contain these variables.  $P(t_1, \dots, t_m)$  is called the *head* of the rule and  $\psi$  the *body*.

A definition in IDP syntax consists of a set of rules, enclosed by ‘{’ and ‘}’. Each rule ends with a ‘.’. The definitional implication  $\leftarrow$  is written ‘<-’. The quantifications before the head may be omitted in IDP, i.e., all free variables of a rule are implicitly universally quantified. If the body of a rule is empty, the rule symbol ‘<-’ can be omitted. Recursive definitions are allowed in IDP. The semantics for a definitions are the wellfounded semantics [TODO: reference](#).

## 6.2 Chains of (in)equalities

As in mathematics, one can write chains of (in)equalities in IDP. They can be used as shorthands for conjunctions of (in)equalities. E.g.:

```
! x y : (1 =< x < y =< 5) => ...
// is a shorthand for
! x y : (1 =< x) & (x < y) & (y =< 5) => ...
```

## 6.3 Aggregates

Aggregates are functions that take a set as argument, instead of a simple variable. IDP supports some aggregates that map a set to an integer. As such, they can be seen as integer terms. There are two kinds of sets in IDP.

- An expression of the form ‘[ (phi\_1,t\_1) ; (phi\_2,t\_2) ; ... ; (phi\_n,t\_n) ]’, where each  $\phi_i$  is a formula and each  $t_i$  is a term.
- An expression of the form ‘{ x\_1 x\_2 ... x\_n : phi:t }’, where the  $x_i$  are variables,  $\phi$  is a formula and  $t$  is a term.

The current system has support for five aggregate functions:

**Cardinality:** The cardinality of a set is the number of elements in that set. The IDP syntax for the cardinality of a set  $S$  is ‘card  $S$ ’ or ‘#  $S$ ’. For the first kind of sets, this denotes the number of formulas  $\phi_i$  that are true. For the second kind, this is interpreted as the number of tuples  $(a_1, a_2, \dots, a_n)$  such that  $\phi$  is true.

**Sum:** Let  $S$  be a set of the second form, i.e., of the form ‘{ x\_1 x\_2 ... x\_n : phi: t }’. Then the interpretation of ‘sum  $S$ ’ denotes the number

$$\sum_{(a_1, a_2, \dots, a_n) \models \phi} t,$$

i.e., it is the sum of all the terms for which there exist  $\mathbf{a}_1, \dots, \mathbf{a}_n$  that make the formula  $\mathbf{phi}$  true. For sets of the first sort, this is interpreted as

$$\sum_{i | I \models \mathbf{phi} \rightarrow i} \mathbf{t}_i.$$

**Product:** Products are defined similar to sum.

**Maximum:** One can write ‘ $\max S$ ’ to denote the maximum value of the term in  $S$ , i.e.,

$$\max(\{\mathbf{t} \mid (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n) | I \models \mathbf{phi} \mathbf{t}, \})$$

for sets of the second sort. Sets of the first sort are handled analogously.

**Minimum:** To get the minimum value, write ‘ $\min S$ ’.

When using cardinality, the terms do not matter. You can choose to write 1 for every term, but are also allowed to leave out the terms.

## 6.4 Partial functions

A normal function is total: it assigns an output value to each of its input values. On the other hand, *partial* functions do not necessarily have this property. In IDP, partial function  $F$  can arise in different situations. Either  $F$  is explicitly declared as partial function, or it is declared total, but its input types or output type are subtypes or integer types.

The semantics of a partial function  $F$  is given by transforming constraints and rules where  $F$  occurs as follows:

- in a *positive* context,  $P(\dots, F(x), \dots)$  is transformed to  $\forall y (F(x) = y \Rightarrow P(\dots, y, \dots))$ ;
- in a *negative* context,  $P(\dots, F(x), \dots)$  is transformed to  $\exists y (F(y) = y \wedge P(\dots, y, \dots))$ .

