5.4 Dicts: structures with named arguments

SWI-Prolog version 7 (2014) introduced the *dict*, an associative data structure (otherwise known as a *map*: a mapping of keys to values), as an abstract object with a concrete modern syntax.

It adds a *dot-notation* `dict.key` for accessing dict entries and `dict.func(x)` for calling “functions associated to a dict'', which resolve to user-defined predicates. The dot-notation is similar to notations found in other programming languages.

The syntax for defining a dict (the *dict literal*) is as follows:

> Tag{Key1:Value1, Key2:Value2, ...}

Where:

- The dict's *tag* is either an unbound variable (then we have an *anonymous dict*) or an atomic value. In fact, arbitrary terms *are* currently allowed as tags, but this is a quirk of the implementation and one should not rely on this behaviour. Similar to compound terms, there is *no* space between the dict's *tag* and the subsequent opening brace. A tag that is an atom *associates the dict to a module*, i.e. if SWI-Prolog encounters a function call on the dict, it will look for a corresponding predicate in the module that has the name given by the dict tag.
- The dict's *keys* are either atoms or “small''[161] integers. A dict may *not* hold duplicate keys.
- The dict's *values* are arbitrary Prolog terms which are parsed using the same rules as used for arguments in compound terms.

The dict is transformed into an opaque internal representation that does *not* respect the order in which the key-value pairs appear in the dict literal. Any apparent order of keys is irrelevant: the keys have set semantics. If a dict is written, the keys are written according to the standard order of terms (see section 4.6.1).

Here are some examples:

An anonymous dict with two entries:

```
?- X0 = _{x:1, y:2}.
X0 = _28190{x:1, y:2}.
```

A dict with a string tag and two entries:

```
?- X1 = A{x:1, y:2}, A="foo".
X1 = "foo"{x:1,y:2},
A = "foo".
```

A dict representing a point in some 2-dimensional space. The variously written dict literals yield dicts that unify and are “the same'' according to ==/2.

```
?- point{x:1, y:2} = point{y:2, x:1}.
true.
```

```
?- point{x:1, y:2} == point{y:2, x:1}.
true.
```

Dicts are unified following the standard symmetric Prolog unification rules. For unification to succeed:

- the tags must unify;
- each key must be present in both dicts; and
- the corresponding values must unify.

Some examples to illustrate unification behaviour:

Case of tags that unify unify and values that unify for each key:

```
?- point{x:1, y:2} = Tag{y:2, x:X}.
Tag = point,
X = 1.

?- point{x:1} = Tag{x:1}.
Tag = point.

?- point{x:1,y:2} = Tag{x:X,y:Y}.
Tag = point,
X = 1,
Y = 2.
```

Case of tags that don't unify:

```
?- point{x:1} = vector{x:1}.
false.
```

Case of values that don't unify:

```
?- point{x:1} = Tag{x:2}.
false.
```

Case of key sets that differ:

```
?- point{x:1,y:2} = Tag{x:X}.
false.
```

Note In future versions, the notion of unification between dicts could be modified such that two dicts unify if their tags and the values associated with *common* keys unify, turning both dicts into a new dict that holds the union of the keys of the two original dicts.

5.4.1 Growing and shrinking dicts

Unless you decide to use the destructive assignment predicates of section 5.4.3.1, you cannot change a dict's set of keys. -- a dict does not behave like an *open list* to which you can add elements. Of course, you can still refine/bind any variable appearing in the dict's tag and values to arbitrary terms.

To add or remove a key you have to generate a new dict from an existing one, in functional style. Predicates that “modify dicts'' take an input dict at a `+DictIn` position, generate a new dict and unify that dict at a `-DictOut` position (using the common mode indicator notation of section 4.1.1).

