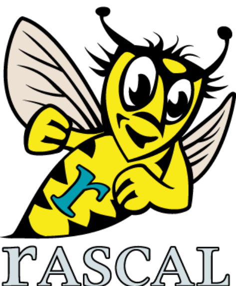
Lists

Operators, Functions, Matching

Paul Klint

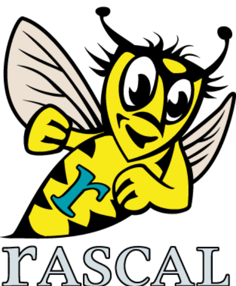
Overview

- Definition of list
- Some history
- Operators on lists
- List comprehensions
- Lists in lists and list splicing
- Library functions for lists
- List matching



A List ...

- Implements an ordered collection of values
  - Values may be repeated
- Has type list[ElementType]
- Is immutable
- Example:
  - [4, 5, 1, 4, 9, 3]
  - [“The”, “world”, “of”, “Warcraft”]



A more formal definition

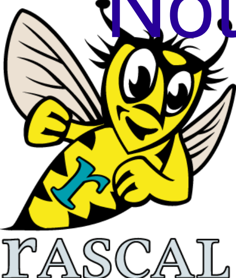
Let LIST and ELEM be the type of lists, resp. list elements. Lists can now be defined as:

```
data LIST = nil() | cons(ELEM elm, LIST lst);
```

And the operations first and rest as follows:

```
ELEM first(cons(e, lst)) = e;

LIST rest(cons(e, lst)) = lst;
```

Note: first(nil()) and rest(nil()) are undefined!



Applying this definition

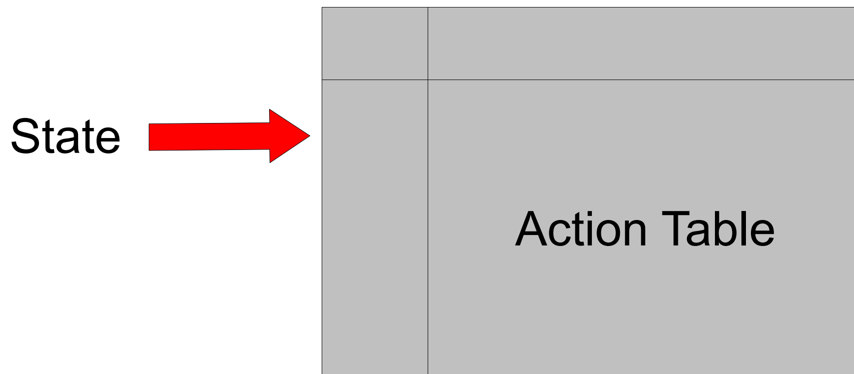
- The list [3, 1, 2] can be represented as
  - cons(3, cons(1, cons(2, nil())))
- And we have:
  - first(cons(3, cons(1, cons(2, nil())))) = 3
  - rest(cons(3, cons(1, cons(2, nil())))) =
    cons(1, cons(2, nil()))
- These definitions form the basis of lists but they are seldomly used in this form.

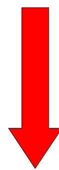
Some List History

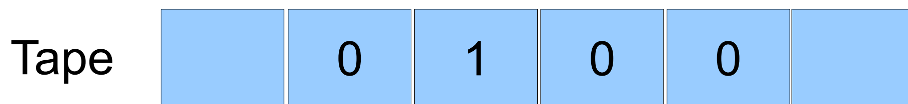
Alan Turing (1912-1954)

- Logician, cryptanalyst, computer scientist
- Inventor of Turing machine (list-based!)
- Deciphered German enigma code
- Suicide after chemical castration

