Section 1

Topics

1. Logic Programming
Logic and Programs

- “All men are mortal; Socrates is a man; Hence Socrates is mortal”
  \[ \forall X. \text{man}(X) \Rightarrow \text{mortal}(X) \]
  \[
  \text{man}(\text{socrates})
  \]

- Predicate logic
  - Predicates (e.g. man, mortal) which define sets.
  - Atoms (e.g. socrates) which are data values
  - Variables (e.g. X) which range over data values
  - Rules (e.g. \( \forall X. \text{man}(X) \Rightarrow \text{mortal}(X) \)) which define relationships between predicates.

<table>
<thead>
<tr>
<th>mortal(X) :- man(X).</th>
<th>let isMortal(x) = isMan(x);</th>
</tr>
</thead>
<tbody>
<tr>
<td>man(socrates).</td>
<td>let isMan(x) = (x = socrates);</td>
</tr>
</tbody>
</table>

Logic Programs

?- mortal(socrates).
  yes

mortal(X) :- man(X).
  man(socrates).

?- mortal(X).
  X=socrates ;
  no

Relations and Logic Programs

- Unary predicates (e.g. man, mortal) define sets.
  Predicates with higher arity (binary, ternary etc) define relations. Example:
    \[
    \text{flight}(\text{jfk}, \text{dfw}).
    \text{flight}(\text{dfw}, \text{lax}).
    \text{flight}(\text{stl}, \text{jfk}).
    \text{flight}(\text{stl}, \text{dfw}).
    \text{flight}(\text{lga}, \text{stl}).
    \]

- Facts: sets and relations whose definitions do not depend on anything else. (e.g. \text{man(socrates)}).
  “extensional data base” (EDB)
**Rules** define *computed* sets and relations (e.g. mortal).

“intensional data base” (IDB) relations

\[
\text{canFly(Source, Dest)} :- \text{flight(Source, Dest)}. \\
\text{canFly(Source, Dest)} :- \text{flight(Source, Stopover)}, \text{canFly(Stopover, Dest)}.
\]

**Programming with Logic**

- Data structures:
  - Atomic data such as *socrates*, *lga*, etc.
  - Data structures by constructing *terms* (tree structures):
    - `[]`: nil list
    - `[X|Xs]`: list with `X` as its head and `Xs` as its tail
    - `prog(P, D, S)`: a structure with `prog` as the *root* symbol, and `P`, `D`, and `S` as its children
  - Example programs: `append(Xs, Ys, Zs)`: `Xs`, `Ys`, and `Zs` are lists such that `Zs` is the concatenation of `Xs` and `Ys`.

\[
\text{append}([], Ys, Zs) :- Zs = Ys. \\
\text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs).
\]

**From Functional to Relational Programming**

\[
\text{let rec append(1, ys)} = \\
\phantom{\text{let rec append(1, ys)} = }\text{match 1 with} \\
\phantom{\text{let rec append(1, ys)} = }[\] \rightarrow ys \\
\phantom{\text{let rec append(1, ys)} = }x::xs \rightarrow x::\text{append}(xs, ys)
\]

\[
\text{let rec reverse l =} \\
\phantom{\text{let rec reverse l =}}\text{match l with} \\
\phantom{\text{let rec reverse l =}}[\] \rightarrow [\] \\
\phantom{\text{let rec reverse l =}}x::xs \rightarrow \text{append(\{(reverse xs), [x]\})}
\]
SML and Prolog

fun rev1(x::xs, ys) = 
  rev1(xs, x::ys) 
| rev1(nil, ys) = ys
fun rev(xs) = rev1(xs, [])

datatype tree =
  Node of int * tree * tree 
| Leaf of int;

fun search(Node(i,l,r), j) = 
  if (j<=i) then search(l,j) 
  else search(r,j)
| search(Leaf(i), j) = i = j;

Syntax of Prolog Programs

- **Names:**
  - Variable names start with uppercase letters
  - Predicate names start with lowercase letters
  - Data constructors (called “function symbols” and “constants”) start with lowercase letters or enclosed in single quotes

- **Data structures:** a term (a tree of symbols) built using function symbols and variables.
  - lgs
    - [1] (same as [ 1  |  | ] )
    - [1,2] (same as [1  |  | 2  |  | ] )
    - f(g(a))
    - f(g(h(X)))
    - f(X, g(X))
    - (lga, jfk)

Syntax of Prolog Programs (Contd.)