Here,  $P(\dots, F(x), \dots)$  occurs in a positive context if it occurs in sentence and in the scope of an even number of negations, or it occurs in a body of a rule and in the scope of an odd number of negations. All other occurrences are in a negative context.

## 6.5 Fixpoint definitions

TODO: nog niet getest. ik vermoed dat er hier niets over komt

## 6.6 The Type of a Variable

There are two ways to assign a type  $t$  to a variable  $v$ :

- Explicitly mention the type of  $v$  between ‘[’ and ‘]’ when  $v$  is quantified. Then  $v$  gets type  $t$  in the scope of the quantifier. E.g.,

```
theory T:V{
  ! MyVar[MyType] : ? MyVar2[MyType2] MyVar3[MyType3] : // ...
}
```

- Do not mention the type of  $v$  but let the system automatically derive it. The rest of this section explains how this is done.



### 6.6.1 Automatic derivation of types for variables

We distinguish between *typed* and *untyped* occurrences. The following are typed occurrences of a variable  $x$ :

- an occurrence as argument of a non-overloaded predicate:  $P(\dots, x, \dots)$ ;
- an occurrence as argument of a non-overloaded function:  $F(\dots, x, \dots) = \dots$ ;
- an occurrence as return value of a non-overloaded function:  $F(\dots) = x$  or  $F(\dots) \neq x$ .

All others positions are untyped.

An overloaded predicate or function symbol can be disambiguated by specifying its vocabulary and / or types. E.g.,

```
! x : MyVoc :: P[A, A](x, x).  
! y : ?1 x : F[A : A](x) = y.  
MyVoc :: C[: A] > 2.
```

In this case, the occurrences of all variables are typed.

Basically, if a variable occurs in a typed position, it gets the type of that position. If a declared variable with type  $T_1$  occurs in a typed position of type  $T_2$ , then  $T_1$  and  $T_2$  should have a common ancestor type.

The more complicated cases arise when a variable does not occur in any typed position, or it occurs in two typed positions with a different type. The system is designed to give a reasonable type to such variables. However, the choices made by the system are ad hoc and are probably not the ones the user intended. [TODO: stukje over wat typederivation precies doet voor gelijkheid en zo.. Broes?](#)

First consider the case where a variable occurs in typed positions with different types. The IDP system will then give a warning. If all the typed positions where the variable occurs have a common ancestor type  $T$ , then the variable is assigned this type  $T$ . If they do not have a common ancestor, no derivation is done.

Now consider the case where a variable does not occur in a typed position. Then, the IDP system tries to find out what the type of the variable should be using its occurrences in untyped position in built-in overloaded functions. For example, when a variable  $x$  only occurs in  $x = t$ , then  $x$  will get the same type as  $t$ . This behaviour might not always be the desired, so the IDP system will give a warning, including which type it derived for the variable. It's always safer to declare a type for the variable in this case. If it is not possible to derive a type for  $x$  in this way either, the IDP system reports an error.

## 7 Structures

A (three-valued) structure with name `MyStruct` over a vocabulary `MyVoc` is declared by

```
structure MyStruct : MyVoc {  
  // contents of the structures  
}
```

or by

```
asp_structure MyStruct: MyVoc{
    //contents of the structures
}
```

## 7.1 Contents of a structure

A particular input to a problem can be given by giving a (three valued) interpretation to all types and some predicate and function symbols of a given vocabulary. Here, we describe the different ways to specify a structure.

### 7.1.1 Type Enumeration

The syntax for a type enumeration is

```
MyType = { El_1; El_2; ... ; El_n }
```

where **MyType** is the name of the enumerated type and **El\_1; El\_2; ... ; El\_n** are the names of the objects of that type. Names of objects can be (positive and negative) integers, strings, chars, compound domain elements, or identifiers that start with an upper- or lowercase letter. In the case all subtypes of a given type are specified, the supertype is derived to be the union of all elements of the subtypes. If a type is not specified, all domain elements of that type that occur in a predicate or function interpretation (see below) are automatically added to that type.