Turing Machine



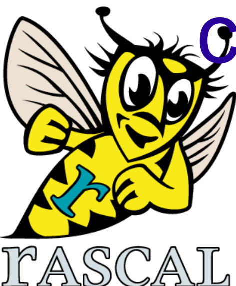




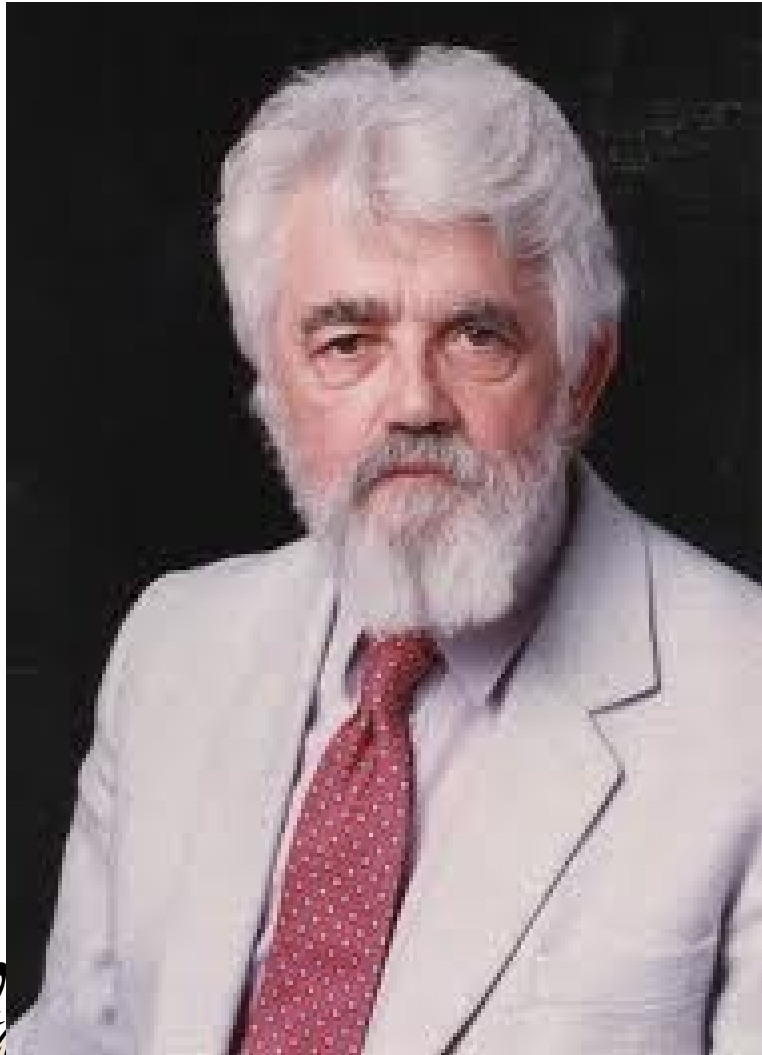
- Infinite tape
- Symbols can be read/written/erased
- State register
- Action table:
  - State
  - Symbol
  - Action (E/W, L/R)
  - NextState

Turing Machines

- Captures the essence of the concept *algorithm*
- In Turing's words: *It is possible to invent a **single** machine which can be used to compute **any** computable sequence*.
- Forms the theoretical basis for computers
- Also characterizes the limits of computing: the halting problem
- Lists (e.g., the tape) can be used to model any computation



John McCarthy (1927-2011)



- Computer scientist
- Invented the term "Artificial Intelligence" (AI)
- Invented Lisp, a list-based programming language for AI.
- Scheme is based on Lisp

List Operators

Rascal/Expressions/Values/List

- Append, Insert, Concatenation: +
- Difference: -
- Intersection: &
- Product: *
- In: in, NotIn: notin
- Subscription

- Compare:
  - Equal: ==
  - NotEqual: !=
  - StrictSubList: <
  - SubList: <=
  - StrictSuperList: >
  - SuperList: >=

Append/Insert/Concatenate:+

- Append (element at end of list):
  - [3,2,4] + 7 gives [3,2,4,7]
- Insert (element at begin of list):
  - 3 + [2,4,7] gives [3,2,4,7]
- Concatenate (two lists):
  - [3, 2] + [4,7] gives [3,2,4,7]

