Computational Logic

Developing Programs with a Logic Programming System

Our Development Environment: The Ciao System

- We use the (ISO-Prolog subset of the) Ciao multiparadigm programming system.
- In particular, the Ciao system offers both command line and graphical environments for editing, compiling, debugging verifying, optimizing, and documenting programs, including:
  - ◇ A traditional, command line interactive top level.
  - ◇ A stand-alone compiler (`ciaoc`).
  - ◇ Compilation of standalone executables, which can be:
    - * eager dynamic load
    - * lazy dynamic load
    - * static (without the engine –architecture independent)
    - * fully static/standalone (architecture dependent)
  - ◇ Prolog scripts (architecture independent).
  - ◇ Source debugger, embeddable debugger, error location, ...
  - ◇ Auto-documenter.
  - ◇ Compile-time checking of assertions (types, modes, determinacy, non-failure, etc. ...) and static debugging, etc.!

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

Programmer Interface: The Classical Top-Level Shell

- Modern Prolog compilers offer several ways of writing, compiling, and running programs.
- Classical model:
  - User interacts directly with top level (includes compiler/interpreter).
  - 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 1.11 #211: Thu Mar 18 15:28:12 CET 2004`	
`?-` **use_module(file).**	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**

Traditional (“Edinburgh”) Program Load

- Compile program (much faster, but typically no debugging capabilities):

```
?- compile(file).
```

- Consult program (interpreted, slower, used for debugging in traditional systems):

```
?- consult(file).
?- [file].
```

- Compiling/consulting several programs:

```
?- compile([file1,file2]).
?- [file1,file2].
```

- Enter clauses from the terminal (not recommended, except for quick hacks):

```
?- [user].
| append([],Ys,Ys).
| append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
| ^D
{user consulted, 0 msec 480 bytes}
yes
?-
```

Ciao Program Load

- Most traditional (“Edinburgh”) program load commands can be used.
- But more modern primitives available which take into account module system. *Same commands used as in the code inside a module:*
  - `use_module/1` – for loading modules.
  - `ensure_loaded/1` – for loading user files.
  - `use_package/1` – for loading packages (see later).
- In summary, top-level behaves essentially like a module.
- In practice, *done automatically within 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

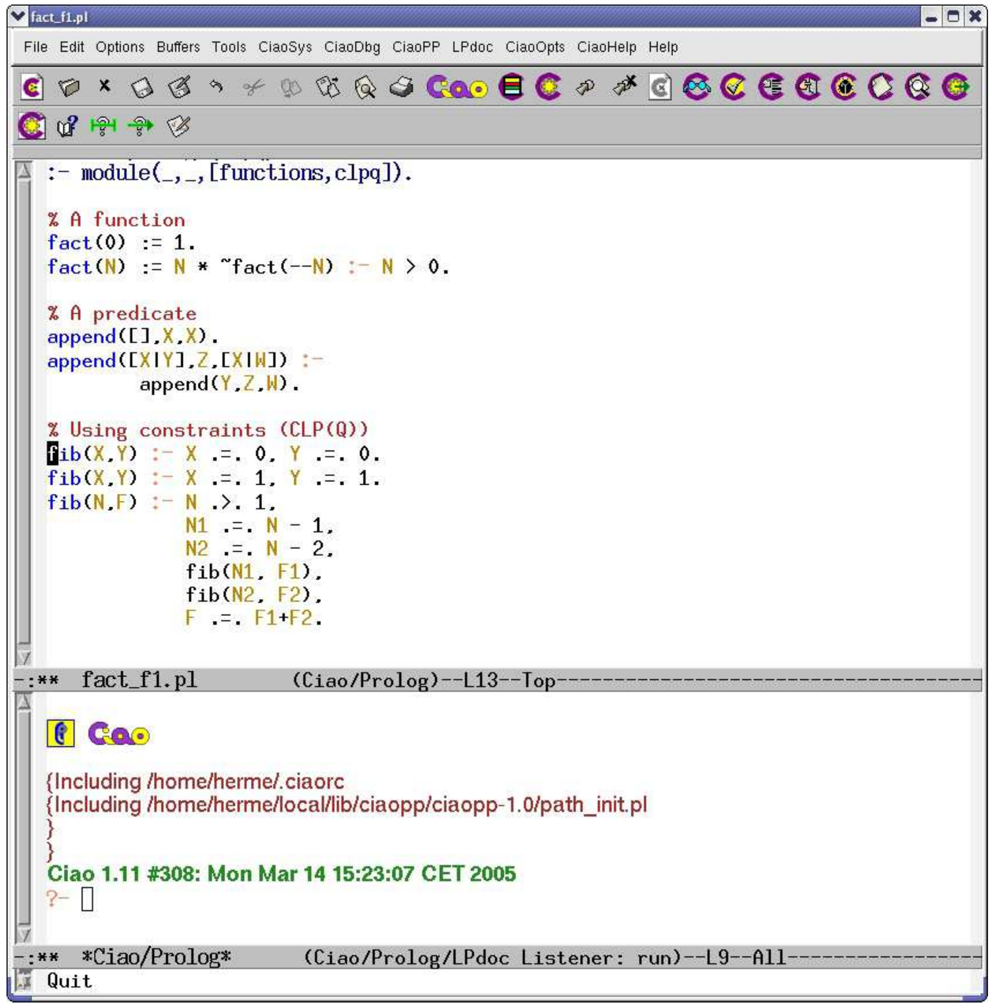
- File `member.pl`:

