Computational Logic

Developing Programs with a Logic Programming System

System used in the Course

- In the course we use the **Ciao** multiparadigm programming system.
- It supports all the programming paradigms that we will study in the course:
  - For the first parts of the course, *pure logic programming* (LP):
    - With several *search rules:*
      breadth-first, depth-first, iterative deepening, det-first, tabling, ...
    - Also, modules can be set to *pure* mode so that impure built-ins are not accessible to the code in that module.

    This provides a reasonable approximation of pure logic programming (i.e., “Green’s dream” –of course, at a cost in memory and execution time).

  - For other parts of the course Ciao supports:
    - (ISO-)Prolog.
    - Functional programming.
    - Constraint programming (CLP).

Using the Ciao System

- It includes a number of command line and graphical tools for:

  editing / compiling / debugging / verifying / optimizing / documenting / ...

- Main tools:
  - A traditional, command line interactive top level (`ciaosh`).
  - A stand-alone compiler (`ciaoc`) which can generate standalone executables.
  - A build system.
  - Scripts (architecture independent).
  - Source debugger, embeddable debugger, error location, ...
  - An auto-documenter (`LPdoc`).
  - Assertions, with combined static and dynamic checking, of types, modes, determinacy, non-failure, etc. (`CiaoPP`).
  - Assertion-based unit testing and test generation (`LPtest`).

Reading the first slides of the **Ciao tutorial** and the corresponding parts of the **Ciao manuals** regarding the use of the compiler, top-level, debuggers, environment, module system, etc. is suggested at this point.

The Classical Top-Level Shell

- Modern Logic Programming Systems offer several ways of writing, compiling, debugging, and running programs.
- Classical model:
  - ◇ User interacts directly with a top-level shell (includes compiler/interpreter, debugger, etc.).
  - ◇ A prototypical session with a classical Prolog-style, text-based, top-level shell (details are those of the Ciao system, user input in **bold**):

`[37]>` **`ciao`**	Invoke the system
`Ciao X.YY ...`	
`?-` **`use_module('`*`file`*`.pl').`**	Load your program file
`yes`	
`?-` ***`query_containing_variable_X.`***	Query the program
`X =` *`binding_for_X`* **`;`**	See one answer, ask for another using "**;**"
`X =` *`another_binding_for_X`* <**enter**>	Discard rest of answers using <**enter**>
`?-` ***`another query.`***	Submit another query
`?- .......`	
`?-` **`halt.`**	End the session, also with ˆ**D**

Program Load in the Top-Level Shell

- To load a program into the top level use the same commands used as when using code inside a module:
  - `use_module/1` – for loading *modules*.
  - `use_package/1` – for loading *packages* (see later).
  - `ensure_loaded/1` – for loading *user files* (discouraged, modules preferred).

  Note: it is recommended to always use a module declaration, even if empty:

  ```
  :- module(_,_).
  ```

  since it allows the compiler to detect many more errors.

- In summary, the top-level behaves essentially the same as a module.

- Program load can also be *done automatically within the graphical environment*:
  - Open the source file in the graphical environment.
  - Edit it (with syntax coloring, etc.).
  - Load it by typing `C-c l` or using menus.
  - Interact with it in top level.

Top Level Interaction Example

- File member.pl:

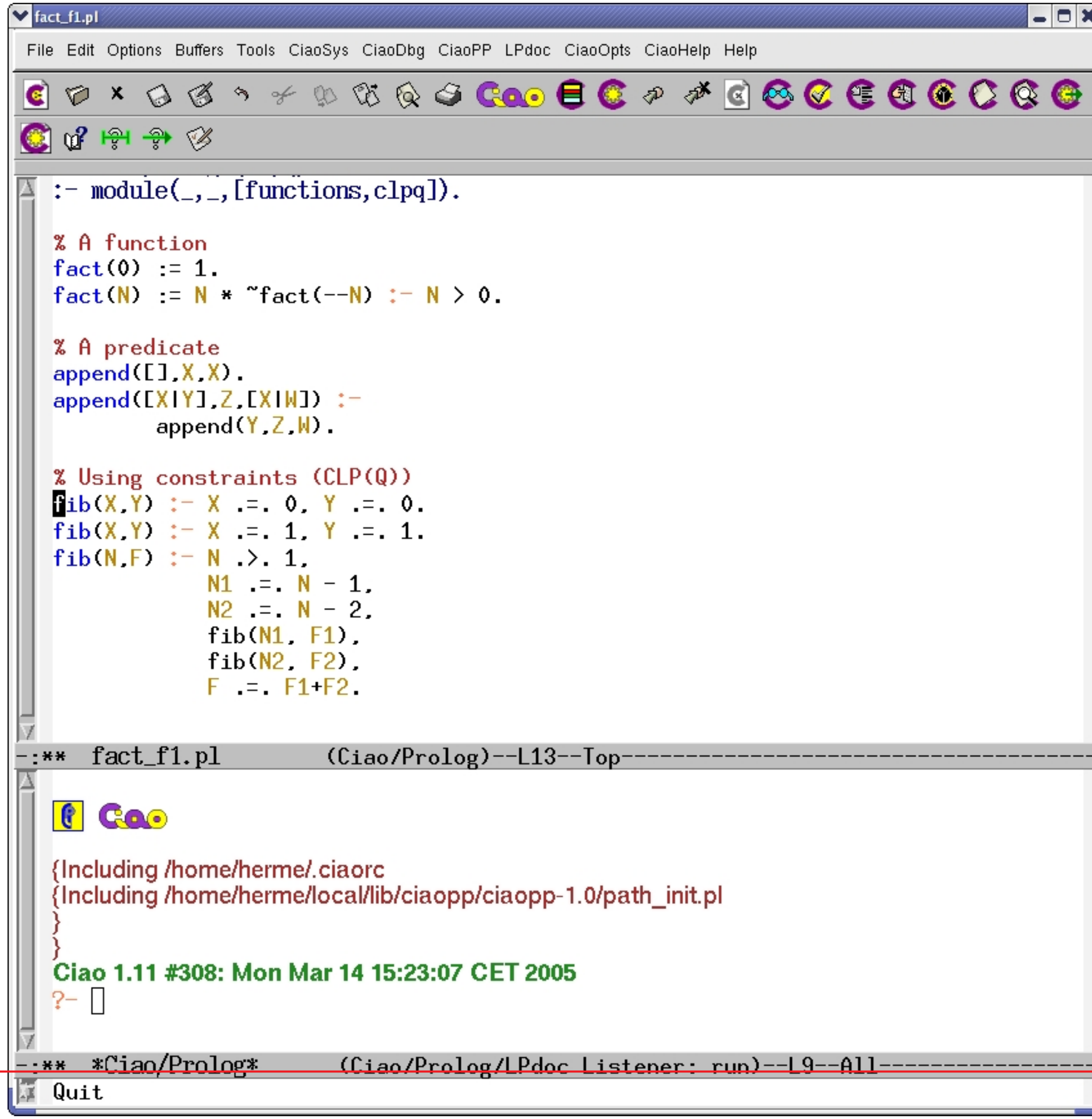
```
:- module(member,[member/2]).

member(X, [X|_Rest]).
member(X, [_|Rest]):- member(X, Rest).
```

- Load into top level and run (issue queries):

```
?- use_module(member).
yes
?- member(c,[a,b,c]).
yes
?- member(d,[a,b,c]).
no
?- member(X,[a,b,c]).
X = a ?   ;
X = b ?   (intro)
yes
```

Ciao Programming Environment: file being edited and top level



Defining a module, its exports, and packages to load

- `:- module(`*module_name*`, `*list_of_exports*`, `*list_of_packages*`).`

  Declares a module of name *module_name*, which exports *list_of_exports* and loads *list_of_packages* (packages are syntactic and semantic extensions).

- Example: `:- module(lists, [list/1, member/2], [functions]).`

- Examples of some standard uses and packages:

  ◇ `:- module(`*module_name*`, [`*exports*`], []).`

  ⇒ Module has access to the kernel language.