Difference: -

- Difference (of two lists):
  - L1 - L2 returns L1 with each occurrence of an element of L2 removed from it
  - [3,2,4,7] - [6,5,3,4] gives [2,7]
  - [3,2,4,7,3] - [6,5,3,4] gives [2,7]

Intersection: &

- Intersection (of two lists):
  - L1 & L2 returns L1 with all elements that do *not* occur in L2 removed
  - [3,2,4,7] & [6,5,3,4] gives [3,4]

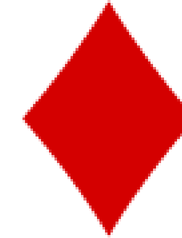
Exercises

- [13,9,21,13,7] + 13
- 7 + [13,9,21,13,7]
- [13,9,21,13,7] & [7,13]
- [13,9,21,13,7] - [13]
- ([13,9,21,13,7] - [13]) + ([5,1,3] & [1,2,3])

Product: *

- The product of two lists gives a list of tuples for each combination of elements from the two lists:
  - [1, 2, 3] * [4, 5, 6] gives [<1,4>, <1,5>, <1,6>, <2,4>, <2,5>, <2,6>, <3,4>, <3,5>, <3,6>]
- ["clubs", "hearts", "diamonds", "spades"] * [1 .. 12] gives ...

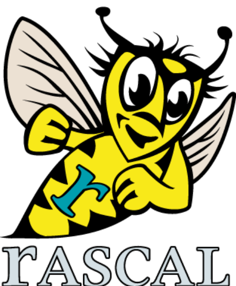
Card Deck!



```
[
  <"clubs",1>, <"clubs",2>, <"clubs",3>, <"clubs",4>, <"clubs",5>, <"clubs",6>,
  <"clubs",7>, <"clubs",8>, <"clubs",9>, <"clubs",10>, <"clubs",11>, <"clubs",12>,
  <"hearts",1>, <"hearts",2>, <"hearts",3>, <"hearts",4>, <"hearts",5>, <"hearts",6>,
  <"hearts",7>, <"hearts",8>, <"hearts",9>, <"hearts",10>, <"hearts",11>, <"hearts",12>,
  <"diamonds",1>, <"diamonds",2>, <"diamonds",3>, <"diamonds",4>, <"diamonds",5>,
  <"diamonds",6>, <"diamonds",7>, <"diamonds",8>, <"diamonds",9>, <"diamonds",10>,
  <"diamonds",11>, <"diamonds",12>,
  <"spades",1>, <"spades",2>, <"spades",3>, <"spades",4>, <"spades",5>, <"spades",6>,
  <"spades",7>, <"spades",8>, <"spades",9>, <"spades",10>, <"spades",11>, <"spades",12>
]
```

Membership: in, notin

- Test membership of element in list:
  - 3 in [6,5,3,4] gives true
  - 7 in [6,5,3,4] gives false
  - 7 notin [6,5,3,4] gives true



Subscription: [ ]

- Fetch or replace element at given index:
- L = [13, 22, 7]; M = L;
- L[1] gives 22
- Use in assignment:
  - L[1] = 33;
  - L now has value [13,33,7]
  - M still has the old value [13, 22, 7]

Comparison: ==, !=, <, <=, >, >=

- [1, 2, 3] == [1, 2, 3] gives true
- [1, 2, 3] == [3,2,1] gives false
- [1, 2, 3] < [1, 2, 3] gives false
- [1, 2, 3] <= [1, 2, 3] gives true
- [1, 2, 3] < [4, 3, 2, 1] gives true
- [1, 2, 3] < [1, 3, 4] gives false

Exercises

- [15, 3, 2, 7] < [7, 3, 2, 15, 16]
- 7 in ([15, 3, 2, 7] - [3, 15, 16])
- [15, 3, 2, 7] & [7, 3, 15, 16] == [15, 3]
- <"clubs", 3> in ["clubs", "hearts", "diamonds", "spades"] * [1 .. 12]
- ["a", "b"] * [1, 2]

