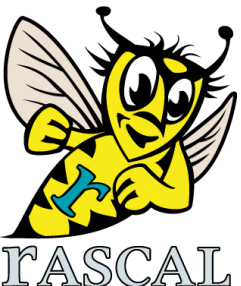
Sets

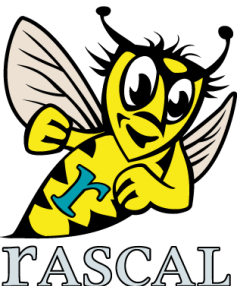
Operators, Functions, Matching

Paul Klint



Overview

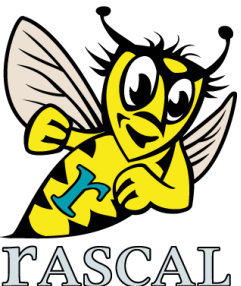
- Definition of set
- Some set history
- Operators on sets
- Set comprehensions
- Sets in sets and set splicing
- Library functions for sets
- Set matching



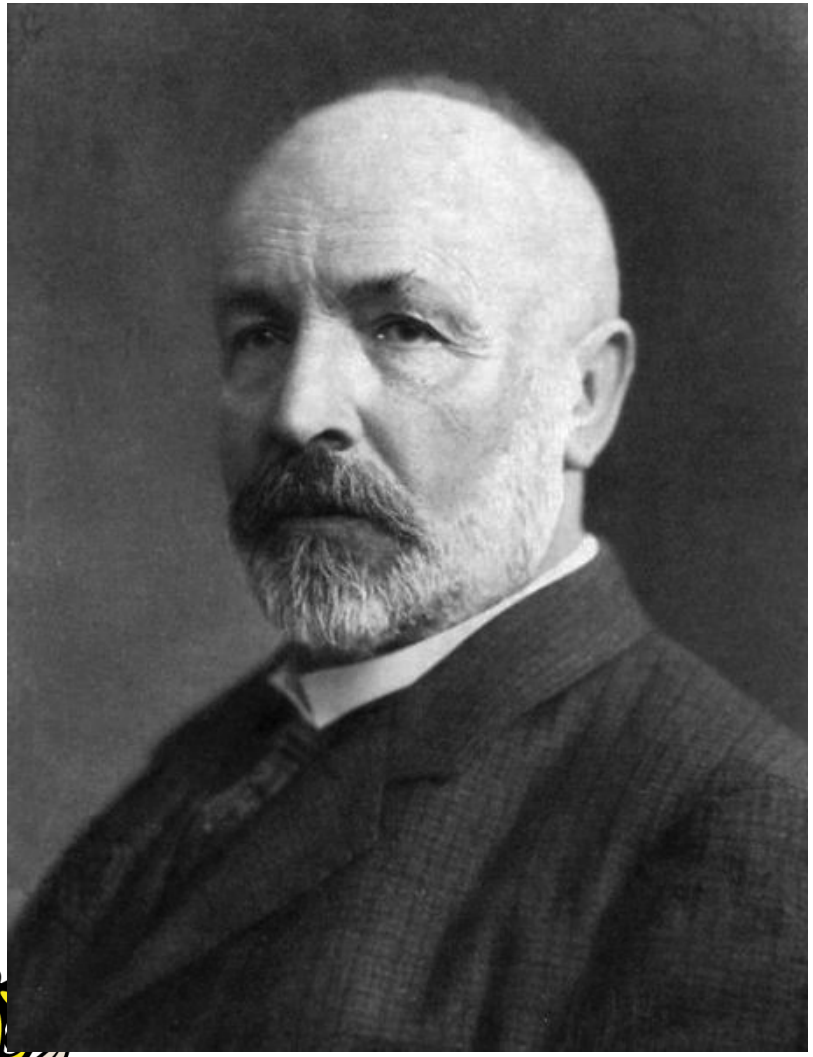
A Set ...

