9a. Modules in SWI Prolog

– **July 24, 2017** –

Creating and Populating Modules

Special Modules and Default Import

~~Predicate Lookup (Static and Dynamic Binding)~~

Using Modules like Objects

Modules

- Module = name space for predicate definitions

- Each file has an associated module
  - either the explicitly declared module

```
:- module('JTmeta',
       [ ast_node_template/4  % (?Lang, ?NodeType, ?Arity, ?Template)
       , ast_arg_nr_name/4    % (?Lang, ?NodeType, ?ArgNumber, ?ArgName)
       , ast_parent/3         % (?Lang, ?Id, ?ParentId)
       , ...
       ]
).

...
```

Module name ← (pointing to 'JTmeta')

Export list (bracket around the list)

Document the arguments and modes of exported predicates!

  - or otherwise the implicit default module 'user'

Why Modules?

- Different Scopes / Namespaces
  - No name collisions
  - User-Defined Predicates
  - Library Predicates
- Structuring Large Applications
  - Separation of concerns
- Object-oriented Programming???

universitätbonn

Creating and Populating Modules

module/2 directive

use_module/1

use_module/2

import/1

universitätbonn

Exporting and Importing

- A module defines predicates that it exports

- Syntax
  - `:- module( name, [exported/1, …]).`
  - This directive must precede any other directive or declaration in a file.
  - The module name does not need to be the same as the containing filename (but it is strongly recommended to keep them consistent!).

- Semantics
  - Exported predicates will be visible in the module that loads name via `consult(filename)` or `use_module(filename)`.
  - It is an error if two modules export the same predicates (name/arity) into the same importing module.
  - All non-exported predicates are not visible outside name
    - ⇨ … unless some other module uses import/2 to bypass the protection

use_module/3

- **use_module***(+Module, -File, +Imports)* is **det**
  - import from a named module whose file-location is unknown or irrelevant
- **use_module***(-Module, +File, +Imports)* is **det**
  - get access to the name of the module loaded from a file

If both, *Module* and *File,* are given we use *Module*, ignoring *File*.

Recommendation: Avoid use_module/2. Call use_module(+,-,+) instead!

“use_module” versus “consult”

- Think of “use_module/1” as defined by

```
use_module(File) :-
   contains_module(File)
   -> consult(File)
    ; report_missing_module(File)
    .
```

- Recommended way of loading module files
  - Better because it checks that there is actually a module and also makes the intention explicit → easier to understand code

```
:- module( pureImporter, [ myownstuff/4 ] ).

:- use_module('JTmeta').
```

- This also works, but lacks the above advantages:

```
:- module( pureImporter, [ myownstuff/4 ] ).

:- consult('JTmeta').
```


universitätbonn

Implicit import

- Let M be a module declared in the file `filename`.
  - Exported predicates of M will be visible in any Importer module in whose context `consult(filename)` or `use_module(filename)` i is executed
  - This is called "implicit import".

- Importing modules can use the imported predicates internally …

```
:- module( pureImporter, [ myownstuff/4 ] ).

:- use_module('JTmeta').

... internal use of predicates from JTmeta ...
```

- … but can also re-export (some of) them

```
:- module( 're-exporter',    [ myownstuff/4
                             , ast_node_template/4
                             ] ).
:- use_module('JTmeta').

... internal use of predicates from JTmeta ...
```

Reexported predicate from JTmeta (refers to `ast_node_template/4`)

Re-Export Shortcut

- Re-Exporting all predicates of an imported module
  - `reexport(+Files)`
    - Import all predicates from the export list of each file in Files like use_module/1
    - ... but additionally re-export all imported predicates
    - ... without having to manually list them in the export list of the importing module!

```
:- module(cultivate_naming, []).

:- reexport( [term_cache, term_metrics, term_graph] ).
```

