Prolog

Need to learn

Prolog database of facts and rules

Inference engine of Prolog

- Backtrack search (Depth-First Search)

Logical variables

- Different from variables in most languages
- Can be assigned with any data type

Unification

- Built-in pattern matcher

Prolog Program

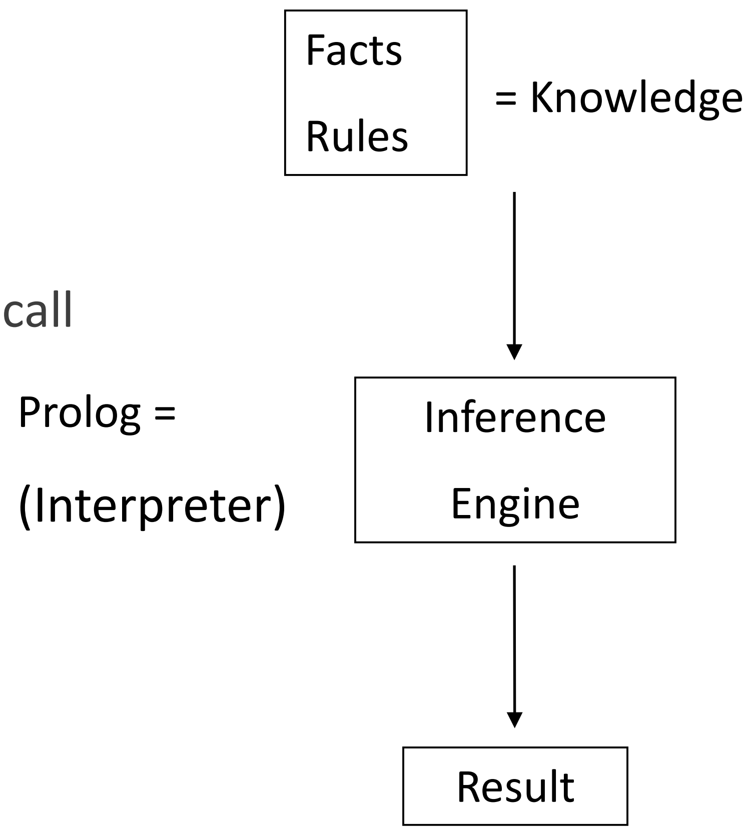
Database

- A set of clauses (sentences)
  - Facts or rules
  - All are predicates
    - Small units, similar to subroutine call
    - Return truth values

```
person(socrates).
person(plato).
person(aristotle).
age(peter,15)
mortal(X) :- person(X).
```

Facts: person(socrates). person(plato). person(aristotle). age(peter,15)

Rule: mortal(X) :- person(X).



Prolog

Prolog Listener
- Listen to queries from user

Query
- ?- mortal(X).
- Return X = socrates.

Arity
- Number of arguments in a predicate
- mortal(X): arity = 1
- parent(peter, mary): arity = 2
- Represented as mortal/1, parent/2

Predicates with different arity
- Different
- name/2 ≠ name/3

```
person(socrates).

person(plato).

person(aristotle).

mortal(X) :- person(X).
```

Syntax of Prolog

Syntax

pred(arg1, arg2, ... argN).

- pred: name of predicate
- arg1... : arguments
- N: arity
- .: syntactic end of Prolog clause
- E.g. parent(peter, mary) ← a wrong syntax

Predicate of arity 0

- pred.

Data types

Integer

Atom

- Text constant
- Begin with a **lowercase** letter

Variable

- Begins with an **uppercase** letter, or **underscore** (_)

Structure

- Complex terms, list

Facts

Data or information of Prolog program

E.g. customer/3

```
customer('John Jones', boston, good_credit).
customer('Sally Smith', chicago, good_credit).
```

Uppercase letter

Blank

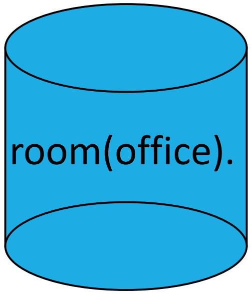
customer	Name	Place	Credit
1	‘John Jones’	boston	good
2	…	…	…

Simple Queries

Prolog Queries

Work by pattern matching

- ?- room(X).
- Check if Query = facts or rules in DB



Query = goal

- If there is a fact that matches the goal
  - Query succeeds → Responds ‘yes’
  - Otherwise → Responds ‘no’

Unification

Process of pattern matching

- For computer
- room(X) = room(office).

