Today

- Higher order logic programming
  - Extending the logic
  - Extending the search
  - Examples
- Examinable material.

Recall:

definite clause logic is extended by adding:

- A type structure: syntax items have user declared types; there is a special type o of propositions; functions from type t_1 to type t_2 have type $t_1 \longrightarrow t_2$. Predicates on objects of type t have type $t \longrightarrow o$.
- Implication as a new connective: `G => H`.
- Universal quantification (in programs and queries).
- Existential quantification (just in queries).

Quantifiers

Use the following to express quantification:
for $\forall x\ A$, use a lambda term to express the binding of the variable, and then a constant `pi` to quantify. Thus a goal

$$\forall x\ x = x$$

becomes

```
pi ( x\ (x = x) )
```

and $\forall P\ P(0) \rightarrow P(0)$ becomes

```
pi ( p\ ( (p 0) => (p 0) )).
```

Search

What search operations are used to solve queries?
There are search operations associated with different connectives in the goal; for example:

- To solve `D => G`, add `D` to the program clauses, and solve `G`.
- To solve `pi (x\ G x)`, pick a new parameter `c` (i.e. a constant that does not appear in the current problem), and solve `G c`.
- Analogously to Prolog, to solve atomic `G`, find a program clause whose head can be unified with `G`, and solve the body with the unifier applied.

Example (McCarthy)

Try to formalise the following:

> *Something is sterile if all the bugs in it are dead.*
> *If a bug is in an object which is heated, then the bug is dead.*
> *This jar is heated.*
> *So, the jar is sterile.*

This is a natural and simple argument, and we want to express in directly. We could use full predicate calculus (but search is hard there).

In the language above, we get as follows.

```
kind i        type.
type sterile (i -> o).
type in      (i -> i -> o).
type heated  (i -> o).
type bug     (i -> o).
type dead    (i -> o).
type j        i.

sterile Jar :- pi x\ ( (bug x) =>
                 (in x Jar) => (dead x) ).
dead X      :- heated Y, in X Y, bug X.
heated j.

?- sterile X.

X = j
```

Often we want to do similar things for different predicates we are reasoning about. For example, the standard `ancestor/2` predicate is defined as a transitive extension of `parent/2`:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Similarly, get less than from the successor relation, descendent from child ...

Now, do this once and for all:

```
type  trans (A -> A -> o) -> (A -> A -> o).

trans Pred X Y :- Pred X Y.
trans Pred X Z :- Pred X Y, trans Pred Y Z.
```

and define `ancestor` via

```
ancestor X Y :- trans parent X Y.
```

Here the predicate `parent` is used as an argument to the `trans` procedure.

Map

We can exploit the higher-order features to write a `map` predicate; this takes a list and a function, and returns the result of applying the function to each member of the list.
Because we have relations available, we can also think of mapping predicates (what could this mean?).

```
type  mapfun    (A -> B) -> list A -> list B -> o.
type  mappred   (A -> B -> o) -> list A -> list B -> o
type  for_each  (A -> o) -> list A -> o.
```

Because we are in a relational, rather than functional, setting, what is available with the typing `A -> B` is limited.
Here's the definition of `mapfun`:

```
mapfun F nil nil.
mapfun F (X :: K) ((F X) :: L) :- mapfun F K L.
```

Notice the use of `F` as a variable for a function –
this goes beyond Prolog, and keeps reversibility.

Running mapfun

We can query for “output” list, or “input” list:

```
?-  mapfun  (x\ x + x) (3 :: 4 :: 5 :: nil) Y.

Y = 3 + 3 :: 4 + 4 :: 5 + 5 :: nil

?- mapfun (x\ (x + x))  X ((3 + 3) :: ( 8 + 8) :: nil)
    .

X = 3 :: 8 :: nil
```

and even query for the function:

```
?- mapfun F (3 :: 8 :: nil) ((3 + 3) :: ( 8 + 8) ::
    nil).

F = x\ x + x
```

Here's a definition for `mappred`;
again note the variable standing for a predicate:

```
mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.
```

What will happen on back-tracking?

Suppose we have some background predicate:

```
likes jane moses.        likes john peter.
likes jane john.         likes james peter.
```

mappred reversible

```
?- mappred likes (jane :: john :: nil) L.

L = moses :: peter :: nil ;
L = john :: peter :: nil ;
no more solutions

?- mappred likes X (john :: peter :: nil).

X = jane :: john :: nil ;
X = jane :: james :: nil ;
no more solutions
```

"Difference" lists in λProlog

Recall standard Prolog difference lists, which give an efficient way to do some list operations — and also need care in use.