  - `reexport(+File, ImportList)`
    - Import only predicates from ImportList, like use_module/2
    - ... but additionally re-export all imported predicates

```
:- module(cultivate_naming, []).

:- reexport( module_4, [ pred/1 ] ).
:- reexport( module_5,  except([do_not_use/1]).
```


universitätbonn

Exports of the "same" predicate

SHOW

```
% In file f1:
:- module( m1, [ p/1 ] ).
p(m1).
```

```
% In file f2:
:- module( m2, [ p/1 ] ).
p(m2).
```

```
% In file "ambiguous":
:- module( 'ambiguous', [...] ).
:- use_module(f1).                    % implicitly imports p/1
:- use_module(f2).                    % implicitly imports p/1
...
```

Question:

What happens in "ambiguous"?

Name Clash Rule: The same predicate may not be imported into the same module from two different modules!

universitätbonn

Explicit Import: Avoiding Name Clashes (1)

`use_module(+File, +ImportList)`

- Consults File, importing only the predicates in ImportList
- Also allows for renaming or "import-everything-except":

```
:- use_module(library(lists), [ member/2,
                                , append/2 as list_concat
                                ]).
:- use_module(library(option), except([meta_options/3])).
```

New name in importing module

- Warns if a predicate in ImportList is not in the ExportList of the module defined in File (but imports it nevertheless)
- Only works for static modules (declared in a file)

universitätbonn

Explicit Import: Avoiding Name Clashes (2)

`import(+QualifiedHead)`

- QualifiedHead = Module:Head
- Imports definition of Head from any module Module
- … also from dynamically created modules
- … also for predicates that are not exported

Static use

```
% In file "noexport.pl":
:- module(noexport, []).

p(noexport).
```

```
% In file "explicit_static_import":
:- module(import, []).

% Assumes noexport has been loaded:
:- import(noexport:p/1).

:- p(X), write(X).
```

Dynamic use

```
:- % anytime:
   assert(newm:p(dynamic) ),
   newm:import(noexport:p/1)).
```

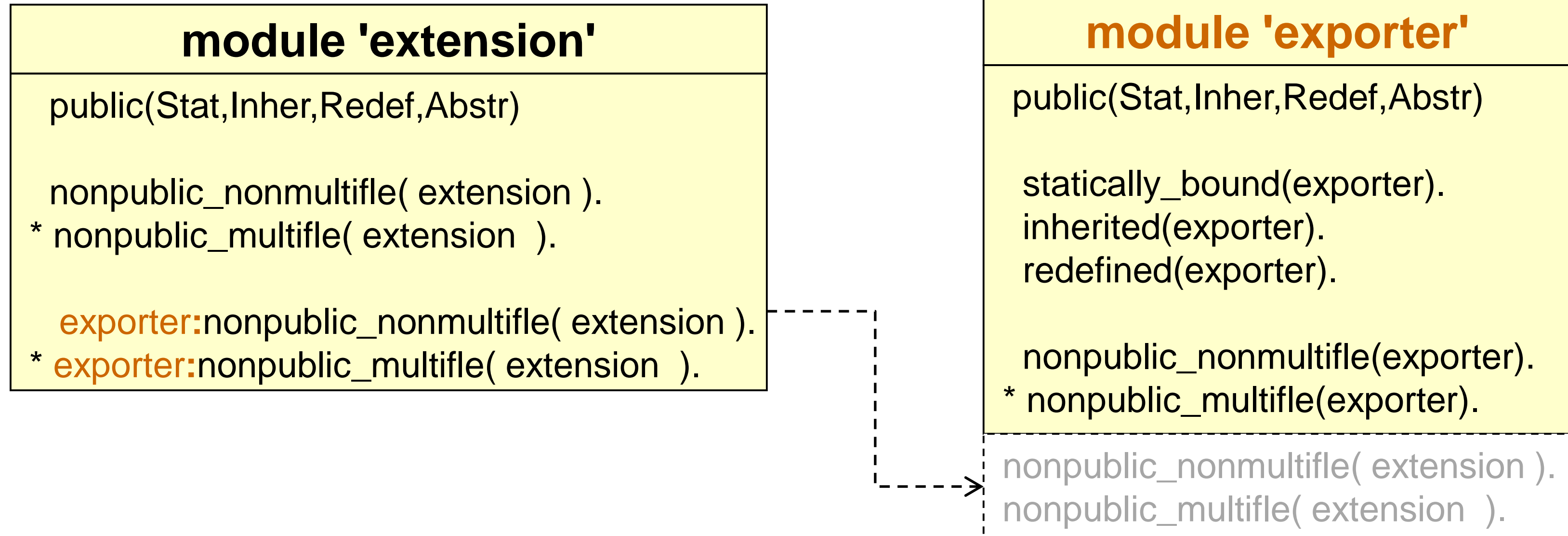
```
% In file "explicit_static_import":
:- module(import, []).

% Assumes noexport has been loaded:


:- noexport:p(X), write(X).
```