To grow a dict:[162]

you can use the corresponding predefined dict *function* put/1 - `.put(+Map)`:

```
?- Pt = point{x:1, y:2}, NewPt = Pt.put(_{z:3}).
Pt = point{x:1, y:2},
NewPt = point{x:1, y:2, z:3}.

?- Pt = point{x:1, y:2}, NewPt = Pt.put([z-3,i-4,j-5,k-6]).
Pt = point{x:1, y:2},
NewPt = point{i:4, j:5, k:6, x:1, y:2, z:3}.
```

you can call the *predicate* put_dict/3, - `put_dict(+New, +DictIn, -DictOut)`:

```
?- Pt = point{x:1, y:2}, put_dict([z-3,i-4,j-5,k-6],Pt,NewPt).
Pt = point{x:1, y:2},
NewPt = point{i:4, j:5, k:6, x:1, y:2, z:3}.
```

or you can call the *predicate* put_dict/4 - `put_dict(+Key, +DictIn, +Value, -DictOut)` - to modify a single entry only:

```
?- Pt = point{x:1, y:2}, put_dict(z,Pt,3,NewPt).
Pt = point{x:1, y:2},
NewPt = point{x:1, y:2, z:3}.
```

On the other hand, *shrinking* a dict can only be done with the *predicate* del_dict/4, - `del_dict(+Key, +DictIn, ?Value, -DictOut)` - ; there is no corresponding function:

```
?- Pt = point{x:1, y:2}, del_dict(x,Pt,OldVal,NewPt).
Pt = point{x:1, y:2},
OldVal = 1,
NewPt = point{y:2}.
```

5.4.2 Functions on dicts

The infix operator *dot*, `op(100, yfx, .)`, is used both to select dict entries (access their value), and to call functions associated to dicts.

5.4.2.1 General approach

If the `./2` appears in a goal, (possibly nested inside some term) the compiler translates it into a call to predicate ./3, `.(+Dict, +Function, -Result)`, defined in the `system` module.

.(+*Dict*, +*Function*, -*Result*)

This predicate is called to evaluate `./2` terms found in the arguments of a goal.

If argument *Function* denotes an atom `k` then the semantics are those of a selector: the value associated to key `k` of *Dict* is unified with *Result*, which is then used at the position where the `./2` appeared.

The entry must exist. If it is missing, an exception with error term `existence_error(key, KeyName, Dict)` is raised.[163]

See section 5.4.2.2 for more.

If argument *Function* denotes a compound term then the predicate corresponding to that compound term, extended with rightmost arguments *Dict* and *Result*, is called.

If the predicate call succeeds, *Result* is used at the position where the `./2` appeared, in the sense that *Result* is unified with a fresh variable at that position. If the predicate call fails, the code ultimately behaves as if that unification failed (but see below).

The predicate must exist. If it is missing, an ISO standard exception with error term `existence_error(procedure, DictTag:PredicateIndicator)` is raised.

See section 5.4.2.3 and section 5.4.2.4 for more on this.

Note: Whether both sides of an “unification of terms obtained from function calls'' will actually be called depends.

Consider this function on 2-dimensional points:

```
len2d(Pt,L) :-
   L is sqrt(Pt.x**2 + Pt.y**2),
   (L<2 -> R=true ; R=false),
   format("Now testing ~q<2 on ~q: ~q\n",[L,Pt,R]),
   L<2.
```

Then:

```
?- L = point{x:5,y:1}.len2d().
Now testing 5.0990195135927845<2 on point{x:5,y:1}: false
false.

?- L = point{x:1,y:1}.len2d().
Now testing 1.4142135623730951<2 on point{x:1,y:1}: true
L = 1.4142135623730951.

?- point{x:1,y:1}.len2d() = point{x:5,y:1}.len2d().
Now testing 1.4142135623730951<2 on point{x:1,y:1}: true
Now testing 5.0990195135927845<2 on point{x:5,y:1}: false
false.

?- point{x:5,y:1}.len2d() = point{x:1,y:1}.len2d().
Now testing 5.0990195135927845<2 on point{x:5,y:1}: false
false.
```

5.4.2.2 Value selection using ./2

Examples:

Select the value of key `x` directly on the dict literal:

```
?- X = point{x:1,y:2}.x.
X = 1.
```