Rules of unification

- Predicate name in the goal and the one in DB are the same
- Both predicates have same arity
  - Same number of arguments
- All of the arguments are the same
  - Variables can be instantiated
  - Constants cannot be instantiated

Bindings of Variables

When a logical variable is assigned

- A value /a structure /a term
- Logical variable is bound
- E.g. A = 1
  - A is bound to the value of 1
  - “Binding of A” = 1
- Variable once bound, cannot change value
  - E.g. ?- A=1, A=2.
  - Fail
  - Can unbind A by backtracking automatically

Bindings of Variables

```
?-room(X).
X= kitchen
```

```
?-room(X), X = 1.
fail
```

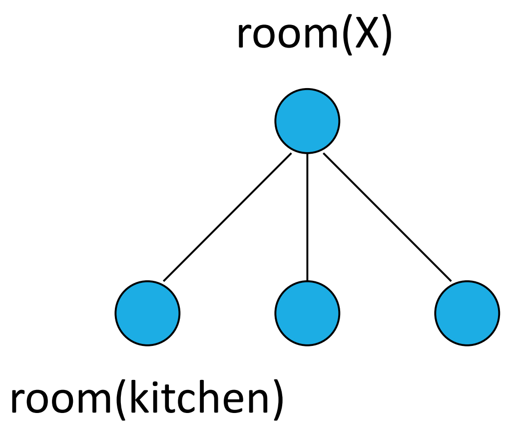
Unbind value by backtracking

- Use (;) to find alternatives
- no → no more answer

```
?-room(X).
X= kitchen;
X= office;
X= hall;
X= 'dinning room';
X= cellar;
no
```

Prolog program

```
room(kitchen).
room(office).
room(hall).
room('dining room').
room(cellar).
```



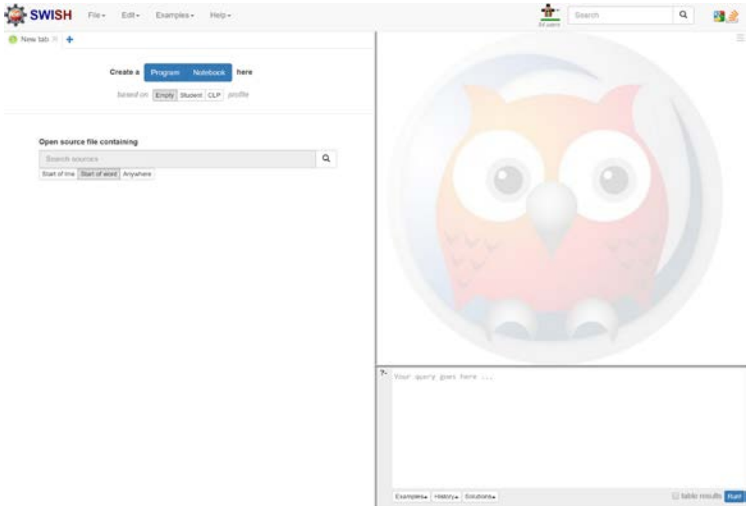
To use Prolog

Online

- http://swish.swi-prolog.org/

Download

- http://www.swi-prolog.org/download/stable



To use Prolog

Unix: type ‘pl’.

Windows: double click the icon

Writing prolog program

- Consult the program
  - ?- consult(‘a.pl’).
  - ?- consult(a).
- Ask your questions

If program is changed

- Consult the program again

Exercise 1

Family database

- Create a Prolog file called “family.pl”
- Add members of family
  - male/1, female/1
  - male(dennis). female(diana).
- Add predicate that records parent-child relationship
  - parent(diana, dennis).

Queries

Consult program “family.pl”

Ask queries

- ?- male(dennis).
- ?- male(X).
- ?- parent(X, dennis).
- ?- parent(dennis, X).
- ?- parent(diana, _). Don’t care
- ?- parent(X, Y).

