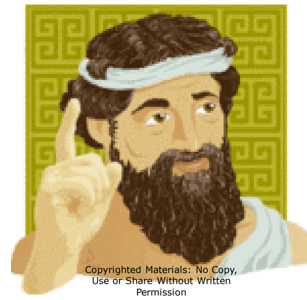
Think Relational



Logic for Contextual Association

- Logic provide us with a universal structure of information governed by contextual associations. With such associations we can capture and represent rules and natural structures that are useful for building smarter, more human-like systems. According to Joe's Theory of Everything (JTE), relations between data items are the fabric of science. "For our course purposes, it is enough to say that relations among facts much or all of business, science, and knowledge can be described as links between facts.
- **Facts represent relations not functions.**

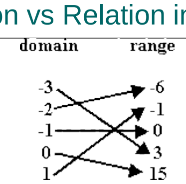
What is a Relation and how it differs from Function?

- A "relation" is just a relationship between sets of information. **Think of all the people in one of your classes, and think of their heights. The pairing of names and heights is a relation.**
- A function is a "well-behaved" relation. This means given a starting point, we know exactly where to go; given an *x*, we get only and exactly one *y*.

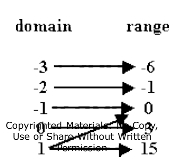
Relations have Multiple Behaviors?

- Let's return to our relation of your classmates and their heights, and let's suppose that the domain is the set of everybody's heights. Let's suppose that there's a pizza-delivery guy waiting in the hallway. And all the delivery guy knows is that the pizza is for the student in your classroom who is five-foot-five. Now let the guy in. Who does he go to? What if nobody is five-foot-five? What if there are six people in the room that are five-five?

Function vs Relation in Math



Function



Relation

How to Think Relational?

- It is to represent business rules on data of the problem defined. Business rules specify conditions and relationships that must always be true, or must always be false.
- Business rules represent constraints (.i.e conditions or relations)

Relational Programming = Constraints Satisfaction Problem (CSP)

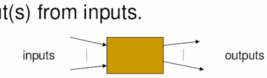
- Constraint programming is a programming paradigm wherein relations between variables are stated in the form of constraints. Constraints differ from the common primitives of imperative programming languages in that they do not specify a step or sequence of steps to execute, but rather the properties of a solution to be found. **This makes constraint programming a form of declarative programming.**

Relational Programming

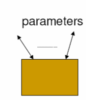
- A relation is also called a predicate pred(a,b,c)
- Tuple is an element in a relation
- Logic program is a specification of a relation (contrast to functional programming)
  brother (sam, bill)
  brother (sam, bob)
  Brother is not a function, since it maps "sam" to two different range elements
  Brother is a relation
- Relations are n-ary, not just binary
  family(jane,sam,[ann,tim,sean])
- Relations are multi-directional
  - It works backward, forward and hybrid

Relation as Bidirectional Functions

- Functions are *directional* and computes output(s) from inputs.



- Relations are *bidirectional* and used to relate a tuple of parameters.



Examples of Relations

- Parent-child relations e.g. `parent(X,Y)`
- Classification e.g. `male(X)` or `female(Y)`
- Operations e.g. `append(Xs,Ys,Zs)`
- Databases: `employee(Name,… )` relational tables?
- Geometry problems: how are sides of rectangles related, e.g. `rect(X,Y,X,Y)`
- Each function is a special case of relation.

Relational Programming

- Logic programs define relations and allow you to express patterns to extract various tuples from the relations
- Infinite relations cannot be defined by rote… need rules
  - *(A,B) are related if B is A*A*
  - *(B,H,A) are related if A is ½ B*H*

or… gen all tuples like this (B,H,B*H*0.5)

Logic program uses Horn clauses for explicit definition (facts) and for rules

Inference Making or Resolution

- The derivation of new statements from given knowledge base is called

  Resolution
- The logic programming system combines existing statements to find new statements…
- *Three Basic Techniques are used in this direction:*
  - *Pattern Matching*
  - *Unification*
  - *Backtracking*

$x + y = 9$ $2x + 4y = 24$

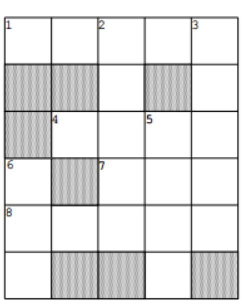
x	0	1	2	3	4	5	6	7	8	9
y	0	1	2	3	4	5	6	7	8	9

$x + y = 9$ $2x + 4y = 24$

x	0	1	2	3	4	5	6	7	8	9
y	0	1	2	3	4	5	6	7	8	9

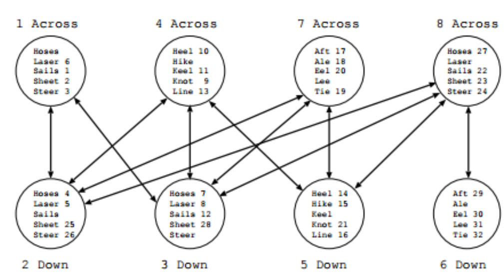
$x + y = 9$ $2x + 4y = 24$

x	0	1	2	3	4	5	6	7	8	9
y	0	1	2	3	4	5	6	7	8	9



Word List

Aft
Ale
Eel
Heel
Hike
Hoses
Keel
Knot
Laser
Lee
Line
Sails
Sheet
Steer
Tie



The Relational Model in OZ

```
⟨S⟩::= skip                               empty statement
     | ⟨S1⟩ ⟨S2⟩                          statement sequence
     | ...
     | choice ⟨S1⟩ [] … [] ⟨Sn⟩ end       choice
     | fail                               failure
```

Choice allows alternatives to be explored, while failure indicates no answer at that branch.