  ◇ `:- module(`*module_name*`, [`*exports*`], [`*packages*`]).`

  ⇒ Module has access to the kernel language + some packages.

  ◇ `:- module(`*module_name*`,[`*exports*`], [functions]).`

  ⇒ Adds support for functional programming.

  ◇ `:- module(`*module_name*`,[`*exports*`],[assertions,functions]).`

  ⇒ Adds support for assertions (types, modes, etc.) and func. prog.

Pure modules and search rule selection

- For writing *pure logic programs*, files should start with the following line:

  - `:- module(_,_,['bf/bfall']).`
    To execute in *breadth-first* mode.

  - `:- module(_,_,[]).`
    To execute in *depth-first* mode.

  - Also, the package `pure` can be added so that impure built-ins are not accessible to the code in that module.

(ISO-)Prolog modules

- (ISO-)Prolog:
  - `:- module(module_name, [exports], [iso]).`

    ⇒ module has access to the ISO Prolog predefined predicates.

  - `:- module(module_name,[exports], [classic]).`

    ⇒ "Classic" Prolog module
    (ISO + all other predicates that traditional Prologs offer as "built-ins").

  - Special form:
    `:- module(module_name, [exports]).`
    Equivalent to:
    `:- module(module_name, [exports], [classic]).`

    ⇒ Provides compatibility with traditional Prolog systems.

Defining modules and exports (Contd.)

- Useful shortcuts:
  - ◇ `:- module(_,list_of_exports).`

    If given as "_" module name taken from file name (default).
    Example: `:- module(_, [list/1, member/2]).` (file is `lists.pl`)
  - ◇ `:- module(_,_).`

    If "_" all predicates exported (useful when prototyping / experimenting).
- "User" files:
  - ◇ Traditional name for files including predicates but no module declaration.
  - ◇ Provided for backwards compatibility with non-modular Prolog systems.
  - ◇ Not recommended: they are *problematic* (and, essentially, deprecated).
  - ◇ Much better alternative: use `:- module(_, _).` at top of file.
    - * As easy to use for quick prototyping as "user" files.
    - * Lots of advantages: *much* better error detection, compilation, optimization, ...

Importing from another module

- Using other modules in a module:
  - `:- use_module(`*filename*`).`

    Imports all predicates that *filename* exports.
  - `:- use_module(`*filename*, *list_of_imports*`).`

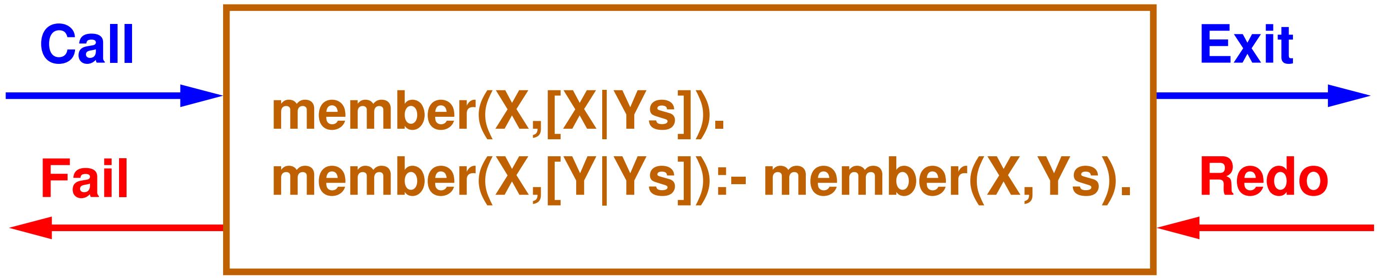
    Imports predicates in *list_of_imports* from *filename*.
  - `:- ensure_loaded(`*filename*`).` —for loading user files (deprecated).
- When importing predicates with the same name from different modules, module name is used to disambiguate:

```
:- module(main,[main/0]).
:- use_module(lists,[member/2]).
:- use_module(trees,[member/2]).

main :-
     produce_list(L),
     lists:member(X,L),
     ...
```

Tracing an Execution with The "Byrd Box Model"

- Procedures (predicates) seen as "black boxes" in the usual way.
- However, simple call/return not enough, due to backtracking.
- Instead, "4-port box view" of predicates:



- Principal events in Prolog execution (*goal* is a unique, run-time call to a predicate):
  - *Call* goal: Start to execute goal.
  - *Exit* goal: Succeed in producing a solution to goal.
  - *Redo* goal: Attempt to find an alternative solution to goal (sol$_{i+1}$ if sol$_i$ was the one computed in the previous *exit*).
  - *Fail* goal: exit with fail, if no further solutions to goal found (i.e., sol$_i$ was the last one, and the goal which called this box is entered via the "redo" port).