Exercise 2

Family database

- Write queries
  - Confirm a parent relationship
  - Find someone’s parent
  - Find someone’s children
  - List all parent-children
    - Semi-colon “;”
    - Predicate “fail”
- Compound queries to find family relationships
  - Father, mother, sons, daughters, grandmothers, grandfathers

Prolog Rules

Prolog Rules

Stored query

Syntax
- head :- body.
- where_food(X,Y) :- location(X,Y), edible(X).

head
- Predicate definition (like a fact)

:-
- Neck symbol, read as "if"

body
- One or more goals
- Simple or compound query

Example

?- where_food(X, kitchen).

- X = apple ;
- X = crackers ;
- no

?- where_food(Thing, 'dining room').

- no

?- where_food(apple, kitchen).

- yes

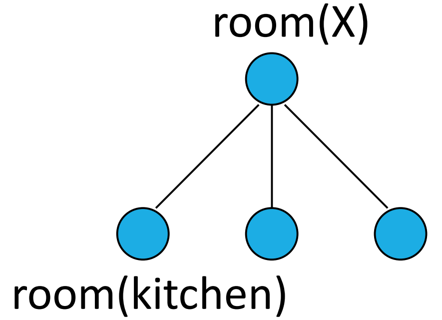
```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).

edible(apple).
edible(crackers).

tastes_yucky(broccoli).

here(kitchen).
```

Multiple Rules

Have multiple facts

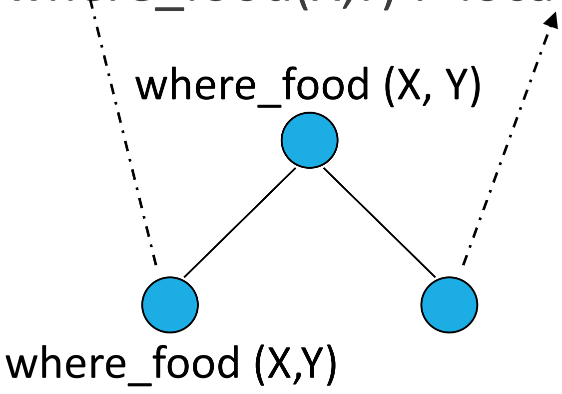
Have multiple rules

- Defining a predicate



For example

- where_food(X,Y) :- location(X,Y), edible(X).
- where_food(X,Y) :- location(X,Y), tastes_yucky(X).



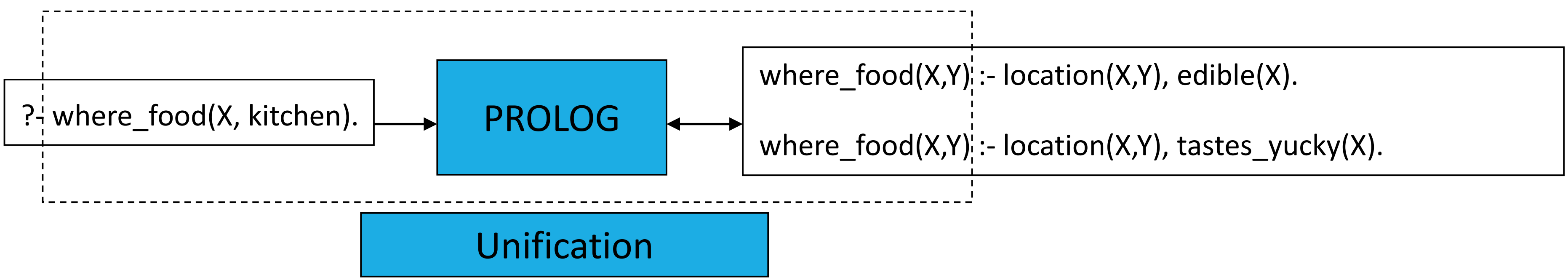

```
?- where_food(X, kitchen).
X = apple ;
X = crackers ;
X = broccoli ;
no
```

How Rules Work?