Choice-Fail

- The relational computation model extends the declarative model with two new statements, choice and fail:
- The choice statement groups together a set of alternative statements. Executing a **choice** statement provisionally picks one of these alternatives. If the alternative is found to be wrong later on, then another one is picked.
- The fail statement indicates that the current alternative is wrong. A **fail** is executed implicitly when trying to bind two incompatible values,

Relational Program uses Backtracking and generates a Search Tree

- A relational program is executed sequentially.
- The **choice** statements are executed in the order that they are encountered during execution.
- When a **choice** is first executed, its first alternative is picked. When a **fail** is executed, execution "backs up" to the most recent **choice** statement, which picks its next alternative. If there are none, then the next most recent **choice** picks another alternative, and so forth. Each **choice** statement picks alternatives in order from left to right.
- This execution strategy can be illustrated with a tree called the *search tree*. Each node in the search tree corresponds to a **choice** statement and each subtree corresponds to one of the alternatives.

Execution of Relational Program

- A relational program is interesting because it can potentially execute in many different ways, depending on the choices it makes. We would like to control which choices are made and when they are made. For example, we would like to specify the search strategy: depth-first search, breadth-first search, or some other strategy. We would like to specify how many solutions are calculated: just one solution, all solutions right away, or new solutions on demand. Briefly, we would like the same relational program to be executed in many different ways.
- There are two ways to perform search in OZ:
  - **1. Basic Search Engines**
  - **2. General Search Engines**

Basic Search Engines

- Using built-in Search.base functions:
  1. Search.base.one
  2. Search.base.all
- **base.one**
  - {Search.base.one +*ScriptP*?*Xs*}
  - returns a singleton list containing the first solution of the script +*ScriptP* (a unary procedure) obtained by depth-first search. If no solution exists, nil is returned.
- **base.all**
  - {Search.base.all +*ScriptP*?*Xs*}
  - returns the list of all solutions of the script +*ScriptP* (a unary procedure) obtained by depth-first serach.

Example

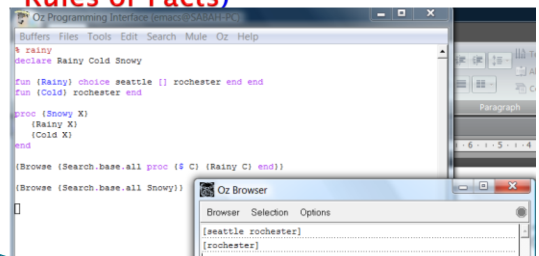
```
{Search.base.one proc {$ X}
          choice
             choice X=ape
                [] X=bear
             end
             [] X=cat
          end
 end}
```

- returns the list [ape].

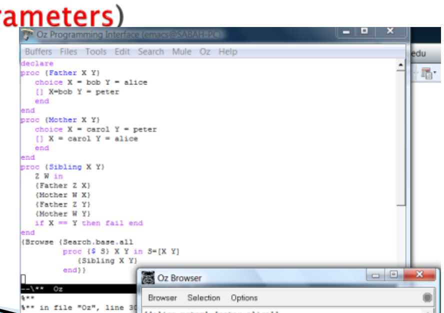
Example

- {Search.base.all proc {$ X}
  choice
  choice X=ape [] X=bear end
  [] X=cat
  end
  end}
- returns the list [ape bear cat].

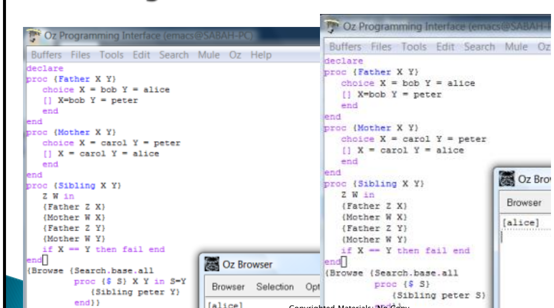
Searching for Facts and Rules (Simple Query–Direct Calls to Rules or Facts)



Example 2: Family Problem: (General Query–Thinking a bout Parameters)



Family Problem: General Query–Thinking about Parameters…

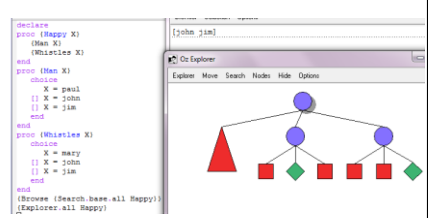


From English to OZ KR?

