# Programming with Lists

## CS3100 Fall 2019

# Review

## Previously

- Solving a logic puzzle

## This lecture

- Programming with Lists
- And many concepts in the process
  - Arithmetic
  - Last call optimisation
  - Backtracking & Choice points.

# Trouble with Zebras

- Our zebra puzzle only works with 5 houses.
  - Problem is the 5-tuple `(H1,H2,H3,H4,H5)`.
- Need to rewrite the entire program if more houses were added.
- What we really need is lists...

# Support for lists in Prolog

- Notated with square brackets `[1,2,3,4]`.
- The empty list is `[]`.
- List pattern matching is `[H | T]`, where `H` is a list element and `T` is a list.
  - Can also match `[1,2,3 | T]`.

# Finding the last element of the list

In [1]:

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

```
Added 2 clauses(s).
```

In [2]:

```
?- last([1,2],X).
```

```
X = 2 .
```

# Tracing the example by hand

```
last([1,2],X).
```

# Tracing the example in SWI-Prolog

```
last ([1,2],X)
```

# Quiz

What happens if I ask for `last([],X)` ?

1. pattern match exception
2. Prolog says false.
3. Prolog says true, X = [].
4. Prolog says true, X = ???.

# Quiz

What happens if I ask for `last([],X)` ?

1. pattern match exception
2. Prolog says false. ✓
3. Prolog says true, X = [].
4. Prolog says true, X = ???.

# Arithmetic

- How do we compute the length of the list?
  - We need support for arithmetic.
- Arithmetic is quite natural in imperative and functional paradigms.
  - Since computation is deduction in logic programming, arithmetic is quite special.

# Arithmetic equality != Unification

`=` operator is used up by unification.

In [3]:

```
?- A = 1+2.
```

```
A = +(1, 2) .
```

In [4]:

```
?- 1+2 = 3.
```

```
false.
```

In [5]:

```
?- A = money+power.
```

```
A = +(money, power) .
```

## Use the `is` operator

The “is” operator tells prolog to evaluate the righthand expression numerically and unify with the left.

In [6]:

```
?- X is 1, A is X+2, X is 2.
```

```
false.
```

In [7]:

```
?- A is money+power.
```

```
ERROR: Caused by: '  A is money+power'. Returned: 'error(type_error(ev
aluable, /(power, 0)), context(:(system, /(is, 2)), _1768))'.
```

## Restriction on `is` operator

The RHS must be a ground term (no variables).

In [8]:

```
?- A is B+2.
```

```
ERROR: Caused by: '  A is B+2'. Returned: 'error(instantiation_error,
context(:(system, /(is, 2)), _1914))'.
```

In [9]:

```
?- 3 is B+2.
```

```
ERROR: Caused by: '  3 is B+2'. Returned: 'error(instantiation_error,
context(:(system, /(is, 2)), _2048))'.
```

## Quiz

What is the result of `A is *(3,+(1,2))`?

1. Error
2. 9
3. 8
4. 6

## Quiz

What is the result of `A is *(3,+(1,2))`?

1. Error
2. 9 ✓
3. 8
4. 6

## Arithmetic

There is support for `+,*, /, <, =<, >, >=` ,etc.

In [10]:

```
?- 20 / 20.
```

```
ERROR: Caused by: '  20 / 20'. Returned: 'error(type_error(lambda_fre
e, 20), _2146)'.
```

## List Sum

Compute the sum of the list. This is the example we saw in the first Prolog lecture.

In [11]:

```
sum([],0).
sum([H | T], N) :- sum(T,M), N is M+H.
```

```
Added 2 clauses(s).
```

In [12]:

```
?- sum([1,2,3],X).
```

```
X = 6 .
```

In [13]:

```
?- sum(X,3).
```

```
ERROR: Caused by: '  sum(X,3)'. Returned: 'error(instantiation_error,
context(:(system, /(is, 2)), _2280))'.
```

## Length of list

In [14]:

```
len([],0).
len([_ | T], N) :- len(T,M), N is M+1.
```

```
Added 2 clauses(s).
```

In [15]:

```
?- len([1,2,3],X).
```

```
X = 3 .
```

## Last call optimisation

- `len` uses O(N) stack space.

## Trace `len` by hand

```
?- len([1,2],X)
```

## Tail recursive length

In [16]:

```
len2([],Acc,Acc).
len2([H|T],Acc,N) :- M is Acc+1, len2(T,M,N).
```

```
Added 2 clauses(s).
```

In [17]:

```
?- len2([1,2],0,X).
```

```
X = 2 .
```

# Trace `len2` by hand.

```
?- len2([1,2],0,X).
```

# Predicate Overloading

In [18]:

```
len2(L,X) :- len2(L,0,X).
```

```
Added 1 clauses(s).
```

In [19]:

```
?- len2([1,2,3],X).
```

```
X = 3 .
```

# Last Call Optimisation

- This technique is applied by the prolog interpreter
- The last clause of the rule is executed as a branch and not a call
- We can only do this if the rule is **determinate** up to that point
  - No further choices for the rule
  - Relates to **choice points** (to be seen).

# List append

In [20]:

```
append([],Q,Q).
append([H | P], Q, [H | R]) :- append(P, Q, R).
```

```
Added 2 clauses(s).
```

In [21]:

```
?- append([1,2],X,[1,2,3,4]).
```

```
X = [ 3, 4 ] .
```

# Prefix and Suffix

Prefix and Suffix of list can be defined using append.

In [22]:

```
prefix(X,Z) :- append(X,Y,Z).
suffix(Y,Z) :- append(X,Y,Z).
```

```
Added 2 clauses(s).
```

# Prefix and Suffix

In [23]:

```
?- prefix(X,[1,2,3]).
```

```
X = [  ] ;
X = [ 1 ] ;
X = [ 1, 2 ] ;
X = [ 1, 2, 3 ] .
```

In [24]:

```
?- suffix(X,[1,2,3]).
```

```
X = [ 1, 2, 3 ] ;
X = [ 2, 3 ] ;
X = [ 3 ] ;
X = [  ] .
```

## Backtracking

The way prolog fetches multiple results for the given query is through Backtracking.

## Trace `prefix` by hand

```
?- prefix([1,2],X).
```

## Choice Points

- Choice points are locations in the search where we could take another option.
- If there are no choice points left then Prolog doesn't offer the user any more answers

## Quiz

What is the first result of query `len(A,2)` ?

1. Error due uninstantiated arithmetic expression.
2. A = [_,_]
3. Query runs forever

4. Error due to invalid arguments

# Quiz

What is the first result of query `len(A,2)` ?

1. Error due uninstantiated arithmetic expression.
2. A = [_,_] ✓
3. Query runs forever
4. Error due to invalid arguments

# Trace `len` by hand

```
?- len(A,2)
```

# Quiz

What is the second result of query `len(A,2)` ?

1. Error due uninstantiated arithmetic expression.
2. A = [_,_]
3. Query runs forever
4. Error due to invalid arguments

# Quiz

What is the second result of query `len(A,2)` ?

1. Error due uninstantiated arithmetic expression.
2. A = [_,_]
3. Query runs forever ✓
4. Error due to invalid arguments

# Trace `len` by hand

```
?- len(A,2)
```

# Limiting the number of results

In [25]:

```
?- len(A,2) {1}.
```

```
A = [ _2380, _2386 ] .
```

# Fin.