Unification

- Unify goal pattern with head of the rule
- Succeeds
  - Initiate a new query
    - With goals in body of the rule

```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).

edible(apple).
edible(crackers).

tastes_yucky(broccoli).

here(kitchen).
```



Using Rules

Solve the problem of door/2

- Define a two-way predicate

```
connect(X,Y):- door(X,Y).
connect(X,Y):- door(Y,X).
```

Means
'OR'

```
door(office, kitchen).
```

```
?- connect(kitchen, office).

yes

?- connect(office, kitchen).

yes
```

Exercises

Family database

- Build a rule for siblings
  - siblings(X,Y) :- parent(P,X), parent(P,Y).
  - Allow an individual as his/her own sibling
- Fix the problem
  - With a built-in predicate "Not Equal" \=
  - siblings(X,Y) :- parent(P,X), parent(P,Y), X \= Y.
- Use sibling to build rules
  - Brothers, sisters, uncles, aunts
  - brothers(X,Y) :- siblings(X,Y), male(X).
  - sisters(X,Y) :- siblings(X,Y), female(X).
  - uncles(X,Y) :- parent(P,Y), brothers(X,P).
  - aunts(X,Y) :- parent(P,Y), sisters(X,P).

Exercises

Define married/2 by using facts spouse/2

- married(X,Y) :- spouse(X,Y).
- married(X,Y) :- spouse(Y,X).

Define uncles & aunts using married/2 further

- uncles(X,Y):- parent(P,Y), brothers(X,P).
- uncles(X,Y):- aunts(A,Y), married(X,A).

- aunts(X,Y):- parent(P,Y), sisters(X,P).
- aunts(X,Y):- uncles(U,Y), married(X,U).

Arithmetic & Logic

Arithmetic

Evaluation of arithmetic expression

- Built-in predicate: ‘is’
- X is <arithmetic expression>
- E.g. Tmp = X, X is Tmp + 1.

Variable X

- Set to value of the arithmetic expression

Examples

Use Prolog as calculator

- ?- X is 2 + 2.
- X = 4
- ?- X is 3 * 4 + 2.
- X = 14

Parentheses clarify precedence

- ?- X is 3 * (4 + 2).
- X = 18
- ?- X is (8 / 4) / 2.
- X = 1

Comparison

Prolog provides a number of operators

- X > Y
- X < Y
- X >= Y
- X =< Y

```
?- X is 2 + 2, X > 3.
X = 4
?- X is 2 + 2, 3 >= X.
no
?- 3+4 > 3*2.
yes
```

Recursion

Recursion

Ability for a unit of code to call itself

- Repeatedly if necessary
- Prolog
  - A predicate contains a goal that refers to itself

At least two parts

- Boundary / termination condition
- Recursive case

Recursive Definition

Boundary condition

- Simple case that we know to be true

Recursive case

- Simplify the problem
  - First remove a layer of complexity
  - Then call itself
- At each level
  - Check boundary condition
  - If it is reached, recursion ends
  - Otherwise, recursion continues

Example

Flashlight is in desk, and desk is in office.

?- location(flashlight, office).

no

```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).
```

is_contained_in/2

- Dig through layers of nested things

Boundary condition

- T1 is directly located in T2
- is_contained_in(T1,T2) :- location(T1,T2).

Simplify & recur

- T1 is contained in **X**, some intermediate thing, which is located in T2
- is_contained_in(T1,T2) :- location(X,T2), is_contained_in(T1,X).

Exercises

Family database

- Use recursion to write ancestor/2

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

  ancestor(X,Y) :- parent(P,Y), ancestor(X, P).

- Use ancestor/2
  - Find all of a person's ancestors

    all_ancestor(X) :- ancestor(Y, X), write(Y), nl, fail.
  - Find all of a person's descendants

    all_descendent(X) :- ancestor(X, Y), write(Y), nl, fail.