- Suppose you know the following:
- Every man who whistles is happy.
- John is a man.
- John whistles.
- Mary whistles.
- Paul is a man.
- Jim is a man and he whistles.
- Write a program in Oz that is able to infer that **John is happy and Jim is happy**

KR + Inference in OZ

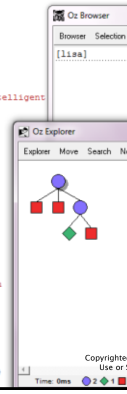
- declare
- proc {Happy X}
- {Man X}
- {Whistles X}
- end
- proc {Man X}
- choice
- X = paul
- [] X = john
- [] X = jim
- end
- end
- proc {Whistles X}
- choice
- X = mary
- [] X = john
- [] X = jim
- end
- end
- {Browse {Search.base.all Happy}}
- {Explorer.all Happy}



Example 2: From English to OZ KR

- Suppose you know the following:
- Every human who is intelligent and rich is happy. Happy people are kind. If you can count, you are intelligent. John is rich. Paul is intelligent. Lisa is rich, and she can count. Paul and John and Lisa are human.
- Write a program in Oz that is able to infer that Lisa is kind. Use the Oz Explorer to investigate the search tree.


```
declare
% Every human who is intelligent and rich is happy.
proc {Happy X}
   {Human X}
   {Rich X}
   {Intelligent X}
end
% Happy people are kind
proc {Kind X}
   {Happy X}
end
% If you can count, you are intelligent
% John is intelligent
proc {Intelligent X}
   choice
      {CanCount X}
   [] X = john
   end
end
% Lisa is rich
proc {Rich X}
   X = lisa
end
% Lisa can count
proc {CanCount X}
   X = lisa
end

% Paul, John and Lisa are human
proc {Human X}
   choice
      X = paul
   [] X = john
   [] X = lisa
   end
end
{Browse {Search.base.all Kind}}
{Explorer.all Kind}
```



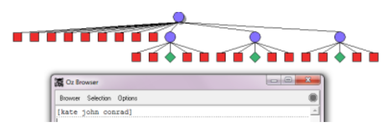
Given the following relationships, define and test the following relations:
X is the father of Y
X is the grandmother of Y
X is the sister of Y
X is the uncle of Y
X is the cousin of Y

X is an ancestor of Y

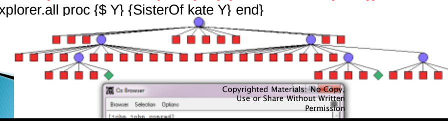
```
proc {Parent X Y}
   choice
      X = katherine Y = bertrand
   [] X = katherine Y = frank
   [] X = katherine Y = rachel
   [] X = dora Y = kate
   [] X = dora Y = john
   [] X = anne Y = conrad
   [] X = amberley Y = bertrand
   [] X = amberley Y = frank
   [] X = amberley Y = rachel
   [] X = bertrand Y = kate
   [] X = bertrand Y = john
   [] X = bertrand Y = conrad
   end
end

proc {Female X}
   choice
      X = katherine
   [] X = rachel
   [] X = dora
   [] X = anne
   [] X = kate
   end
end

proc {Male X}
   choice
      X = amberley
   [] X = frank
   [] X = bertrand
   [] X = conrad
   [] X = john
   end
end

proc {NotEq X Y}
   if X == Y then fail
   else skip
   end
end
```



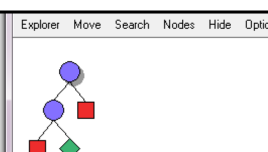
```
proc {FatherOf X Y}
   {Parent X Y}
   {Male X}
end
proc {SisterOf X Y}
   Z in
   {Parent Z X}
   {Parent Z Y}
   {NotEq X Y}
   {Female X}
End
```



% Who is Bertrand the father of?
{Browse {Search.base.all proc {$ Y} {FatherOf bertrand Y} end}}
{Explorer.all proc {$ Y} {FatherOf bertrand Y} end}
% Who is kate the sister of?
{Browse {Search.base.all proc {$ Y} {SisterOf kate Y} end}}
{Explorer.all proc {$ Y} {SisterOf kate Y} end}




```
declare

proc {S P0 P}
   P1 in
   {NP P0 P1}
   {VP P1 P}
end

proc {NP P0 P}
   choice
      P0 = john|P
   [] P0 = mary|P
   end
end

proc {VP P0 P}
   choice
      P0 = whistles|P
   [] P0 = walks|P
   end
end

{Explorer.all proc {$ _} {S [john walks] nil} end}
% {Explorer.all proc {$ _} {S [whistles walks] nil} end}
```



Given is the following fragment of English:

john walks
john whistles
mary walks
mary whistles

captured by the following grammar

```
S --> NP VP
NP --> john | mary
VP --> walks | whistles
```

Write a program in Oz capable of recognizing the above sentences solve it.

Given is the following fragment of English:

john walks
mary whistles
mary likes john
mary eats a banana
every man likes mary
…

captured by the following grammar

```
S --> NP VP
NP --> PN
NP --> DET N
VP --> IV
VP --> TV NP

PN --> john | mary
DET --> a | every
N --> banana | man
IV --> walks | whistles
TV --> likes | eats
```

Exercise: Write a program in Oz which is able to recognize them. Use the Oz Explorer to investigate the search tree.