### 7.1.2 Predicate Enumeration

The syntax for enumerating all tuples for which a predicate **MyPred** with  $n$  arguments is true is as follows.

```
MyPred = { El_1_1, ..., El_1_n;
           ... ;
           El_m_1, ..., El_m_n
        }
```

It is also possible to write parentheses around tuples.

```
MyPred = { (El_1_1, ..., El_1_n);
           ... ;
           (El_m_1, ..., El_m_n)
        }
```

This notation makes it possible to state that a proposition (a predicate with no arguments) is true, by using an empty tuple.

```
true = { () }
false = { }
```

However, it might be easier to use **true** and **false** instead of **{ () }** and **{ }**.

### 7.1.3 Function Enumeration

The syntax for enumerating a function `MyFunc` with  $n$  arguments is

```
MyFunc = { El_1_1, ..., El_1_n -> El_1;
          ...;
          El_m_1, ..., El_m_n -> El_m
        }
```

To give the interpretation of a constant, one can simply write ‘`MyConst = El`’ instead of ‘`MyConst = { -> El }`’.

### 7.1.4 Compound Domain Elements

A function applied to a tuple of domain elements can be used as a domain element. We call such a domain element a *compound domain element*. An example is the domain element  $F(1, a)$ . If  $F/n$  is a function then

```
F = generate
```

specifies that the interpretation of  $F$  is the two-valued interpretation that maps each tuple  $(d_1, \dots, d_n)$  to the compound domain element  $F(d_1, \dots, d_n)$ .

### 7.1.5 Three-Valued Predicate/Function interpretations

Three-valued interpretations are given by either

- enumerating the certainly true and certainly false tuples;
- enumerating the certainly true and the unknown tuples;
- enumerating the unknown and the certainly false tuples.

To specify which tuples are enumerated, use `<ct>`, `<cf>` and `<u>`. For example

```
P<ct> = { /* enumeration of the certainly true tuples of P */ }
P<u> = { /* enumeration of the unknown tuples of P */ }
```

### 7.1.6 Interpretation by Procedures

The syntax

```
P = procedure MyProc
```

is used to interpret a predicate or function symbol  $P$  by a procedure `MyProc` (see below). If  $P$  is an  $n$ -ary predicate, then `MyProc` should be an  $n$ -ary procedure that returns a boolean. If  $P$  is an  $n$ -ary function, then `MyProc` should be an  $n$ -ary function that returns a number, string, or compound domain element.

### 7.1.7 Shorthands

Shorthands like ‘`MyType = {1..10; 15..20}`’ or ‘`MyType = { a..e; A..E }`’ may be used for enumerating types or predicates with only one argument.

## 7.2 ASP structures

An ASP structure consists of a list of facts in the usual ASP syntax. In particular, everything from a % till the end of the line is considered a comment, and - before an atom denotes classical negation (negation as failure is not available). A fact about functions is written like  $F(a) = b$  or  $\neg F(c) = d$ .

## 8 Procedures

### 8.1 Declaring a procedure

A procedure with name `MyProc` and arguments `A1, ..., An` is declared by

```
procedure MyProc(A1, ..., An) {  
    // contents of the procedure  
}
```

Inside a procedure, any chunk of Lua code can be written. For Lua's reference manual, see <http://www.lua.org/manual/5.1/>. In the following, we assume that the reader is familiar with the basic concepts of Lua.

### 8.2 IDP types

Besides the standard types of variables available in Lua, the following extra types are available in IDP procedures.

**sort** A set of sorts with the same name. Can be used as a single sort if the set is a singleton.

**predicate\_symbol** A set of predicates with the same name, but possibly with different arities. Can be used as a single predicate if the set is a singleton. If `P` is a `predicate_symbol` and `n` an integer, then `P/n` returns a `predicate_symbol` containing all predicates in `P` with arity `n`. If `s1, ..., sn` are sorts, then `P[s1, ..., sn]` returns a `predicate_symbol` containing all predicates `Q/n` in `P`, such that the  $i$ 'th sort of `Q` belongs to the set `si`, for  $1 \leq i \leq n$ .