May be more than one

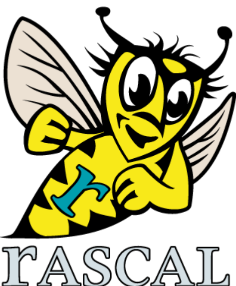
List Comprehensions

- $[E_1, ..., E_n \mid G_1, ..., G_n]$
- G_i: generators (produce values) or filters (select values)
- E_i: contributions to the resulting list, may use variables introduced in a G_i

Lists of lists

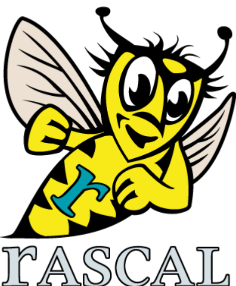
- Perfectly ok to have lists of lists of arbitrary depth:
  - [[1,2], [3,4,5,6], [7,8,9]] is of type list[list[int]]
- Consider:
  - L = [1, 2, 3];
  - What is the value of [10, L, 20]?

[10, [1, 2, 3], 20] and is of type list[value]



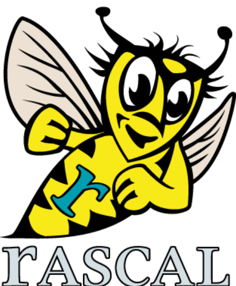
List Splicing: *

- List splicing is needed when inserting a list in another list without introducing an extra list level:
  - L = [1, 2, 3];
  - [10, *L, 20] gives [10, 1, 2, 3, 20]
- List splicing can also be used for contributions to a list comprehension:
  - [*L | L <- [[1,2], [3, 4, 5], [6, 7]]] gives
  - [1,2,3,4,5,6,7]



Exercises

- Given is L = [1, 2, 3]; Also give type of answer
- [L, L]
- [*L, L]
- [*L, *L]
- [*M | M <- [L, L, L]]

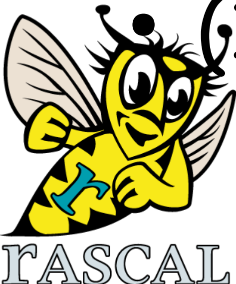


Reducer

- (Exp | RedExp | G_1, ..., G_n)
- Equivalent to:

```
it = Exp;
for(G1, ..., Gn)
    it = RedExp;
it;
```

- L = [1, 3, 5, 7];
- (0 | it + e | int e <- L) gives 16
- (1 | it * e | int e <- L) gives 105



Exercises

- Use a reducer to compute the sum of the squares of the elements of a list.
- Use a reducer to compute the minimum of a list.

Library Functions for Lists (> 40)

Rascal/Libraries/Prelude/List

- head, tail
- push, pop, top
- delete
- reverse
- domain, index
- size
- min, max, sum
- sort
- getOneFrom, takeOneFrom
- mapper

Examples

- Let L = [10, 11, 5]
- size(L) gives 3
- head(L) gives 10
- tail(L) gives [11,5]
- push(3, L) gives [3, 10, 11, 5]
- reverse(L) gives [5,11,10]
- sort(L) gives [5, 10, 11]
- index(L) gives [0,1,2]
- L[2] gives 5
- takeOneFrom(L) gives <11, [10,5]> Note: an arbitrary element!

mapper

- Takes a list and a function to be applied to each list element
- Define: int incr(int x) { return x + 1;}
- mapper([11,2,7], incr) gives [12,3,8]
- mapper([11,2,7], int(int x) { return x +1;}) gives [12,3,8]

Exercises

- Use `mapper` to square the numbers in a list.
- Write the same using a list comprehension.