General Search Engine: The Solve

- We provide encapsulated search by adding one function, Solve, to the computation model. This is an external search engine. Load them from your D2L.
- The call {Solve F} is given a zero-argument function F (or equivalently, a one-argument procedure) that returns a solution to a relational program. The call returns a lazy list of all solutions, ordered according to a depth-first search strategy. For example, the call:
- L={Solve fun {$} choice 1 [] 2 [] 3 end end}
- returns the lazy list [1 2 3]. Because Solve is lazy, it only calculates the solutions that are needed. Solve is compositional, i.e., it can be nested: the function F can contain calls to Solve. Using Solve as a basic operation, we can define both one-solution and all-solutions search. To get one-solution search, we look at just the first element of the list and never look at the rest:
- The Solve function includes SolveOne and SolveAll.
- You need to \insert 'Solve.oz at your program

Using the Solve: SolveOne, SolveAll

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'Solve.oz'

declare
fun {Digit}
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {SolveOne Digit}}
{Browse {SolveAll Digit}}
```

Oz Browser

Browser Selection Options

```
[0]
[0 1 2 3 4 5 6 7 8 9]
```


How about SolveSome? You can programm it.

Oz Programming Interface (emacs@SABAH-PC)

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'Solve.oz'

declare
fun {Digit}
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {List.take {Solve Digit} 5}}
```

Oz Browser

Browser Selection Options

```
[0 1 2 3 4]
```


Using to Solve to satisf more complicated constraints?

I want list of four numbers where N1<N2<N3<N4 where Ni is a Digit

Buffers Files Tools Edit Search Mule Oz Help

```
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
%{Browse {SolveOne Digit}}
%{Browse {SolveAll Digit}}

declare
fun {DigitSeq}
   local L A B
   in
      L = [{Digit} {Digit} {Digit} {Digit}]
      {L.1 < L.2.1} = true
      {L.2.1 < L.2.2.1} = true
      {L.2.2.1 < L.2.2.2.1} = true
      L
   end
end
{Browse {SolveAll DigitSeq}}
{Browse {Length {SolveAll DigitSeq}}}
```

Oz Browser

Browser Selection Options

```
[0 1 2 3]|[0 1 2 4]|[0 1 2 5]|[0 1 2 6]|[0 1 2 7]|[0 1 2 8]
[0 1 2 9]|[0 1 3 4]|[0 1 3 5]|[0 1 3 6]|[0 1 3 7]|[0 1 3 8]
[0 1 3 9]|[0 1 4 5]|[0 1 4 6]|[0 1 4 7]|[0 1 4 8]|[0 1 4 9]
[0 1 5 6]|[0 1 5 7]|[0 1 5 8]|[0 1 5 9]|[0 1 6 7]|[0 1 6 8]
[0 1 6 9]|[0 1 7 8]|[0 1 7 9]|[0 1 8 9]|[0 2 3 4]|[0 2 3 5]
[0 2 3 6]|[0 2 3 7]|[0 2 3 8]|[0 2 3 9]|[0 2 4 5]|[0 2 4 6]
[0 2 4 7]|[0 2 4 8]|[0 2 4 9]|[0 2 5 6]|[0 2 5 7]|[0 2 5 8]
[0 2 5 9]|[0 2 6 7]|[0 2 6 8]|[0 2 6 9]|[0 2 7 8]|[0 2 7 9]
[0 2 8 9]|[0 3 4 5]|,,,
210
```


Using the Explorer

Oz Programming Interface (emacs@SABAH-PC)

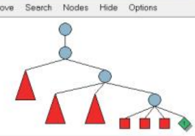
Buffers Files Tools Edit Search Mule Oz Help

```
declare
fun {Digit}
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end

declare
fun {DigitSeq}
   local L A B
   in
      L = [{Digit} {Digit} {Digit} {Digit}]
      {L.1 < L.2.1} = true
      {L.2.1 < L.2.2.1} = true
      {L.2.2.1 < L.2.2.2.1} = true
      L
   end
end
{ExploreOne DigitSeq}
```

```
--\--  Dig5.oz          (Oz)
%**
%** unused local variable B
%**
%** in file "f:/CS4411/Dig5.oz
Declared variables:
```

Oz Explorer

Explorer Move Search Nodes Hide Options



Oz Inspector

Inspector Selection Options

```
1#
[0 1 2 3]
```


Can I add constraints inside the Query? With Implicit Fail

```
\insert 'Solve.oz'
declare

proc {Vowel ?R}
   choice
      R=a
   [] R=e
   [] R=i
   [] R=o
   [] R=u
   end
end

% But can be used in a search with Solve
{Browse {List.take {Solve Vowel} 4}}
```

Browser Sel

```
[a e i o]
```

With Explicit Fail

```
declare
proc {Distinct ?Res}
   V1 V2
in
   {Vowel V1}
   {Vowel V2}
   if V1 == V2 then fail end
   Res=V1#V2
end

{Browse {List.take {Solve Distinct} 7}}
```

Oz Browser

Browser Selection Options

```
[a#e a#i a#o a#u e#a e#i e#o]
```


How about Querying Family?