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

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

```
?- 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



Top Level Interaction Example

- File `pets.pl` contains:

```
:- module(_,_,[bf]).
```

+ *the pet example code as in previous slides.*

- Interaction with the system query evaluator (the “top level”):

```
Ciao 1.13 #0: Mon Nov 7 09:48:51 MST 2005
?- use_module(pets).
yes
?- pet(spot).
yes
?- pet(X).
X = spot ? ;
X = barry ? ;
no
?-
```

The Ciao Module System

- Ciao implements a module system [?] which meets a number of objectives:
  - ◇ High extensibility in syntax and functionality:
    allows having pure logic programming and many extensions.
  - ◇ Makes it possible to perform modular (separate) processing of program components (without “makefiles”).
  - ◇ Greatly enhanced error detection (e.g., undefined predicates).
  - ◇ Facilitates (modular) global analysis.
  - ◇ Support for meta-programming and higher-order.
  - ◇ Predicate based-like, but with functor/type hiding.

  while at the same time providing:
  - ◇ High compatibility with traditional standards (Quintus, SICStus, ...).
  - ◇ Backward compatible with files which are not modules.

Defining modules and exports

- `:- 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 uses (pure) kernel language.

  ◇ `:- module(module_name, [exports], [packages]).`
  ⇒ Module uses kernel language + some packages.

  ◇ `:- module(module_name,[exports], [functions]).`
  ⇒ Functional programming.

  ◇ `:- module(module_name,[exports],[assertions,functions]).`
  ⇒ Assertions (types, modes, etc.) and functional programming.

Defining modules and exports (Contd.)

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

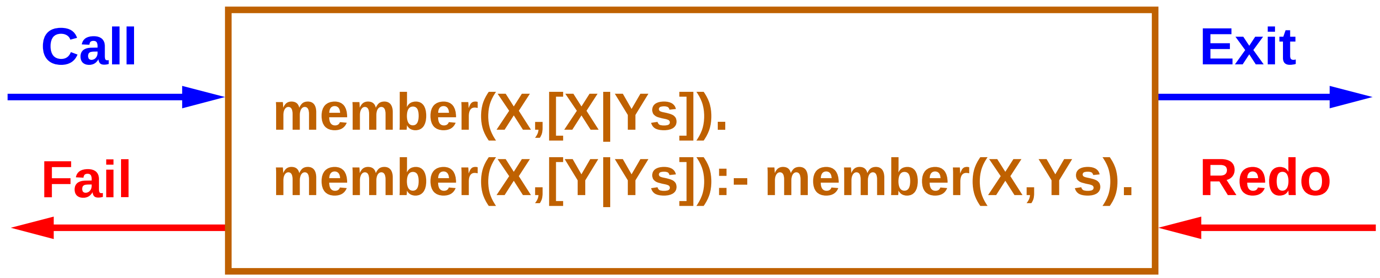
- Using other modules in a module:
  - ◇ `:- use_module(`*filename*`).`
    Imports all predicates that *filename* exports.
  - ◇ `:- use_module(`*filename*`,`*list_of_imports*`).`
    Imports predicates in *list_of_imports* from *filename*.
  - ◇ `:- ensure_loaded(`*filename*). —for loading user files (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 1.13 #0: Fri Jul 8 11:46:55 CEST 2005
?- 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 and type `C-c d` (or use CiaoDbg menu).

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.

Running Pure Logic Programs: the Ciao System's bf/af Packages

- We will be using *Ciao*, a multiparadigm programming system which includes (as one of its "paradigms") a *pure logic programming* subsystem:
  - ◇ A number of *fair* search rules are available (breadth-first, iterative deepening, ...): we will use "breadth-first" (`bf` or `af`).
  - ◇ Also, a module can be set to *pure* mode so that impure built-ins are not accessible to the code in that module.
  - ◇ This provides a reasonable first approximation of "Greene's dream" (of course, at a cost in memory and execution time).
- Writing programs to execute in `bf` mode:
  - ◇ All files should start with the following line:

    `:- module(_,_,[bf]).`   `(or :- module(_,_,['bf/af']).)`

    or, for "user" files, i.e., files that are not modules: `:- use_package(bf).`
  - ◇ The *neck* (arrow) of rules must be `<-` .
  - ◇ Facts must end with `<- .` .