In a higher-order setting, we can achieve the same efficiency gain, but remain declarative, and indeed retain reversibility.

The idea is that a normal list:

```
[1,3,5]
```

is represented by a **function** that maps any list to the list with `[1,3,5]` prepended; in Haskell syntax:

```
\x -> (1 : 3 : 5 : x)
```

Difference lists in λProlog (2)

We can define functions to convert between the normal representation and this "difference" list version, and get an efficient way to append lists. Here are the type declarations; `list T` is a polymorphically typed list, and the difference lists have type `list T -> list T`:

```
type mkDList  list T -> (list T -> list T) -> o.

type append_dl
     (list T -> list T) ->
       (list T -> list T) ->
         (list T -> list T) -> o.
```

Difference lists in λProlog (2)

These are implemented as follows:

```
% mkDList/2 uses standard recursion

mkDList nil (x\ x).
mkDList (H::T) (x\ H::(T' x)) :- mkDList T T'.
```

This works in both directions:

```
?- mkDList (1::3::5::nil) L.

L = x\ 1 :: 3 :: 5 :: x

?- mkDList L (x\ 1::3::5::x).

L = 1 :: 3 :: 5 :: nil
```

Difference list append

Now think a bit what corresponds to appending lists in this representation:

```
append_dl L M (x\ L (M x)).
```

So append is done via unification; we get reversibility here (there can be several unifiers, unlike in the usual Prolog situation).

Reversible append

```
?- mkDList (1::3::5::nil) L, append_dl L L Y.

L = x\ 1 :: 3 :: 5 :: x
Y = x1\ 1 :: 3 :: 5 :: 1 :: 3 :: 5 :: x1
```

Reverse direction:

```
?- mkDList (1::3::5::nil) L, append_dl X Y L.

L = x\ 1 :: 3 :: 5 :: x
X = x1\ 1 :: 3 :: 5 :: x1
Y = x2\ x2 ;

L = x3\ 1 :: 3 :: 5 :: x3
X = x4\ 1 :: 3 :: x4
Y = x5\ 5 :: x5 ;

% and another two solutions
```

Examinable material: Programming

- Material covered in LPN, ch. 1-6:
- Terms, variables, unification (+/- occurs check)
- Arithmetic expressions/evaluation
- Recursion, avoiding non-termination
- Programming with lists and terms
- Expect ability to solve problems similar to those in tutorial programming exercises (or textbook exercises)

Examinable material: Programming (2)

- Material covered in LPN, ch. 7-11:
- Definite clause grammars
- Difference lists
- Non-logical features ("is", cut, negation, assert/retract)
- Collecting solutions (findall, bagof, setof)
- Term manipulation (var, =.., functor, arg, call)
- Expect ability to explain concepts & use in simple Prolog programs

Examinable material: Programming (3)

- Advanced topics (Bratko ch. 11-12, 14, 23)
- Search techniques (DFS, IDS, BFS)
- Symbolic programming & meta-programming
- Expect understanding of basic ideas
- **not** ability to write large programs from scratch under time pressure.

Some exam info

- Programming exam: 2 hours
- DICE machine with SICSTUS Prolog available
- (Documentation won't be, but exam will not rely on memorizing obscure details)
- Sample exam on course web page

Examinable material: Theory

- Definite clauses, syntax and semantics for the propositional case
- Backchain inference rule for propositional case
- Soundness and completeness of inference system with respect to logical consequence
- Proof search as inference procedure, the Prolog search procedure
- Notion of decision procedure.
- Monotone functions and fixed points, least fixed point
- Least fixed point for propositional definite clauses

Examinable material: Theory (2)

- Completeness of inference procedure wrt inference system
- predicate calculus, syntax, informal semantics, definite clauses
- substitution, unification, most general unifier, occurs check
- backchain inference rule, Prolog search and its properties
- **not** general existence of lfp
- the result that backchain is complete for definite clauses (**not** the proof)
- what Herbrand model is, and least Herbrand model.
- result that complete decision procedure for inference in definite clauses is impossible (not proof)

Examinable material: Theory (2)

- Algorithm for basic definite clause interpreter
- object language vs meta-language distinction
- Prolog meta-predicates and why they do not fit the declarative reading
- negation by failure and the closed world assumption
- inference using CWA as a form of non-monotonic reasoning
- Clark completion algorithm
- Higher-order logic programming and dealing with the extensions

Summary

- Higher order logic programming: λProlog
- Higher-order predicates combined with search
- Examinable material.