Buffers Files Tools Edit Search Mule Oz

```
proc {Dad ?Father ?Kid}
   choice
      Father=pop Kid=sarah
   [] Father=pop Kid=john
   [] Father=pop Kid=robert
   [] Father=pop Kid=jill
   end
end
proc {Mom ?Mother ?Kid}
   {Dad _ Kid}
   Mother=mom
end
proc {Male ?Person}
   choice
      Person=pop
   [] Person=robert
   [] Person=john
   end
end
proc {Female ?Person}
   choice
      Person=mom
   [] Person=sarah
   [] Person=jill
   end
end
```

```
{Browse {SolveAll proc {$ Who}
                     {Dad pop Who}
                  end}}

% Who are the children of mom?
{Browse {SolveAll proc {$ Who}
                     {Mom mom Who}
                  end}}

% Who is male?
{Browse {SolveAll proc {$ Who}
                     {Male Who}
                  end}}

% Who is female?
{Browse {SolveAll proc {$ Who}
                     {Female Who}
                  end}}
```

Browser Selection Options

```
[sarah john robert jill]
[sarah john robert jill]
[pop robert john]
[mom sarah jill]
```


Querying Family: Another Example

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'Solve.oz'
declare
proc {Son ?X ?Y}
   {Parent Y X} {Male X}
end

proc {Daughter ?X ?Y}
   {Parent Y X} {Female X}
end

proc {Parent ?Y ?X}
   choice
      {Dad Y X}
   [] {Mom Y X}
   end
end
% What children are sons of what adults?
{Browse {SolveAll proc {$ Ans}
                     Child#Adult=Ans
                  in
                     {Son Child Adult}
                  end}}
```

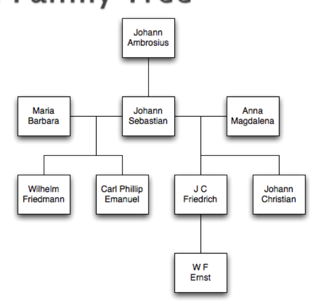
Oz Browser

Browser Selection Options

```
[john#pop robert#pop john#mom robert#mom]
```


DIY Exercise: Family Tree

Represent and query this family tree in OZ.
Represent at least three different family relationships and query each one forward and backward.



Relational Programming

- http://www.idi.ntnu.no/emner/tdt4165/handouts/
- The relational model of computation covered in the last lecture is an extension of the declarative model of computation with:
- ▲ non-deterministic **choice** statements,
- ▲ **failure** statements.
- This means an ability to describe and a process to infer or propagate solution(s) via (unification-matching-navigating through alternatives (choice/backtracking)

Concrete Represention of Relations in OZ?

•Relations can be represented using Functions or Procedures
•Parameters in Relations can be either Input or Output
•Relations works Backwards or Forward: Example:
- Append Relation may work as Concat or Split?

Is Append Fully Relational?

```
declare
proc {Append List1 List2 List3}
try List1 = nil List2 = List3
catch _ then Head Tail1 Tail3 in
    List1 = Head|Tail1
    List3 = Head|Tail3
    {Append Tail1 List2 Tail3} end end

{Browse {Append [1] [2 3] $}} % prints [1 2 3]
{Browse {Append [1] $ [1 2 3]}} % prints [2 3]

{Browse {Append $ [2 3] [1 2 3]}} % fails at nil = _|_
```

Browser

```
[1 2 3]
[2 3]
```


Better version? Using or?

```
declare
proc {Append List1 List2 List3}
   or List1 = nil
      List2 = List3
   [] Head Tail1 Tail3 in
      List1 = Head|Tail1
      List3 = Head|Tail3
      {Append Tail1 List2 Tail3} end end
{Browse {Append [1] [2 3] $}}
{Browse {Append [1] $ [1 2 3]}}
{Browse {Append $ [2 3] [1 2 3]}}
```

Browser

```
[1 2 3]
[2 3]
[1]
```


Fully Relational–Using Choice

```
declare
\insert 'solve.oz'
proc {Append List1 List2 List3}
   choice
      List1 = nil
      List2 = List3
   [] Head Tail1 Tail3 in
      List1 = Head|Tail1
      List3 = Head|Tail3
      {Append Tail1 List2 Tail3} end end
{Browse
 {SolveAll
  fun {$}
     {Append $ [2 3] [1 2 3]} end}}

{Browse
 {SolveAll
  fun {$}
     List1 List2 in
     {Append List1 List2 [1 2 3]}
     solution(List1 List2) end}}
```

Oz Browser

Browser Selection Options

```
[[1]]
[solution(nil [1 2 3]) solution([1] [2 3])
 solution([1 2] [3]) solution([1 2 3] nil)]
```


More Concrete way to represent Append as a Relation

```
declare
proc {RAppend ?A ?B ?C}
   choice
      A=nil B=C
   [] local As Cs X in
         A=X|As
         C=X|Cs
         {RAppend As B Cs}
      end
   end
end
% Forward use
{Browse {SolveFirst
         proc {$ S} Z=S in {RAppend [3 4] [5 6] Z} end}}
% Backward use
{Browse {SolveFirst
         proc {$ S} X=S in {RAppend X [2 3] [1 2 3]} end}}
{Browse {SolveFirst
         proc {$ S} Y=S in {RAppend [3 4] Y [3 4 5 6]} end}}
```

Oz Browser

Browser Selection Options

```
[3 4 5 6]
[1]
[5 6]
```


Relational Memeber

Oz Programming Interface (emacs@

Buffers Files Tools Edit Search Mule O

```
declare
proc {RMember ?X ?Ls}
   local E Es in
      Ls=E|Es
      choice
         X=E
      [] {RMember X Es}
      end
   end
end

proc {RMember2 ?X ?Ls}
   choice
      Ls=X|_
   [] local Es in
         Ls=_|Es
         {RMember2 X Es}
      end
   end
end