Select the value of key `y` on a variable bound to the dict:

```
?- Pt = point{x:1,y:2}, write(Pt.y).
2
Pt = point{x:1,y:2}.
```

The entry must exist:

```
?-  Z = point{x:1,y:2}.z.
ERROR: key `z' does not exist in point{x:1,y:2}
```

The selector can be a variable itself, here given by `Coord`. It backtracks over the key set. Backtracking over the key set allows one to grab values that unify with a template term in a compact way:

```
?- bagof([Coord-Val],(point{x:1,y:2}.Coord = Val),Bag).
Bag = [[x-1], [y-2]].

?- p(Arg,1) = pairs{a:p(foo,2),b:p(bar,1),c:p(baz,1)}.Key.
Arg = bar,
Key = b ;
Arg = baz,
Key = c.
```

You can inspect the compiler's translation of `./2` to standard predicate calls using listing/1:

```
f :-
   Pt = pt{x:1,y:2},
   format("x = ~q, y = ~q\n",[Pt.x,Pt.y]).

?- listing(f/0).
f :-
    A=pt{x:1, y:2},
    '.'(A, x, B),
    '.'(A, y, C),
    format("x = ~q, y = ~q\n", [B, C]).
```

5.4.2.3 User-defined functions on dicts

Function calls are mapped to normal Prolog predicate calls which can be coded as usual. However, the dict infrastructure provides a convenient syntax for representing the head of such predicates without worrying about the argument calling conventions.

The tag of a dict *associates the dict to a module*, i.e. if the tag is an atom and there is a module which has the same atom as name, then SWI-Prolog looks for a predicate to service function calls in the module named after the tag.

The following goal will then be called:

> Tag:FunctionName(Arg1, ..., +Dict, -Value)

As this approach suggests, defining functions with multiple clauses is allowed. One can thus provide overloading and non-determinism as usual.

Example:

len2d/2 is a predicate which takes a dict representing a point in a 2-dimensional space and computes its euclidean distance from the origin. Here, the clause head does not demand specific structure from a dict argument (maybe it should):

```
len2d(Pt,L) :- L is sqrt(Pt.x**2 + Pt.y**2).
```

Call as a predicate:

```
?- len2d(point{x:3,y:4},L).
L = 5.0.
```

Call as 0-ary function associated to a dict:

```
?- L = point{x:3,y:4}.len2d().
L = 5.0.
```

Pack the 0-ary function call into a predicate and examine the predicate's code:

```
f_len2d(P,L) :- (L = P.len2d()).

?- listing(f_len2d/2).
f_len2d(A, B) :-
    '.'(A, len2d(), C),
    B=C.
```

More conveniently, the `:=` operator allows one to abstract the location of the predicate arguments.[164]

The same len2d/2 as above, but now `Pt` and `L` are declared differently. The arity is 0:

```
Pt.len2d() := L :-
   L is sqrt(Pt.x**2 + Pt.y**2).
```

Add another function to scale the point by a factor `F`, returning a new dict:

```
Pt.scale2d(F) := point{x:X, y:Y} :-
   X is Pt.x*F,
   Y is Pt.y*F.
```

After these definitions, we can call the following functions:

```
?- X = point{x:1, y:2}.scale2d(2).
X = point{x:2, y:4}.
```

Chaining is allowed:

```
?- X = point{x:1, y:2}.scale2d(2).len2d().
X = 4.47213595499958.
```

5.4.2.4 Predefined functions on dicts

Dicts currently define the following reserved functions:

get(*?Key*)

Same as *Dict.Key*, but fails silently (instead of throwing) if the dict does not contain *Key*. See also :</2, which can be used to test for existence and unify multiple key values from a dict. For example:

```
?- write(point{x:3,y:4}.get(x)).
3
true.
```

Select a nonexistent key. Note that it is not the write/2 which fails but the selection, which is actually done prior to calling write/2:

```
?- write(point{x:3,y:4}.get(z)).
false.
```

put(+*NewPairs*)

Evaluates to a new dict where the key-value pairs from *NewPairs* replace or extend the key-value pairs in the original dict. *NewPairs* is either dict or a valid input for dict_create/3. See the put_dict/3 predicate.

put(+*KeyPath, +Value*)

Evaluates to a new dict where the *KeyPath-Value* replaces or extends the key-value pairs in the original dict. *KeyPath* is either a key or a term *KeyPath*/*Key*[165], replacing the value associated with *Key* in a sub-dict of the dict on which the function operates. See the put_dict/4 predicate.