**function\_symbol** A set of first-order functions with the same name, but possibly with different arities. Can be used as a single first-order function if the set is a singleton. If `F` is a `function_symbol` and `n` an integer, then `F/n:1` returns a `function_symbol` containing all function in `F` with arity `n`. If `s1, ..., sn, t` are sorts, then `F[s1, ..., sn:t]` returns a `function_symbol` containing all functions `G/n` in `F`, such that the  $i$ 'th sort of `F` belongs to the set `si`, for  $1 \leq i \leq n$ , and the output sort of `G` belongs to `t`.

**symbol** A set of symbols of a vocabulary with the same name. Can be used as if it were a sort, `predicate_symbol`, or `function_symbol`.

**vocabulary** A vocabulary. If `V` is a vocabulary and `s` a string, `V[s]` returns the symbols in `V` with name `s`.

**compound** A domainelement of the form  $F(d_1, \dots, d_n)$ , where  $F$  is a first-order function and  $d_1, \dots, d_n$  are domain elements.

**tuple** A tuple of domain elements. `T[n]` returns the  $n$ 'th element in tuple `T`.

**predicate\_table** A table of tuples of domain elements.

**predicate\_interpretation** An interpretation for a predicate. If **T** is a **predicate\_interpretation**, then **T.ct**, **T.pt**, **T.cf**, **T.pf** return a **predicate\_table** containing, respectively, the certainly true, possibly true, certainly false, and possibly false tuples in **T**.

**function\_interpretation** An interpretation for a function. **F.graph** returns the **predicate\_interpretation** of the graph associated to the **function\_interpretation** **F**.

**structure** A first-order structure. To obtain the interpretation of a sort, singleton **predicate\_symbol**, or singleton **function\_symbol** **symb** in structure **S**, write **S[symb]**.

**theory** A logic theory.

**options** A set of options.

**namespace** A namespace.

**overloaded** An overloaded object.

### 8.3 Built-in procedures

A lot of procedures are already built in. Typing **help** in interactive mode shows an overview of the available procedures, together with a description.

## 9 Options

The IDP system has various options. To set an option, you can use the following lua-code

```
stdoptions.MyOption = MyValue
```

where **MyOption** is the name of the option and **MyValue** is the value you want to give it. If you want to have multiple option sets, you can make them with them with

```
FirstOptionSet = newOptions()  
SecondOptionSet = newOptions()  
FirstOptionSet.MyOption = MyValue  
SecondOptionSet.MyOption = MyValue
```

To activate an option set, use the procedure **setascurrentoptions(MyOptionSet)**. From that moment, **MyOptionSet** will be used in all commands.

**autocomplete** = [**false**, **true**] Turn autocompletion on or off

**groundverbosity** = [**0..2147483647**] Verbosity of the grounder. The higher the verbosity, the more debug information is printed.

**language** = [**ecnf**, **idp**, **tptp**] The language used when printing objects.

**longnames** = [**false**, **true**] If true, everything is printed with reference to their vocabulary. For example, a predicate **P** from vocabulary **V** will be printed as **V::P** instead of **P**.

**nbmodelequivalent** = [false, true ] If set true, the result of modelexpansion will be **all** models.  
If set false, some models might be left out for efficiency.

**nrmodels** = [0..2147483647 ] Set the number of models wanted from the modelexpansion inference. If set to 0, all models are returned.

**satverbosity** = [0..2147483647 ] Like groundverbosity, but controls the verbosity of MINISAT(ID)

**timeout** = [0..2147483647 ] Set the timeout for inferences (in seconds)

**trace** = [false, true ] If true, the procedure modelexpand produces also an execution trace of MINISAT(ID)