List Matching

- Given a list pattern and a list value:
  - Determine whether the pattern matches the value
  - If so, bind any variables occurring in the pattern to corresponding subparts of the list value.
- Used with
  - Explicit match operator `Pattern := Value`
  - Switch: matching controls case selection
  - Visit: matching controls visit of tree nodes

List Matching

```
rascal> L = [1, 2, 3, 1, 2];
list[int]: [1,2,3,1,2]

rascal> [1, int X, 3, 1, 2] := L;
bool: true

rascal> X;
Error: X is undefined

rascal> if([1, int X, 3, 1, 2] := L) println("X = <X>");
X = 2
ok
```

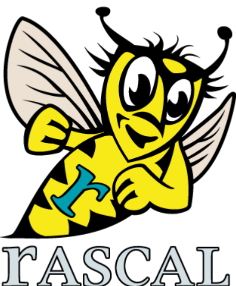
Pattern variable

X is bound but has limited scope

List Matching

```
rascal> L = [1, 2, 3, 1, 2];
list[int]: [1,2,3,1,2]

rascal> [1, int X, 3, 1, x] := L;
bool: true

rascal> [1, int X, 3, 1, x] :=[1, 2, 3, 1, 3];
bool: false
```

Use of Pattern variable



List Matching

```
rascal> L = [1, 2, 3, 1, 2];
list[int]: [1,2,3,1,2]

rascal> [list[int] X, 3, 1, 2] := L;
bool: true

rascal> [*X, 3, 1, 2] := L;
bool: true

rascal> [*X, 3, X] := L;
bool: true
```

Pattern variable, will be bound to [1,2]

*X is a list variable and abbreviates list[int] X

Use of Pattern variable

Note

- List matching is non-unitary (>1 solutions)
- E.g., [*L, *M] := [1, 2] has three solutions:
  - L == [ ], M == [1,2]
  - L == [1], M == [2]
  - L == [1,2], M == [ ]
- In boolean expressions, matching, etc. solutions are generated when failure occurs later on (local backtracking)
- Side effects are undone (using recovery cache)

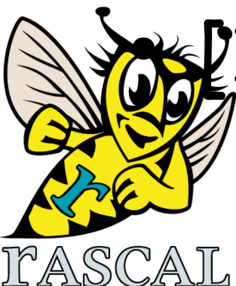
Iterate over the solutions

```
for([*L, *M] :=  [1,2,3]) println("<L>, <M>");
[], [1,2,3]
[1], [2,3]
[1,2], [3]
[1,2,3], []
list[void]: []
```

Exercises

Which matches succeed?; What are the bindings?

- [int X, int Y] := [10, 11, 3]
- [int X, int Y] := [10, 11]
- [int X, X] := [10, 11]
- [int X, X] := [10, 10]
- [10, list[int] L, 31] := [10, 9, 8, 7, 31]
- [10, *L, 31, L] := [10, 9, 8, 7, 31, 9, 8, 7]



Example: mirrored lists

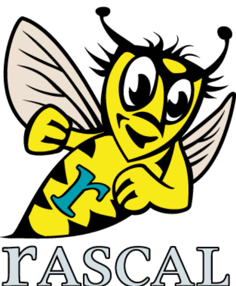
```
public bool isMirrored(list[int] L){
  return [*M1, *M2] := L && M1 == reverse(M2);
}
```

```
rascal>isMirrored([1,2,3,3,2,1])
bool: true
```

Unordered elements in list?

```
public bool unOrdered(list[int] L){
   return [*M1, int X, int Y, *M2] := L && X > Y;
}
```

```
rascal>unOrdered([1,3,2,4,5])
bool: true
```



Sorting Lists: Bubble sort

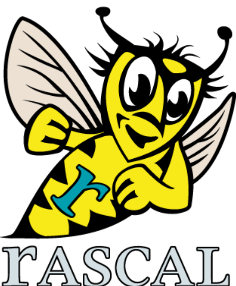
- One of the simplest (and slowest!) ways to sort
- Repeatedly exchange neighboring elements that are out of order
- See http://en.wikipedia.org/wiki/Bubble_sort