Grow by a new entry:

```
?- D1 = point{}.put(x, 1).
D1 = point{x:1}.
```

Modify an existing entry:

```
?- D2 = point{x:1}.put(x, 400).
D2 = point{x:400}.
```

Grow by a new entry which has a (sub)dict as value:

```
?- D3 = point{x:1}.put(y/realpart, 2).
D3 = point{x:1, y:_20294{realpart:2}}.
```

Modify subdict entry of an existing entry:

```
?- D4 = point{x:_{realpart:2}}.put(x/realpart, 400).
D4 = point{x:_22716{realpart:400}}.
```

5.4.2.5 Dict functions calls in a clause head

Parameters of calls to predicate with clauses that have `./2` terms or subterms in their head are unified not with the term found in the clause head as usual (which is a purely syntactic operation) but with the result of an *arbitrary function evaluation* (which may even perform I/O operations).

This is made possible by replacing the `./2` call in the clause head with a fresh variable, which is unified with the result of the call to ./3 made at the beginning of the (modified) clause body.

An example with a *selector* directly in the clause head:

```
extract_x(D,D.x).

?- extract_x(point{x:1,y:1},R).
R = 1.

?- listing(extract_x/2).
extract_x(A, B) :-
    '.'(A, x, B).
```

The 2-dimensional euclidean length example from above, suitably modified:

```
compact_len2d(P,P.len2d()).

?- compact_len2d(point{x:3,y:4},L).
L = 5.0.

?- listing(compact_len2d/2).
compact_len2d(A, B) :-
    '.'(A, len2d(), B).
```

An example whereby the first argument *Dict* is extended with a new entry (the postal code), assuming a find_postal_code/4 predicate. The extended dict is constructed directly in the head:

```
add_postal_code(Dict, Dict.put(postal_code, Code)) :-
        find_postal_code(Dict.city,
                         Dict.street,
                         Dict.house_number,
                         Code).
```

Suppose the following fact has been asserted:

```
find_postal_code("Jamestown","Ash Tree Lane","44","23081").
```

Then:

```
?- add_postal_code(
      house{city:"Jamestown",street:"Ash Tree Lane",house_number:"44"},
      DictOut).
DictOut = house{city:"Jamestown", house_number:"44", postal_code:"23081", street:"Ash Tree Lane"}.
```

Alternatively, you can also unify the result of the inside-the-head .put/2 call with an existing dict:

```
?- add_postal_code(
      house{city:"Jamestown",street:"Ash Tree Lane",house_number:"44"},
      house{city:C,street:S,house_number:N,postal_code:X}).
C = "Jamestown",
S = "Ash Tree Lane",
N = "44",
X = "23081".
```

Let's take a look at the generated code:

```
?- listing(add_postal_code/2).
add_postal_code(A, B) :-
    '.'(A, put(postal_code, C), B),
    '.'(A, city, D),
    '.'(A, street, E),
    '.'(A, house_number, F),
    find_postal_code(D, E, F, C).
```

5.4.2.6 Expression containing ./2 are evaluated eagerly as dict function calls

Evaluation of `./2` is eager. You can think of it as follows: if `./2` appears in a term that is about to be unified, SWI-Prolog tries to perform the corresponding call at once. This implies that such terms cannot be created by writing them explicitly in your source code.