{Browse {SolveAll proc {$ V} {RMember 3 4|3|2|nil} V=ok end}}
{Browse {SolveAll proc {$ E} {RMember E 4|3|2|nil} end}}
```

Browser Selection Options

```
[ok]
[4 3 2]
```


Relational Add to End of List

Buffers Files Tools Edit Search Mule Oz Help

```
declare
proc {AddToEnd ?L ?X ?R}
   choice
      L=nil R=X|nil
   [] local E Es Res in
         L=E|Es
         {AddToEnd Es X Res}
         R=E|Res
      end
   end
end
fun {AddToEnd2 ?L ?X}
   choice
      L=nil X|nil
   [] local E Es in
         L=E|Es
         E|{AddToEnd2 Es X}
      end
   end
end
{Browse {SolveFirst fun {$} {AddToEnd [a b c] d $} end}}
{Browse {SolveFirst fun {$} {AddToEnd [a b c] $ [a b c d]} end}}
{Browse {SolveFirst fun {$} {AddToEnd $ d [a b c d]} end}}
{Browse {SolveFirst fun {$} {AddToEnd2 [a b c] d $} end}}
{Browse {SolveFirst fun {$} {AddToEnd2 [a b c] $ [a b c d]} end}}
{Browse {SolveFirst fun {$} {AddToEnd2 $ d [a b c d]} end}}
```

Oz Browser

Browser Selection Options

```
[a b c d]
d
[a b c]
[a b c d]
d
[a b c]
```

Relational Reverse

```
\insert 'SolveFirst.oz'
\insert 'SolveOne.oz'
\insert 'SolveAll.oz'
\insert 'Solve.oz'
\insert 'AddToEnd.oz'
declare
proc {Reverse L R}
   choice
      L=nil R=nil
   [] local E Es R2 in
         L=E|Es {Reverse Es R2} {AddToEnd R2 E R}
      end
   end
end

{Browse {SolveAll proc {$ V} {Reverse [1 2 3 4] V} end}}
{Browse {SolveAll proc {$ V} {Reverse [2 3 4] V} end}}
```

Oz Browser

Browser Selection Options

```
[[4 3 2 1]]
[[4 3 2]]
```


Relational Insert

Oz Programming Interface (emacs@SABAH-PC)

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'Solve.oz
\insert 'SolveAll.oz

declare
proc {Insert X InpL OutL}
   choice
      A | B = OutL in
      A = X
      B = InpL
   []
      A | B = InpL
      C | D = OutL in
      A = C
      {Insert X B D}
   end
end

{Browse {SolveAll proc {$ S} {Insert y [a b c] S} end}}

{Browse {SolveAll proc {$ S} {Insert S [a b c] [a b x c]} end}}

{Browse {SolveAll proc {$ Y} S#T=Y in {Insert S T [a b x c]} end}}
```

Oz Browser

Browser Selection Options

```
[[y a b c] [a y b c] [a b y c] [a b c y]]
[x]
[a#[b x c] b#[a x c] x#[a b c] c#[a b x]]
```


Relational Delete

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'SolveAll.oz
declare
proc {Delete X InpL ?OutL}
   choice
      local
         A | As = InpL in
         X = A
         As = OutL
      end
   []
      local
         A | As = InpL
         B | Bs = OutL in
         B = A
         {Delete X As Bs}
         OutL = A | Bs
      end
   end
end
{Browse {SolveAll proc {$ S} {Delete 2 S [3 4 2]} end}}
{Browse {SolveAll proc {$ S} {Delete 2 [3 4 2 2] S} end}}
{Browse {SolveAll proc {$ S} {Delete S [3 5 4 2] [3 4 2]} end}}
{Browse {SolveAll proc {$ S} {Delete 6 [3 5 4 2] S} end}}
{Browse {SolveAll proc {$ S} {Delete S [3 5 4 2] [3 4 2]} end}}
```

Oz Browser

Browser Selection Options

```
[[2 3 4 2] [3 2 4 2] [3 4 2 2] [3 4 2 2]]
[[3 4 2] [3 4 2]]
[5]
nil
[5]
```


How to Query a Relation?

Two ways: (1) using a function that uses solve (2) directly using solve

Buffers Files Tools Edit Search Mule Oz Help

```
proc {Capitals ?State ?City}
   choice
      State=florida City=tallahassee
   [] State=iowa City=desMoines
   [] State=kansas City=topeka
   [] State=virginia City=richmond
   [] State=michigan City=lansing
   [] State=newYork City=albany
   [] State=california City=sacremento
   end
end
fun {QueryCapitals State City}
   Ans = {SolveOne
          proc {$ Ok}
             {Capitals State City}
             Ok=yes
          end}
in if Ans \= nil then yes else no end
end
{Browse {QueryCapitals florida tallahassee}}
{Browse {QueryCapitals newYork albany}}
{Browse {QueryCapitals georgia atlanta}}
% queries with unbound variables: What is the capital of iowa?
{Browse {SolveFirst proc {$ What} {Capitals iowa What} end}}
% What state is albany the capital of?
{Browse {SolveFirst proc {$ What} {Capitals What albany} end}}
```

Oz Browser

Browser Selection Options