Debugging Example

```
Ciao 1.XX ...
?- use_module('/home/logalg/public_html/slides/lmember.pl').
yes
?- debug_module(lmember).
{Consider reloading module lmember}
{Modules selected for debugging: [lmember]}
{No module is selected for source debugging}
yes
?- trace.
{The debugger will first creep -- showing everything (trace)}
yes
{trace}
?-
```

- Much easier: open file in Emacs and type `C-c d` (or use the CiaoDbg menu).
- This loads the current module in *source debug* mode, i.e., the debugger traces the position in the source file.

Debugging Example (Contd.)

```
?- lmember(X,[a,b]).
   1   1   Call: lmember:lmember(_282,[a,b]) ?
   1   1   Exit: lmember:lmember(a,[a,b]) ?
X = a ? ;
   1   1   Redo: lmember:lmember(a,[a,b]) ?
   2   2   Call: lmember:lmember(_282,[b]) ?
   2   2   Exit: lmember:lmember(b,[b]) ?
   1   1   Exit: lmember:lmember(b,[a,b]) ?
X = b ? ;
   1   1   Redo: lmember:lmember(b,[a,b]) ?
   2   2   Redo: lmember:lmember(b,[b]) ?
   3   3   Call: lmember:lmember(_282,[]) ?
   3   3   Fail: lmember:lmember(_282,[]) ?
   2   2   Fail: lmember:lmember(_282,[b]) ?
   1   1   Fail: lmember:lmember(_282,[a,b]) ?
no
```

Options During Tracing

`h`	Get help — gives this list (possibly with more options)
`c`	Creep forward to the next event Advances execution until next call/exit/redo/fail
`intro`	(same as above)
`s`	Skip over the details of executing the current goal Resume tracing when execution returns from current goal
`l`	Leap forward to next “spypoint” (see below)
`f`	Make the current goal fail This forces the last pending branch to be taken
`a`	Abort the current execution
`r`	Redo the current goal execution very useful after a failure or exit with weird result
`b`	Break — invoke a recursive top level

- Many other options in modern Prolog systems.
- Also, graphical and source debuggers available in these systems.

Spypoints (and breakpoints)

- `?- spy foo/3.`

  Place a spypoint on predicate `foo` of arity 3 – always trace events involving this predicate.

- `?- nospy foo/3.`

  Remove the spypoint in `foo/3`.

- `?- nospyall.`

  Remove all spypoints.

- In many systems (e.g., Ciao) also *breakpoints* can be set at particular program points within the graphical environment.

Debugger Modes

- `?- debug.`
  Turns debugger on. It will first leap, stopping at spypoints and breakpoints.
- `?- nodebug.`
  Turns debugger off.
- `?- trace.`
  The debugger will first creep, as if at a spypoint.
- `?- notrace.`
  The debugger will leap, stopping at spypoints and breakpoints.

Creating Executables

- Most systems have methods for creating 'executables':
  - ◇ Saved states (`save/1`, `save_program/2`, etc.).
  - ◇ Stadalone compilers (e.g., `ciaoc`).
  - ◇ Scripts (e.g., `prolog-shell`).
  - ◇ "Run-time" systems.
  - ◇ etc.
- E.g., Ciao's compiler allows generating standalone executables, which can be:
  - ◇ eager dynamic load
  - ◇ lazy dynamic load
  - ◇ static (portable, architecture-independent –needs minimal Ciao installed)
  - ◇ fully static/standalone (fully portable, but architecture-dependent).