Unification

Unification

Built-in pattern-matching algorithm

- Make two items identical

Unification process

- Variable & any term

  A = abc

  A = f(a,b)
- Constant & constant

  a = a

  abc = abc
- Structure & structure

  f(a,g(b,c)) = f(a,X)

Explicit Unification “=”

Built-in predicate =/2

- Succeed if two arguments unify
- arg1 = arg2
- =(arg1, arg2)

Warning

- Do not cause arithmetic evaluation
  - Evaluation is done by “is/2”

Example without Variable

```
?- a = a.
yes

?- a = b.
no

?- location(apple, kitchen) =  location(apple, kitchen).
yes

?- location(apple, kitchen) =  location(pear, kitchen).
no

?- a(b,c(d,e(f,g))) = a(b,c(d,e(f,g))).
yes

?- a(b,c(d,e(f,g))) = a(b,c(d,e(g,f))).
no
```

Example for a Variable & a Constant

```
?- X = a.
X = a

?- 4 = Y.
Y = 4

?- location(apple, kitchen) = location(apple, X).
X = kitchen

?- location(X,Y) = location(apple, kitchen).
X = apple
Y = kitchen

?- location(apple, X) = location(Y, kitchen).
X = kitchen
Y = apple
```

Example for Variables

```
?- X = Y.
X = _01
Y = _01
```

```
?- location(X, kitchen) = location(Y, kitchen).
X = _01
Y = _01
```

Bound Variables

```
?- X = Y, Y = hello.
X = hello
Y = hello

?- X = Y, a(Z) = a(Y), X =
hello.
X = hello
Y = hello
Z = hello
```

```
?- X = Y, Y = 3, write(X).
3
X = 3
Y = 3

?- X = Y, tastes_yucky(X),
write(Y).
broccoli
X = broccoli
Y = broccoli
```

Structures with Variables

```
?- X = a(b,c).
X = a(b,c)

?- a(b,X) = a(b,c(d,e)).
X = c(d,e)

?- a(b,X) = a(b,c(Y,e)).
X = c(_01,e)
Y = _01

?- a(b,X) = a(b,c(Y,e)), Y = hello.
X = c(hello, e)
Y = hello
```

```
?- food(X,Y) = Z, write(Z), nl,
tastes_yucky(X), edible(Y),
write(Z).
food(_01,_02)
food(broccoli, apple)
X = broccoli
Y = apple
Z = food(broccoli, apple)
```

Important to Note

If a value unified to a variable in later goal

- Conflicts with pattern set earlier, unification fails

```
?- a(b,X) = a(b,c(Y,e)), X = hello.
no
```

Second goal fails

- No value of X
- Allow hello to unify with c(Y,e)

```
?- a(b,X) = a(b,c(Y,e)), X = c(hello, e).
X = c(hello, e)
Y = hello
```

Succeed

- Same structure can be unified

Important to Note

?- a(X) = a(b,c).

no

?- a(b,c,d) = a(X,X,d).

no

Fail

- Pattern asks that the first two arguments be the same

?- a(c,X,X) = a(Y,Y,b).

no

No value of X and Y allow the two structures to unify

- c = Y
- X = Y
- X = b
- ➔ c = b?

Important to Note

Anonymous variable (_)

- Wildcard variable
- Do not bind to values
- Multiple occurrences do not imply equal values

```
?- a(c,X,X) = a(_,_,b).
X = b
```

Implicit unification

- Prolog searches for the head of a clause
  - Match a goal pattern
  - ?- food(Z, kitchen).

```
food(X, Y):- location(X, Y),
edible(X).
```

Exercises

Predict results of unification

?- a(b,c) = a(X,Y).

- X = b, Y = c

?- a(X,c(d,X)) = a(2,c(d,Y)).

- X = 2, Y = 2

?- a(X,Y) = a(b(c,Y),Z).

- X = b(c, _01), Y = _01, Z = _01

?- tree(left, root, Right) = tree(left, root, tree(a, b, tree(c, d, e))).

- Right = tree(a, b, tree(c, d, e))

List

List

Powerful data structure

- Hold and manipulate groups of things

List in Prolog

- Collection of terms
- Atoms, integers, structures, lists

Represented as

- [apple, broccoli, refrigerator]

List

Comparison

- Without list
  - location(apple, kitchen).
  - location(broccoli, kitchen).
  - location(crackers, kitchen).
- With list
  - loc_list([apple, broccoli, crackers], kitchen).