Such terms can still be created with functor/3, =../2, compound_name_arity/3 and compound_name_arguments/3. [166]

For example:

The term inside the parentheses of write_canonical/1 is not written “as is''; the *dot* operator triggers a function call first:

```
?- write_canonical(_{x:12,y:13}.y :- a).
:-(13,a)
true.
```

If there is an unbound variable at dict position, an exception is raised:

```
?- write_canonical(X.y :- a).
ERROR: Arguments are not sufficiently instantiated
```

This does not happen for other operators:

```
?- write_canonical(X/y :- a).
:-(/(_,y),a)
true.
```

You have to work around term expansion to obtain terms that use the dot operator:

```
?- compound_name_arguments(C,'.',[X,y]),write_canonical(C :- a).
:-('.'(_,y),a)
C = X.y.
```

5.4.3 Predicates for managing dicts

This section documents the predicates that are defined on dicts. We use the naming and argument conventions of the traditional `library(assoc)`.

is_dict(*@Term*)

True if *Term* is a dict. This is the same as `is_dict(Term,_)`.

is_dict(*@Term, -Tag*)

True if *Term* is a dict of *Tag*.

get_dict(*?Key, +Dict, -Value*)

Unify the value associated with *Key* in dict with *Value*. If *Key* is unbound, all associations in *Dict* are returned on backtracking. The order in which the associations are returned is undefined. This predicate is normally accessed using the functional notation `Dict.Key`. See section 5.4.2.

Fails silently if Key does not appear in Dict. This is different from the behavior of the functional‘.`-notation, which throws an existence error in that case.

get_dict(*+Key, +Dict, -Value, -NewDict, +NewValue*) *[semidet]*

Create a new dict after updating the value for *Key*. Fails if *Value* does not unify with the current value associated with *Key*. *Dict* is either a dict or a list the can be converted into a dict.

Has the behavior as if defined in the following way:

```
get_dict(Key, Dict, Value, NewDict, NewValue) :-
        get_dict(Key, Dict, Value),
        put_dict(Key, Dict, NewValue, NewDict).
```

dict_create(*-Dict, +Tag, +Data*)

Create a dict in *Tag* from *Data*. *Data* is a list of attribute-value pairs using the syntax `Key:Value`, `Key=Value`, `Key-Value` or `Key(Value)`. An exception is raised if *Data* is not a proper list, one of the elements is not of the shape above, a key is neither an atom nor a small integer or there is a duplicate key.

dict_pairs(*?Dict, ?Tag, ?Pairs*)

Bi-directional mapping between a dict and an ordered list of pairs (see section A.29).

put_dict(*+New, +DictIn, -DictOut*)

DictOut is a new dict created by replacing or adding key-value pairs from *New* to *Dict*. *New* is either a dict or a valid input for dict_create/3. This predicate is normally accessed using the functional notation. Below are some examples:

```
?- A = point{x:1, y:2}.put(_{x:3}).
A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3]).
A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3,z=0]).
A = point{x:3, y:2, z:0}.
```

put_dict(*+Key, +DictIn, +Value, -DictOut*)

DictOut is a new dict created by replacing or adding *Key-Value* to *DictIn*. For example:

```
?- A = point{x:1, y:2}.put(x, 3).
A = point{x:3, y:2}.
```

This predicate can also be accessed by using the functional notation, in which case Key can also be a *path* of keys. For example:

```
?- Dict = _{}.put(a/b, c).
Dict = _6096{a:_6200{b:c}}.
```

del_dict(*+Key, +DictIn, ?Value, -DictOut*)

True when *Key-Value* is in *DictIn* and *DictOut* contains all associations of *DictIn* except for *Key*.

+Select **:<** *+From* *[semidet]*

True when *Select* is a‘sub dict' of *From*: the tags must unify and all keys in *Select* must appear with unifying values in *From*. *From* may contain keys that are not in *Select*. This operation is frequently used to *match* a dict and at the same time extract relevant values from it. For example:

```
plot(Dict, On) :-
```

```
        _{x:X, y:Y, z:Z} :< Dict, !,
        plot_xyz(X, Y, Z, On).
plot(Dict, On) :-
        _{x:X, y:Y} :< Dict, !,
        plot_xy(X, Y, On).
```

The goal `Select :< From` is equivalent to `select_dict(Select, From, _)`.

select_dict(+*Select*, +*From*, -*Rest*) *[semidet]*

True when the tags of *Select* and *From* have been unified, all keys in *Select* appear in *From* and the corresponding values have been unified. The key-value pairs of *From* that do not appear in *Select* are used to form an anonymous dict, which us unified with *Rest*. For example:

```
?- select_dict(P{x:0, y:Y}, point{x:0, y:1, z:2}, R).
P = point,
Y = 1,
R = _G1705{z:2}.
```

See also :</2 to ignore *Rest* and >:</2 for a symmetric partial unification of two dicts.