```
yes
yes
no
desMoines
newYork
```


Example on Querying a relation in using direct solve: Agriculture Products

Buffers Files Tools Edit Search Mule Oz Help

```
declare
proc {Agriculture ?State ?Product}
   choice
      State=florida Product=oranges
   [] State=iowa Product=pork
   [] State=kansas Product=sunflowers
   [] State=virginia Product=ham
   [] State=michigan Product=cherries
   [] State=newYork Product=dairy
   [] State=california Product=wine
   end
end
% What is the capital and chief agricultural product of florida?
{Browse  {SolveFirst proc {$ Ans}
                        Town#Farmed = Ans
                     in
                        {Capitals florida Town} %% and
                        {Agriculture florida Farmed}
                     end}}
% What is the capital and chief agricultural product of iowa?
{Browse {SolveFirst proc {$ Ans}
                       Town#Farmed=Ans
                    in
                       {Capitals iowa Town}
                       {Agriculture iowa Farmed}
                    end}}
```

Oz Browser

Browser Selection Options

```
tallahassee#oranges
desMoines#pork
```


More Complex Relational Thinking: Coin Change?

- Given a set of coins and amount, Write a Relational OZ program to find out how many ways we can make the change of the amount using the coins given.
- Example (Computing change)
- nominals: nickel (5 cents), dime (10 cents), quarter (25 cents)
  - amount: 25 cents
  - change:
    - 5 nickels
    - 3 nickels and 1 dime
    - 1 nickel and 2 dimes
    - 1 quarter

The Change Relation

```
fun {Change Amount Purse}
   choice Amount = 0 nil
   [] Amount > 0 = true
      Coin = {Purse}
      Rest = {Change Amount-Coin Purse} in
      Coin|Rest end end
```

The version of Change makes two choices:
- the amount is positive, and given a coin it is possible to change the rest; the coin is then returned together with the rest of the change;
- the amount is zero; the change is no coins.

If the amount is negative, the branch of the search simply fails (change impossible).

Testing the Coins Change

```
declare
\insert 'solve.oz'
declare
Nickel = 5
Dime = 10
Quarter = 25
Nominals = [Nickel Dime]
fun {Pick List}
   Head|Tail = List in
   choice Head [] {Pick Tail} end end
fun {Sum List}
   {FoldL List Number.'+' 0} end
fun {Purse}
   {Pick Nominals} end

declare
\insert 'change-alternative.oz'
{Browse
 {SolveAll
  fun {$}
     {Change Quarter Purse} end}}
```

Notice:
the results list is redundant; some of its elements represent the same change, they just differ in the order of coin nominals.

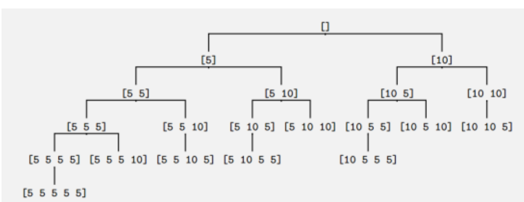
How to eliminate this redundancy??????

Oz Browser

Browser Selection Options

```
[[5 5 5 5 5] [5 5 5 10] [5 5 10 5] [5 10 5 5] [5 10 10]
 [10 5 5 5] [10 5 10] [10 10 5]]
```


The Search Tree



The search is redundant, because after having chosen the second nominal, we can choose the first one again.
- If many more nominals were available, the search space would be exponentially larger

We can prevent the redundance by demanding that a nominal that has been skipped over in a search branch can never be chosen again

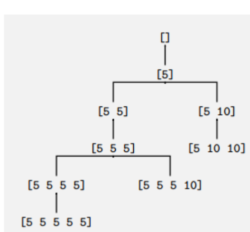
Change without Redundancy

```
fun {Change Amount Nominals}
   choice Amount = 0 nil
   [] Amount > 0 = true
      Coin|Coins = Nominals in
      choice
     Rest = {Change Amount-Coin Nominals} in
     Coin|Rest
       [] {Change Amount Coins} end end end
```

The search is no longer redundant, because after having chosen the second nominal, we cannot choose the first one any more.

Testing the Change without Redundancy

```
declare
\insert 'solve.oz'
declare
Nickel = 5
Dime = 10
Quarter = 25
Nominals = [Nickel Dime]
fun {Pick List}
   Head|Tail = List in
   choice Head [] {Pick Tail} end end
fun {Sum List}
   {FoldL List Number.'+' 0} end
fun {Purse}
   {Pick Nominals} end

declare
\insert 'change-nonredundant.oz'
{Browse
 {SolveAll
  fun {$} {Change Quarter Nominals} end}}
```



Oz Browser

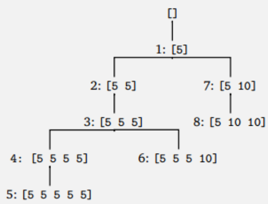
Browser Selection Options