Special list

- Empty list
  - Nil represented as []
  - loc_list([], hall).

List

Work as usual terms

```
loc_list([apple, broccoli, crackers], kitchen).
```

?- loc_list(X, kitchen).

X = [apple, broccoli, crackers]

?- [_,X,_] = [apples, broccoli, crackers].

X = broccoli

Have to know

- Number of items in list
- Order of items

Special Notation

[H | T]

- Allow reference to
  - First element of the list
  - List of remaining elements

H

- Head
- Bound to the first element of the list

T

- Tail
- Bound to list of remaining elements

Unification Using Lists

?- [a|[b,c,d]] = [a,b,c,d].

yes

Succeed

- ‘a’ is a head (an atom)
- [b,c,d] is the tail (a list)

?- [a|b,c,d] = [a,b,c,d].

no

Fail

- The tail is not a list

Examples

?- [H|T] = [apple, broccoli, refrigerator].

H = apple

T = [broccoli, refrigerator]

?- [H|T] = [a, b, c, d, e].

H = a

T = [b, c, d, e]

?- [H|T] = [apples, bananas].

H = apples

T = [bananas]

?- [H|T] = [a, [b,c,d]].

H = a

T = [[b, c, d]]

?- [H|T] = [apples].

H = apples

T = []

Tail is empty list

?- [H|T] = [].

no

Do not unify

- Empty list has no head

Multiple Heads

Before the bar (|)

- Can specify more than one element

```
?- [One, Two | T] = [apple, sprouts, fridge, milk].
One = apple
Two = sprouts
T = [fridge, milk]
```

```
?- [X,Y|T] = [a|Z].
X = a
Y = _01
T = _03
Z = [_01 | _03]
```

```
?- [H|T] = [apple, Z].
H = apple
T = [_01]
Z = _01
```

List Predicate

member/2

- Check membership of a term

```
member(H,[H|T]).
member(X,[H|T]) :- member(X,T).

?- member(banana, [apple, broccoli, crackers]).
no

?- member(X, [apple, broccoli, crackers]).
X = apple ;
X = broccoli ;
X = crackers ;
no
```

append/3
Recursive Definition

Build lists from other lists

Split lists into separate pieces

?- append([a,b,c],[d,e,f],X).

X = [a,b,c,d,e,f]

Boundary condition

- [] + X = X

```
append([],X,X).

append([H|T1],X,[H|T2]) :- append(T1,X,T2).
```

Recursive case

- [H|T1] + X = [H|T2]
- T2 = T1+X

```
[1|[]] + [a,b] = [1|T2]

T2 = [] + [a,b] = [a,b]
```

Exercises

Remove a given element from a list

```
remove(H, [], []).
remove(H, [H|T], T).
remove(X, [H|T], [H|T2]) :- remove(X, T, T2).
```

Get last element of a list

```
last([E], E).
last([H|T], E) :- last(T, E).
```

Count elements in a list

```
len([], 0).
len([H|T], Count) :- len(T, NewCount), Count is 1 + NewCount.
```

Exercises

?- [] = [H|T].

no

?- [a] = [H|T].

H = a, T = []

?- [apple,3,X,'What?'] = [A,B|Z].

A = apple, B = 3, Z = [_01, ‘What?’]

?- [[a,b,c],[d,e,f],[g,h,i]] = [H|T].

H = [a,b,c], T = [[d,e,f], [g,h,i]]

Control Structures

Tail Recursion

Traditional recursion

- $N! = (N - 1)! \cdot N$
- Reduce problem into smaller one
- Problem is not solved until $(N-1)!$ is known
- Stack is necessary to keep information

Iteration

- $N! = N \cdot (N - 1)!$
- No stack is necessary

Tail recursion

- Combine the advantages

Example

```
factorial_1(1,1).
factorial_1(N,F):-
  N > 1,
  NN is N - 1,
  factorial_1(NN,FF),
  F is N * FF.
```

```
factorial_2(1,F,F).
factorial_2(N,T,F):-
  N > 1,
  TT is N * T,
  NN is N - 1,
  factorial_2(NN,TT,F).
```

Need a stack

Handle large amount of value simultaneously