- **Atom:** a term built with function symbols, predicate symbols and variables.
  Example: append([X|Xs], Ys, [X|Zs])

- **Clauses:** of the form lhs :- rhs.
  *Note the trailing period.*
  - Clause head: An atom
  - Clause body: a comma-separated sequence of atoms.
  - Facts: clauses with empty bodies.
    Written as lhs.
  - Rules: clauses with non-empty bodies.

- **Program:** a sequence of clauses.
- **Query:** an atom.
Logic Programming

Arithmetic in Prolog

- Use of “=” simply constructs or inspects term structures.
  - For example, \( X = 1 + 2 \) binds \( X \) to term \( 1+2 \).
- Binary operator “is” should be used to evaluate arithmetic expressions.
  - For example, \( X \text{ is } 1 + 2 \) binds \( X \) to \( 3 \).
  - Rhs of “is” must be ground when the operator is evaluated.
- Expressions mix real and integer arithmetic, lifting values to real whenever necessary.
- Arithmetic comparison operators: \( =, \neq, <, >, =<, >= \) (Note the syntax of “less-than-or-equal-to” etc.)

\[
\text{length(\[] , 0).} \\
\text{length([X|Xs], N) :- length(Xs, M), N is M+1.}
\]

How Prolog Works

Prolog attempts to check if the given query \( q \) is true by
1. Is there a clause whose left hand side corresponds to \( q \)?
2. If not, \( q \) is false (we say that \( q \) fails)
3. If there is such a clause, say \( l : - r_1, r_2, \ldots, r_n \)
   - Now check if all of \( r_1, r_2, \ldots \) are true.
   - If so, \( q \) is true (we say that \( q \) succeeds)
   - If not, repeat step (3) until there is no matching clause
- Clauses are tried in the order they appear in the program.
- If more than one clause applies, they are tried one after another until the goal succeeds

append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).

append([a, b], [c], Z) \hspace{1cm} \text{Clause 2}
append([b], [c], Z') \hspace{1cm} Z = [a|Z'] \hspace{1cm} \text{Clause 2}
append([], [c], Z) \hspace{1cm} Z'=[[b|Z'], Z = [a|Z'] \hspace{1cm} \text{Clause 1}
Z''=[c] \hspace{1cm} Z'=[b|Z''] \hspace{1cm} Z = [a|Z'] \hspace{1cm} \text{Simplify}
Z=[a,b,c]
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).

append(U, V, [a,b])

Clause 1, Clause 2
(1) U=[] , V=[a,b]
(2) append(U',V,[b]), U=[a|U']
   U=[a], V=[b]
   U'=[[] , V=[[] , U’=[b|U’] , U=[a|U’]
   U’=[[] , V=[[] , U’=[b|U’] , U=[a|U’]
   U=[a,b], V=[]

Unification

- **Unification** is the operation to make two data structures identical (i.e. “unify” them).
- Predefined binary predicate = may be used to unify terms.
  - a = a succeeds, a = b fails, X = a succeeds after binding X to a.
  - f(X) = f(a) succeeds after binding X to a.
  - g(a) = f(a), f(a) = f(b), f(a,b) = f(b,a) fail.
  - ?: f(X) = f(a), X = b.
  - ?: f(X,a) = f(b,Y).
  - ?: f(X,a) = f(b,X).

A clause is applicable if the query (also called a **goal** or **subgoal**) unifies with the
left hand side of the clause.

Unification (Contd.)