```
[[5 5 5 5 5] [5 5 5 10] [5 10 10]]
```


Change with Different Searching Strategies

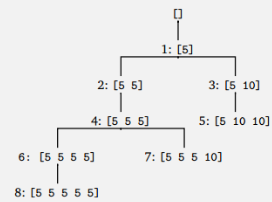
- The Solve we have used so far implements the depth-first search strategy.
- Whenever a node does not correspond to a final solutions, further choices are made, and the node's children are examined before the node's siblings.
- The relational model of computation in Oz uses depth-first search, but it's just because Solve is implemented this way.
- It is possible to reimplement Solve so that it uses, e.g., breadth-first
- search, or even allows the user to choose the strategy when Solve is called.

Example (Non-redundant change, depth-first search)



- Complete solutions are found at nodes 5, 6, and 8, in this order.

Example (Non-redundant change, breadth-first search)



- Complete solutions are found at nodes 5, 7, and 8, in this order.

Changing Solve.oz to include Strategies

```
fun {SolveAll WhichFirst Script}
   {TouchAll {Solve WhichFirst Script}} end
fun {TouchAll List}
   case List of
      nil then skip
   [] _|List then {TouchAll List _} end
   List end
fun {Solve WhichFirst Script}
   fun lazy {SolveSpaces CSpaces}
      case CSpaces of
     nil then nil
      [] CSpace|CSpaces then
     case CSpace
     of commitTo(CSpace I) then
        Clone = {Space.clone CSpace} in
        {Space.commit Clone I}
        {SolveSpaces Clone|CSpaces}
     else
        {SolveSpace {Space.ask CSpace} CSpace CSpaces} end end end
   fun {SolveSpace SpaceState CSpace CSpaces}
      case SpaceState
      of failed then
     {SolveSpaces CSpaces}
      [] succeeded then
     {Space.merge CSpace}|{SolveSpaces CSpaces}
      [] alternatives(N) then
     {SolveSpaces {NewSpaceList {Choices CSpace N} CSpaces}} end end
   fun {Choices CSpace N}
      {List.mapInd
       {List.make N}
       fun {$ I _} commitTo(CSpace I) end} end
   fun {NewSpaceList Choices CSpaces}
      if WhichFirst == depth then {Append Choices CSpaces}
      elseif WhichFirst == breadth then {Append CSpaces Choices}
      else {Raise traversalSpecificationError(WhichFirst)} nil end end in
   {SolveSpaces [{Space.new Script}]} end
```


Testing Change with Strategies

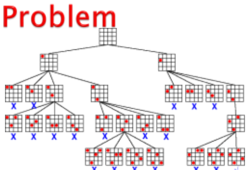
```
declare
\insert 'solve.oz'
declare
Nickel = 5
Dime = 10
Quarter = 25
Nominals = [Nickel Dime]
fun {Pick List}
   Head|Tail = List in
   choice Head [] {Pick Tail} end end
fun {Sum List}
   {FoldL List Number.'+' 0} end
fun {Purse}
   {Pick Nominals} end

declare
\insert 'solve-strategy.oz'
{Browse
 {SolveAll depth
  fun {$} {Change Quarter Nominals} end}}
{Browse
 {SolveAll breadth
  fun {$} {Change Quarter Nominals} end}}
```

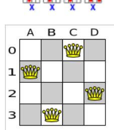
Oz Browser

Browser Selection Options

```
[[5 5 5 5 5] [5 5 5 10] [5 10 10]]
[[5 10 10] [5 5 5 5 5] [5 5 5 10]]
```


Another Example: 4 Queens Problem






```
declare
fun {Queens N}
   proc {PlaceQueens N ?Cs ?Us ?Ds}
      if N>0 then
         Ds2 Us2 in
         Us = _ | Us2
         Ds2= _ | Ds
         {PlaceQueens N-1 Cs Us2 Ds2}
         {PlaceQueen N Cs Us Ds}
      else skip end
   end
   proc {PlaceQueen N ?Cs ?Us ?Ds}
      choice
         Cs=N|_
         Us=N|_
         Ds=N|_
      []
         _|Cs2=Cs
         _|Us2=Us
         _|Ds2=Ds in
         {PlaceQueen N Cs2 Us2 Ds2}
      end
   end
   Qs={MakeList N}
in
   {PlaceQueens N Qs _ _}
   Qs
end
{Browse {Solve.solveOne fun {$} {Queens 4} end}}
{Browse {Solve.solveOne fun {$} {Queens 5} end}}
{Browse {Solve.solveOne fun {$} {Queens 6} end}}
{Browse {Solve.solveOne fun {$} {Queens 7} end}}
{Browse {Solve.solveOne fun {$} {Queens 8} end}}
```


Solution

Oz Programming Interface (emacs@SABAH-PC)

Buffers Files Tools Edit Search Mule Oz Help

```
\insert 'Solve.oz'
declare
fun {Queens N}
   proc {PlaceQueens N ?Cs ?Us ?Ds}
      if N>0 then
         Ds2 Us2 in
         Us = _ | Us2
         Ds2= _ | Ds
         {PlaceQueens N-1 Cs Us2 Ds2}
         {PlaceQueen N Cs Us Ds}
      else skip end
   end
   proc {PlaceQueen N ?Cs ?Us ?Ds}
      choice
         Cs=N|_
         Us=N|_
         Ds=N|_
      []
         _|Cs2=Cs
         _|Us2=Us
         _|Ds2=Ds in
         {PlaceQueen N Cs2 Us2 Ds2}
      end
   end
```

Oz Browser

Browser Selection Op

```
[[3 1 4 2]]
[[1 4 2 5 3]]
[[4 1 5 2 6 3]]
[[1 5 2 6 3 7 4]]
[[1 7 5 8 2 4 6 3]]
```