6 5 3 1 8 7 2 4

Sorting Lists

```
public list[int] sort1(list[int] Numbers){
  for(int I <- [0 .. size(Numbers) - 2 ]){
     if(Numbers[I] > Numbers[I+1]){
       <Numbers[I], Numbers[I+1]> = <Numbers[I+1], Numbers[I]>;
       return sort1(Numbers);
     }
  }
  return Numbers;
}
```

```
rascal>sort1([4,3,5,7,1]);
list[int]: [1,3,4,5,7]
```



Sorting Lists

```
public list[int] sort2(list[int] Numbers){
  while([list[int] Nums1, int P, list[int] Nums2, int Q, list[int] Nums3] := Numbers
        && P > Q)
        Numbers = Nums1 + [Q] + Nums2 + [P] + Nums3;
  return Numbers;
}
```

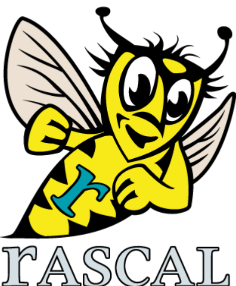
```
public list[int] sort3(list[int] Numbers){
  while([*Nums1, P, *Nums2, Q, *Nums3] := Numbers && P > Q)
        Numbers = Nums1 + [Q] + Nums2 + [P] + Nums3;
  return Numbers;
}
```

```
sort2([4,3,5,7,1]);
list[int]: [1,3,4,5,7]

rascal>sort3([4,3,5,7,1]);
list[int]: [1,3,4,5,7]
```

Sorting Lists

```
public list[int] sort4(list[int] Numbers){
  while([*Nums1, P, *Nums2, Q, *Nums3] := Numbers && P > Q)
        Numbers = [*Nums1, Q, *Nums2, P, *Nums3];
  return Numbers;

}
```

```
rascal>sort4([4,3,5,7,1]);
list[int]: [1,3,4,5,7]
```



How can we be sure that all these functions work properly? TEST!

- Try each function on representative inputs

```
test bool tsort1a() = sort1([3,2]) == [2, 3];
test bool tsort1b() = sort1([9,8,7]) == [7,8,9];
```

- Type in the console:

```
rascal>:test
ok
```

- Result in editor window:

```
88
89 test bool tsort1a() = sort1([3,2]) == [2, 3];
90 test bool tsort1b() = sort1([9,8,7]) == [7,8,9];
91
```

Property-based testing

What characterizes the result of a sort function?

```
public bool isSorted(list[int] L) =
       !any(int i <- index(L), int j <- index(L), i < j && L[i] > L[j]);
```

```
rascal>L = [4,3,5,7,1];
list[int]: [4,3,5,7,1]

rascal>isSorted(sort1(L));
bool: true

rascal>isSorted(sort2(L));
bool: true

rascal>isSorted(sort3(L));
bool: true

rascal>isSorted(sort4(L));
bool: true
```

Automatic Testing

- Write tests with parameters
- The Rascal test framework randomly generates argument values

```
test bool propSort1(list[int] Numbers) = isSorted(sort1(Numbers));
test bool propSort2(list[int] Numbers) = isSorted(sort2(Numbers));
test bool propSort3(list[int] Numbers) = isSorted(sort3(Numbers));
test bool propSort4(list[int] Numbers) = isSorted(sort4(Numbers));
```

- Let's try this:

```
103 test bool propSort1(list[int] Numbers) = isSorted(sort1(Numbers));
104 test bool propSort2(list[int] Numbers) = isSorted(sort2(Numbers));
105 test bool propSort3(list[int] Numbers) = isSorted(sort3(Numbers));
106 test bool propSort4(list[int] Numbers) = isSorted(sort4(Numbers));
107
```

Exercise

- Write a function `list[int] rev(list[int])` that reverses a list.
- Write a test to test `rev`

Keep in mind

Testing can only show the

presence of bugs

not their absence!