A module can declare clauses for others

by prepending a module qualifier to a clause head



- One module “injects” definitions into another one
  - This is akin of “aspect-oriented introduction” mechanism
- May be as powerful and as confusing as aspect-orientation
  - Use it sparingly and document it thoroughly!!!

Summary: Populating Modules

- Predicate `P` is defined in module `M` if it is
  - declared in M's file, without module qualifier in clause heads
    - ⇨ `P :- Body.`
  - declared in another file with clause heads qualified with `M`
    - ⇨ `M:P :- Body.`
  - asserted dynamically into M
    - ⇨ `M:assert((P :- Body))` or `assert(M:(P :- Body))`

- Predicate `P` (that is not defined in `M`) is visible to `M` if it is imported
  - `use_module(+FileOfM)`
    - ⇨ consults FileOfM and imports predicates from the export list of `M`
  - `use_module(+FileOfM, +ImportList)`
    - ⇨ consults File and imports only the predicates in ImportList
  - `import(+QualifiedHead)`
    - ⇨ QualifiedHead = `M:P`
    - ⇨ imports definition of `P` from already created module module `M`

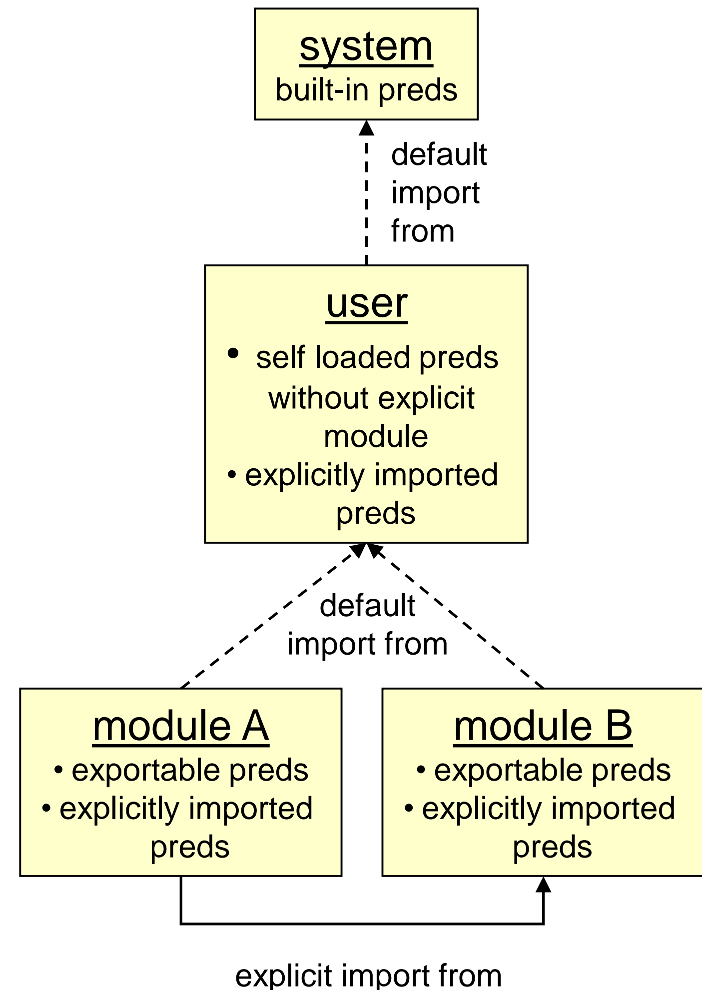
Special Modules

Special modules and default import

Import to “user” ▸ When to do and when to avoid

Special Modules and Default Import

- SWI-Prolog has two special modules
  - "system"
    - ⇨ contains all built-in predicates
  - "user"
    - ⇨ module that the user addresses through the console or from files with no explicit module declaration