- **Substitution**: a function that maps variables to values (terms).
- An unifier of two terms $t_1$ and $t_2$ is a substitution over variables of $t_1$ and $t_2$ that make
  them identical.
  - The substitution $\{X \rightarrow b, Y \rightarrow a\}$ is an unifier of $f(X,a)$ and $f(b,Y)$.
  - The substitution $\{X \rightarrow b, Y \rightarrow a, Z \rightarrow c, W \rightarrow c\}$ is an unifier of $f(X,a,Z)$ and $f(b,Y,W)$.
  - The substitution $\{X \rightarrow b, Y \rightarrow a, Z \rightarrow d, W \rightarrow d\}$ is an unifier of $f(X,a,Z)$ and $f(b,Y,W)$.
  - The substitution $\{X \rightarrow b, Y \rightarrow a, Z \rightarrow W\}$ is an unifier of $f(X,a,Z)$ and $f(b,Y,W)$.
  - Called the **most general unifier**
  
  During query evaluation, clauses are selected by computing the most general unifier.
A Simple Prolog Interpreter: Types

```haskell
let rec unify: subst -> term -> term -> subst = 
  fun subst t1 t2 = match (t1, t2) with
  | (Var(x), _) -> add_subst subst x t2
  | (_, Var(y)) -> add subst y t1
  | (Nvar(c,t1s), Nvar(d,t2s)) ->
    if c=d then unify_list subst t1s t2s
    else raise Unif_fail
and unify_list subst 11 12 = fold_left2 unify subst 11 12
and add_subst: subst->var->term->subst = fun subst x t =
  try let t' = assoc x subst in unify subst' t' t
  with Not_found -> if t<>Var(x) then (x,t)::subst else subst
```

More about unification ...

- Given two terms $t_1$ and $t_2$ containing variables $\bar{x}_1$ and $\bar{x}_2$,
  $t_1$ and $t_2$ are unifiable if and only if the logical formula $\exists \bar{x}_1, \bar{x}_2 \ t_1 = t_2$ is satisfiable.
- Unification procedure computes a solution to the formula, i.e., a valuation for $\bar{x}_1$ and $\bar{x}_2$ that makes this formula true.
- Every solution to the formula is an instance of the solution computed by `unify` — the most general unifier property.
- **Occurs-check**: Note that $\forall X \ X \neq f(X)$.
  - So, in general, we need to check if $X$ occurs in $t$ before taking $t$ as a substitution for $X$.
  - Omitted in Prolog because it has severe impact on performance
  - Interestingly, `unify` terminates even when it computes such cyclic substitutions!
More about unification ... (Continued)

- **Unification** is a *constraint-solving procedure* for equality constraints over terms.
- Many problems can be modeled in terms of such constraints
  
  **Type inference:**
  - For each identifier $i$, associate a variable $T_i$ that holds its type.
  - Constraints on $T_i$'s types are inferred from each use of $i$, whether it be as argument to a function, in an equality or match operation, etc.
  - Most general unifiers yield the most general types for each identifier.

  **Logic program evaluation:**
  - Each “call” introduces a constraint between actual and formal parameters.
  - Most general unifiers correspond to the most general solutions to the query

---

**Type Inference Example**

```
let h y = 0

let g x =
  if (1 x)
  then (h x)
  else (g (x+1))

let rec f t =
  match t with
  | [] -> []
  | z::zs -> (g z)::(f zs)
```

---

**Query evaluation in Prolog**

- The query evaluation procedure in Prolog (called clause resolution) uses *backtracking* search.

  - Given a query (goal), a clause is *applicable* if its head (lhs) unifies with the query.

  - When more than one clause is applicable evaluation,
    - the first clause is selected, and query evaluation continues with the body of the clause
    - ... but we may come back to try the remaining clauses if further query evaluation using the first clause fails.

  - Clauses applicable but not yet tried at any point are remembered *and are tried upon backtracking*.