+*Dict1* **>:<** +*Dict2*

This operator specifies a *partial unification* between *Dict1* and *Dict2*. It is true when the tags and the values associated with all *common* keys have been unified. The values associated to keys that do not appear in the other dict are ignored. Partial unification is symmetric. For example, given a list of dicts, find dicts that represent a point with X equal to zero:

```
    member(Dict, List),
    Dict >:< point{x:0, y:Y}.
```

See also :</2 and select_dict/3.

5.4.3.1 Destructive assignment in dicts

This section describes the destructive update operations defined on dicts. These actions can only *update* keys and not add or remove keys. If the requested key does not exist the predicate raises `existence_error(key, Key, Dict)`. Note the additional argument.

Destructive assignment is a non-logical operation and should be used with care because the system may copy or share identical Prolog terms at any time. Some of this behaviour can be avoided by adding an additional unbound value to the dict. This prevents unwanted sharing and ensures that copy_term/2 actually copies the dict. This pitfall is demonstrated in the example below:

```
?- A = a{a:1}, copy_term(A,B), b_set_dict(a, A, 2).
A = B, B = a{a:2}.

?- A = a{a:1,dummy:_}, copy_term(A,B), b_set_dict(a, A, 2).
A = a{a:2, dummy:_G3195},
B = a{a:1, dummy:_G3391}.
```

b_set_dict(+*Key*, !*Dict*, +*Value*) *[det]*

Destructively update the value associated with *Key* in *Dict* to *Value*. The update is trailed and undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to setarg/3 and b_setval/2.

nb_set_dict(+*Key*, !*Dict*, +*Value*) *[det]*

Destructively update the value associated with *Key* in *Dict* to a copy of *Value*. The update is *not* undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to nb_setarg/3 and nb_setval/2.

nb_link_dict(+*Key*, !*Dict*, +*Value*) *[det]*

Destructively update the value associated with *Key* in *Dict* to *Value*. The update is *not* undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to nb_linkarg/3 and nb_linkval/2. Use with extreme care and consult the documentation of nb_linkval/2 before use.

5.4.4 When to use dicts?

Dicts are a new type in the Prolog world. They compete with several other types and libraries. In the list below we have a closer look at these relations. We will see that dicts are first of all a good replacement for compound terms with a high or not clearly fixed arity, library `library(record)` and option processing.

Compound terms

Compound terms with positional arguments form the traditional way to package data in Prolog. This representation is well understood, fast and compound terms are stored efficiently. Compound terms are still the representation of choice, provided that the number of arguments is low and fixed or compactness or performance are of utmost importance.

A good example of a compound term is the representation of RDF triples using the term `rdf(Subject, Predicate, Object)` because RDF triples are defined to have precisely these three arguments and they are always referred to in this order. An application processing information about persons should probably use dicts because the information that is related to a person is not so fixed. Typically we see first and last name. But there may also be title, middle name, gender, date of birth, etc. The number of arguments becomes unmanageable when using a compound term, while adding or removing an argument leads to many changes in the program.