- Default import
  - "user" automatically imports from "system"
  - All other modules automatically import from "user"
    - ⇨ Thus they can use all predicates imported into user without explicitly importing them
    - ⇨ This includes the built-in predicates imported from "system" to "user"



Importing to „User“

- Intention
  - distribute common definitions over all program modules
  - … without the need to import them each time explicitly
- Typical uses
  - import your own global libraries / utilities
  - import SWI-Prolog libraries
  - redefine built-in predicates
  - Typically, the load file of a large application looks like this:

```
:- use_module(compatibility).     % load XYZ-Prolog compatibility

:- use_module(                    % load generic parts
        [ error                   % errors and warnings
        , goodies                 % general goodies (library extensions)
        , debug                   % application specific debugging
        , virtual_machine         % virtual machine of application
        , ...                     % other generic stuff
        ]).
```

Importing to „User“:

```
:- module(a,[]).
q(N):- p(N)
```

```
:- module(c,[]).
r(1).
r(2).
```

```
:- module(b,[p/1]).
p(N):- c:r(N)
```

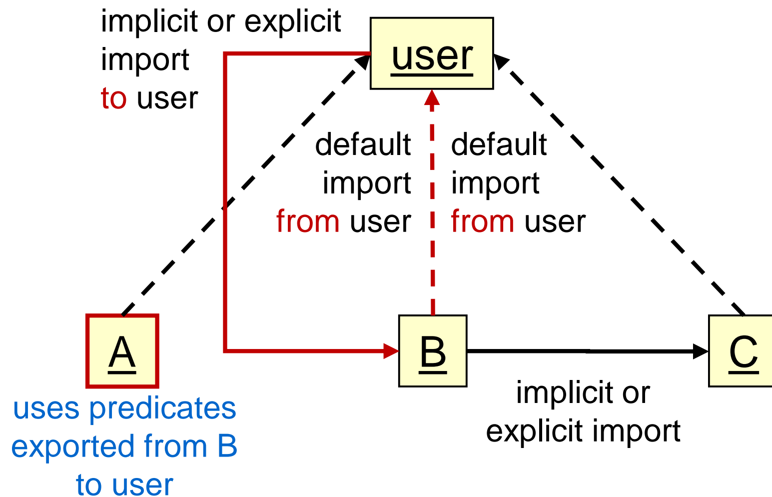
Question:

What is the issue?

Importing to „User“: Use it sparingly!

Import to “user”

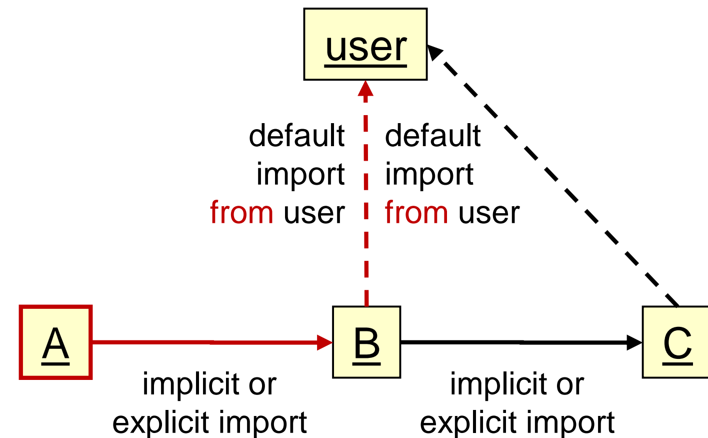
- Only use import to “user” for truly global library predicates!