- **Alternative strategy:** Eagerly compute all solutions
  - Let us write a simple interpreter for this strategy
A simple Prolog interpreter to compute all solutions

```
let rec call: (prog: clause list) (env:env) (goal:goal): env list =
  let paths = (map (find_path goal env) prog) in
  let viable_paths = filter (fun (_, (bp, _)) -> bp > 0) paths
  in exec_paths prog viable_paths

and exec_paths prog paths = match paths with
  | [] -> []
  | p1::ps -> (append (exec_path prog p1) (exec_paths prog ps))

and exec_path: program -> path -> env list =
  fun prog (glist, env) = match glist with
  | [] -> [env]
  | goal::goals ->
    let envs = call prog env goal in
    let newpaths = map (fun e -> (goals, e)) envs
    in (flatten (map (exec_path prog) newpaths))
```

A Prolog interpreter to compute all solutions (Continued)

```
let find_path: goal -> env -> clause -> path =
  fun goal (bp, subst) clause =
    let (hd::body) = alloc_locals bp clause in
    try let subst' = assign_to_formals hd goal subst
        in (body, (bp + (numvars hd) + (numvars list body), subst'))
        with Unif_fail -> ([], (-1, subst))

let assign_to_formals hd goal subst: subst = unify subst hd goal

let rec alloc_locals: int -> term list -> term list =
  fun bp ts = let alloc_local t = match t with
    | Var(i) -> Var(bp+i)
    | Nvar(c, ts) -> Nvar(c, alloc_locals bp ts)
  in map alloc_local ts
```

Implementing Backtracking

- Simply replace eager evaluation used in the interpreter with lazy evaluation!
- But OCaml does not support lazy evaluation
  - Use a language like Haskell that supports lazy evaluation
  - Employ a simple trick to achieve lazy evaluation in OCaml
    - The same trick can also be used in any language that supports lambda abstractions!
    - That includes C++, JavaScript, Python, ...
- Write a top-level print function that consumes the set of solutions one-at-a-time
  - prints the first solution
  - based on user input, either terminates or continues in the print/user-input loop.
Lazy Evaluation in OCaml

- **Lazy evaluation**: suspend actual parameter evaluation until needed
  - The expression is stored as a closure that encapsulates the binding of local variables
- **Lambda definitions** already require this ability
  - The body of the function is an expression that needs to be represented as a closure
- **Idea**: Use lambda definition $f_e$ to represent $e$ needing lazy evaluation
  
  $\text{fun } f_e() \rightarrow e$

  - **Note**: $f_e$ takes an empty argument (technically, a zero-tuple, aka unit in OCaml)
  - Evaluation of $e$ is suspended, until it is applied to a unit argument

Some types and functions for Lazy Evaluation in OCaml

- A type to represent lazily evaluated expressions
  
  $\text{type } 'a \text{ thunk } = \text{Thunk of (unit } \rightarrow 'a) \mid \text{Val of 'a}$

- A function to force evaluation of thunks:
  
  $\text{let force v = match v with Thunk x } \rightarrow x() \mid \text{Val x } \rightarrow x$

- A variant of list type that is evaluated lazily
  
  $\text{type } 'a \text{ lzlist } = \text{Nil } \mid \text{Cons of 'a } * (\text{'a \text{ lzlist thunk}})$

- To operate on such lazy lists, we need to redefine familiar list operations such as append, map, filter, flatten, etc.
  - But almost no other changes needed to the interpreter!

Example: Redefining map for lzlist

$\text{type } 'a \text{ thunk } = \text{Thunk of (unit } \rightarrow 'a) \mid \text{Val of 'a}$

$\text{let rec lzmap (f: 'a } \rightarrow 'b) (l: 'a \text{ lzlist): 'b \text{ lzlist } =}$

$\text{match l with}$

$\mid \text{Nil } \rightarrow \text{Nil}$

$\mid \text{Cons(l1, ls) } \rightarrow$

$\text{Cons((f l1), Thunk(\text{fun } () \rightarrow \text{map f (force ls))})}$
A Backtracking Prolog interpreter

```prolog
let rec call: (prog: clause list) (env: env) (goal: goal): env |\] list =
let paths = (map (find_path goal env) prog) in
let viable_paths = filter (fun (_, (bp, _)) -> bp > 0) paths
in exec_paths prog viable_paths

and exec_paths prog paths = match paths with
  | [] -> Nil
  | p :: ps -> (\append (exec_path prog p) (Thunk(fun () -> (exec_paths prog ps))))

and exec_path: program -> path -> lzenv list =
fun prog (glist, env) = match glist with
  | [] -> Cons(env, Val(Nil))
  | goal :: goals ->
    let envs = call prog env goal in
    let newpaths = map (fun e -> (goals, e)) envs
    in (lflatten (map (exec_path prog) newpaths))
```