Library `library(record)`

Using library `library(record)` relieves the maintenance issues associated with using compound terms significantly. The library generates access and modification predicates for each field in a compound term from a declaration. The library provides sound access to compound terms with many arguments. One of its problems is the verbose syntax needed to access or modify fields which results from long names for the generated predicates and the restriction that each field needs to be extracted with a separate goal. Consider the example below, where the first uses library `library(record)` and the second uses dicts.

```
    ...,
    person_first_name(P, FirstName),
    person_last_name(P, LastName),
    format('Dear ~w ~w,~n~n', [FirstName, LastName]).

    ...,
    format('Dear ~w ~w,~n~n', [Dict.first_name, Dict.last_name]).
```

Records have a fixed number of arguments and (non-)existence of an argument must be represented using a value that is outside the normal domain. This lead to unnatural code. For example, suppose our person also has a title. If we know the first name we use this and else we use the title. The code samples below illustrate this.

```
salutation(P) :-
    person_first_name(P, FirstName), nonvar(FirstName), !,
    person_last_name(P, LastName),
    format('Dear ~w ~w,~n~n', [FirstName, LastName]).
salutation(P) :-
    person_title(P, Title), nonvar(Title), !,
    person_last_name(P, LastName),
    format('Dear ~w ~w,~n~n', [Title, LastName]).

salutation(P) :-
    _{first_name:FirstName, last_name:LastName} :< P, !,
    format('Dear ~w ~w,~n~n', [FirstName, LastName]).
salutation(P) :-
    _{title:Title, last_name:LastName} :< P, !,
    format('Dear ~w ~w,~n~n', [Title, LastName]).
```

Library `library(assoc)`
This library implements a balanced binary tree. Dicts can replace the use of this library if the association is fairly static (i.e., there are few update operations), all keys are atoms or (small) integers and the code does not rely on ordered operations.

Library `library(option)`
Option lists are introduced by ISO Prolog, for example for read_term/3, open/4, etc. The `library(option)` library provides operations to extract options, merge options lists, etc. Dicts are well suited to replace option lists because they are cheaper, can be processed faster and have a more natural syntax.

Library `library(pairs)`
This library is commonly used to process large name-value associations. In many cases this concerns short-lived data structures that result from findall/3, maplist/3 and similar list processing predicates. Dicts may play a role if frequent random key lookups are needed on the resulting association. For example, the skeleton'create a pairs list','use list_to_assoc/2 to create an assoc', followed by frequent usage of get_assoc/3 to extract key values can be replaced using dict_pairs/3 and the dict access functions. Using dicts in this scenario is more efficient and provides a more pleasant access syntax.

5.4.5 A motivation for dicts as primary citizens

Dicts, or key-value associations, are a common data structure. A good old example are *property lists* as found in Lisp, while a good recent example is formed by JavaScript *objects*. Traditional Prolog does not offer native property lists. As a result, people are using a wide range of data structures for key-value associations:

- Using compound terms and positional arguments, e.g., `point(1,2)`.
- Using compound terms with library `library(record)`, which generates access predicates for a term using positional arguments from a description.
- Using lists of terms `Name=Value`, `Name-Value`, `Name:Value` or `Name(Value)`.
- Using library `library(assoc)` which represents the associations as a balanced binary tree.

This situation is unfortunate. Each of these have their advantages and disadvantages. E.g., compound terms are compact and fast, but inflexible and using positional arguments quickly breaks down. Library `library(record)` fixes this, but the syntax is considered hard to use. Lists are flexible, but expensive and the alternative key-value representations that are used complicate the matter even more. Library `library(assoc)` allows for efficient manipulation of changing associations, but the syntactical representation of an assoc is complex, which makes them unsuitable for e.g., *options lists* as seen in predicates such as open/4.

5.4.6 Implementation notes about dicts

Although dicts are designed as an abstract data type and we deliberately reserve the possibility to change the representation and even use multiple representations, this section describes the current implementation.

Dicts are currently represented as a compound term using the functor `` `dict` ``. The first argument is the tag. The remaining arguments create an array of sorted key-value pairs. This representation is compact and guarantees good locality. Lookup is order *log(N)*, while adding values, deleting values and merging with other dicts has order *N*. The main disadvantage is that changing values in large dicts is costly, both in terms of memory and time.

Future versions may share keys in a separate structure or use a binary trees to allow for cheaper updates. One of the issues is that the representation must either be kept canonical or unification must be extended to compensate for alternate representations.