- It creates bad dependencies
  - Hidden dependencies ☹
  - Cyclic dependencies ☹

Import to real clients

- Better architecture:
  Import from B to its real client A

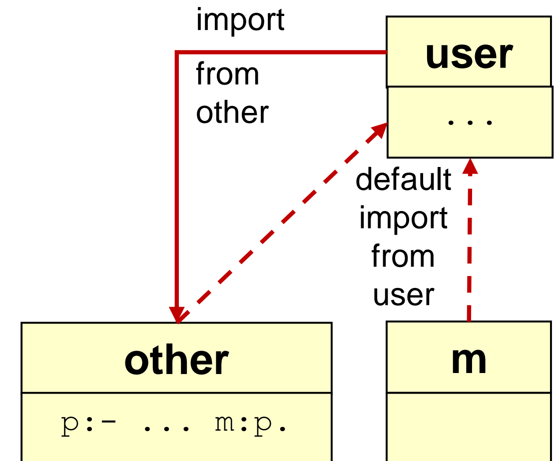


- Better dependencies
  - explicit
  - acyclic

© 2010-2017 Dr. G. Kniesel

More Cyclic Dependencies: Import-Invocation-Forwarding Cycles

- Invocation-Forwarding Cycle
  - Invocation of predicate P to some module M
    - ⇨ ... within a definition of P imported into “user”
  - Forwarding from M back to “user”
    - ⇨ Happens if M has no declaration of P
    - ⇨ Then the default import from “user” is used

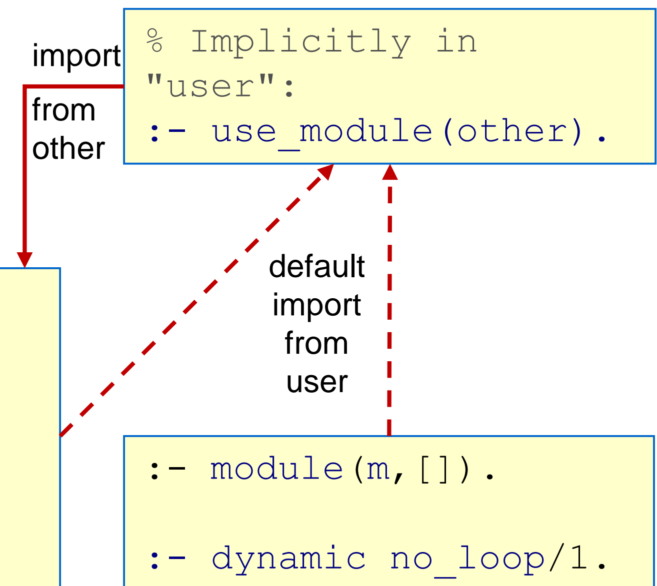


- Example
  - The same as on previous slide, but even harder to detect because the problematic code resided in another module.

```
:- module(other, [loop/1, no_loop/1]).

loop(user).
loop(X) :- m:loop(X).

no_loop(user).
no_loop(X) :- m:no_loop(X).
```



universitätbonn

Summary

- Modules are a namespace mechanism, not for protection
  - They avoid name clashes
  - They control visibility, not accessibility
- Modules have very different semantics in different Prolog implementations
  - See the online documentation of Quintus, YAP, XSB, LPA, ...
- The SWI module system is one of the most advanced ones
  - Separate compilation is supported also for meta-predicates
  - “Module-Transparent” is supported only in SWI- and Eclipse-Prolog
- Fear of non-portability should not stop you from using modules
  - An unmanageable, badly structured program is unusable even on a single platform

Conclusions

- Modules are better than nothing but they have clear limitations
- You need to understand them because they are used a lot
- For you own development, consider using objects upfront
  - See Second part of Chapter 9 “Objects”.