Controlling Search

- **If-then-else**: Written as (c -> t ; e) where c, t, e are conjunction of atoms.
  Example:
  ```prolog
gen(N, L) :-
  (N = 0
   -> L = []
   ; M is N-1, gen(M, K), L = [N|R]).
```

Controlling Search (Contd.)

- **Pruning**: Proof search can be pruned using “!” (cut).
  - Cut throws away other choices when more than one clause is applicable.
  - **Use with care**: Prolog's proof process may be hard to understand, and cuts may make the program difficult to comprehend!

  member(X, [X |\]L].
  member(X, [Y|Ys]) :-
    member(X, Ys).

  Finds elements of a list.
  Given X and L, member(X, L) determines whether X is in L or not.
  Given L alone, member(X, L) binds X to elements of L (one by one, when backtracking).

  member(X, [X |\]L] :: L.
  member(X, [Y|Ys]) :-
    member(X, Ys).

  Finds whether or not an element is in a list.
  Given X and L, member(X, L) determines whether X is in L or not.
  Given L alone, member(X, L) binds X to the first element of L.
### Change for a dollar

```
change([H,Q,D,N,P]) :-
    member(H,[0,1,2]), /*Half-dollars*/
    member(Q,[0,1,2,3,4]), /*quarters*/
    member(D,[0,1,2,3,4,5,6,7,8,9,10]), /* dimes */
    member(N,[0,1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15,16,17,18,19,20]), /*nickels*/

    S is 50*H+25*Q+10*D+5*N,
    S=<100,
    P is 100-S.
```

### Permutation

```
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).

perm([],[]).
perm([X|Y],Z) :-perm(Y,W), takeout(X,Z,W).
```

### Tree Isomorphism

```
isomorphic(void, void).
isomorphic(tree(Node, Left1, Right1),
           tree(Node, Left2, Right2)) :-
    isomorphic(Left1, Left2),
    isomorphic(Right1, Right2).
isomorphic(tree(Node, Left1, Right1),
           tree(Node, Left2, Right2)) :-
    isomorphic(Left1, Right2),
    isomorphic(Right1, Left2).
```
Checking/Generating Subtrees

```
subtree(Tree1, Tree2) :-
    isomorphic(Tree1, Tree2).
subtree(Tree1, tree(Node, Left, Right)) :-
    subtree(Tree1, Left); subtree(Tree1, Right).
```

N-Queens

```
solve(P) :-
    perm([1,2,3,4,5,6,7,8],P),
    combine([1,2,3,4,5,6,7,8],P,S,D),
    all_diff(S), all_diff(D).

combine([X1|X], [Y1|Y], [S1|S], [D1|D]) :-
    S1 is X1+Y1, D1 is X1-Y1,
    combine(X,Y,S,D).
combine([],[],[],[]).
all_diff([X|Y]) :- \+member(X,Y), all_diff(Y).
all_diff([X]).
```

Merge Sort

```
merge_sort([], []).
merge_sort([X], [X]).
merge_sort(List, SortedList) :-
    split(List, First, Second),
    merge_sort(First, SortedFirst),
    merge_sort(Second, SortedSecond),
    merge(SortedFirst, SortedSecond, SortedList).

split([], [], []).
split([X], [X], []).
split([X1,X2|Xs], [X1|Ys], [X2|Zs]) :- split(Xs, Ys, Zs).
```
merge([], X, X).
merge(X, [], X).
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X=<Y,
    merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y,
    merge([X|Xs], Ys, Zs).