- Is a collection of well-defined, distinct, values:
  - An *unordered* collection
  - Values are distinct, i.e., they *cannot* be repeated
- Has type `set[ElementType]`
- Is immutable
- Examples:
  - {4, 5, 1, 9, 3}
  - {"Red", "Green", "Yellow", "Blue"}

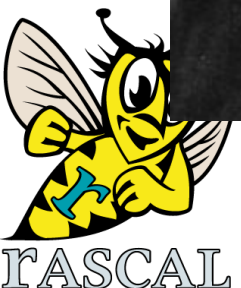
Some Set History



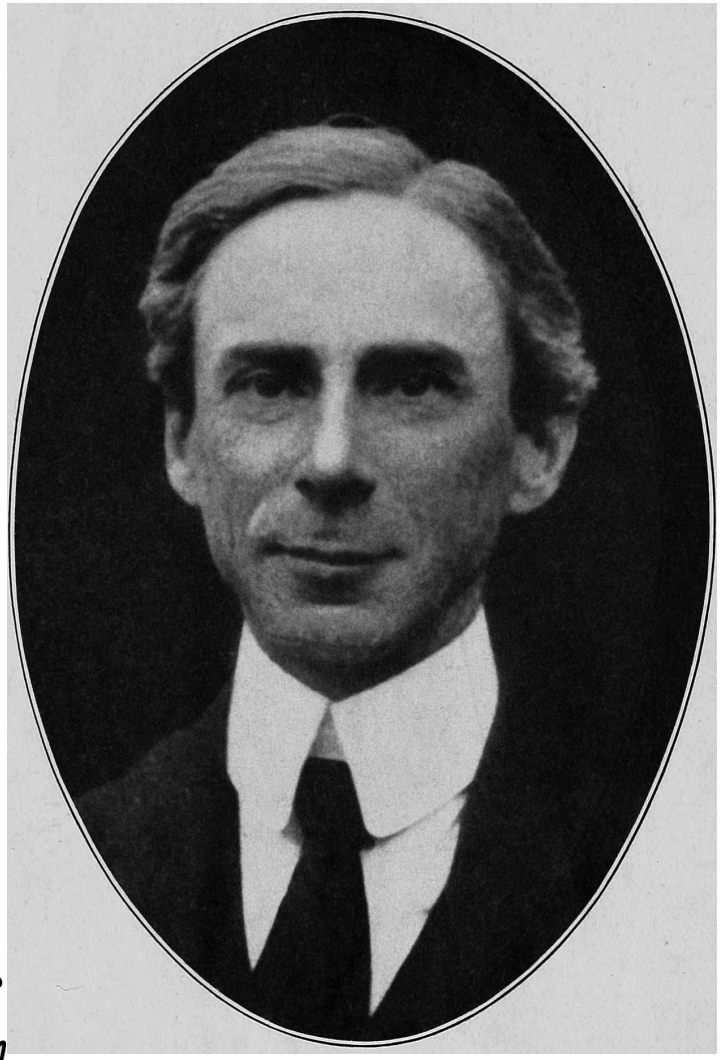
Georg Cantor (1845 - 1918)



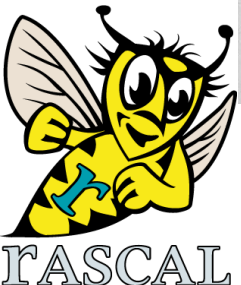
- German Mathematician
- Many contributions to the theory of numbers and functions
- Inventor of set theory
- Every definable collection is a set



Bertrand Russell (1872-1970)



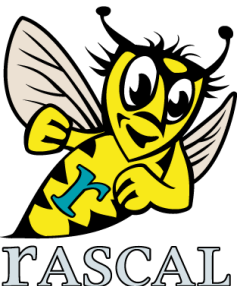
- British philosopher, logician and mathematician
- *Principia Mathematica* (with A.N. Whitehead): a formalization of mathematics



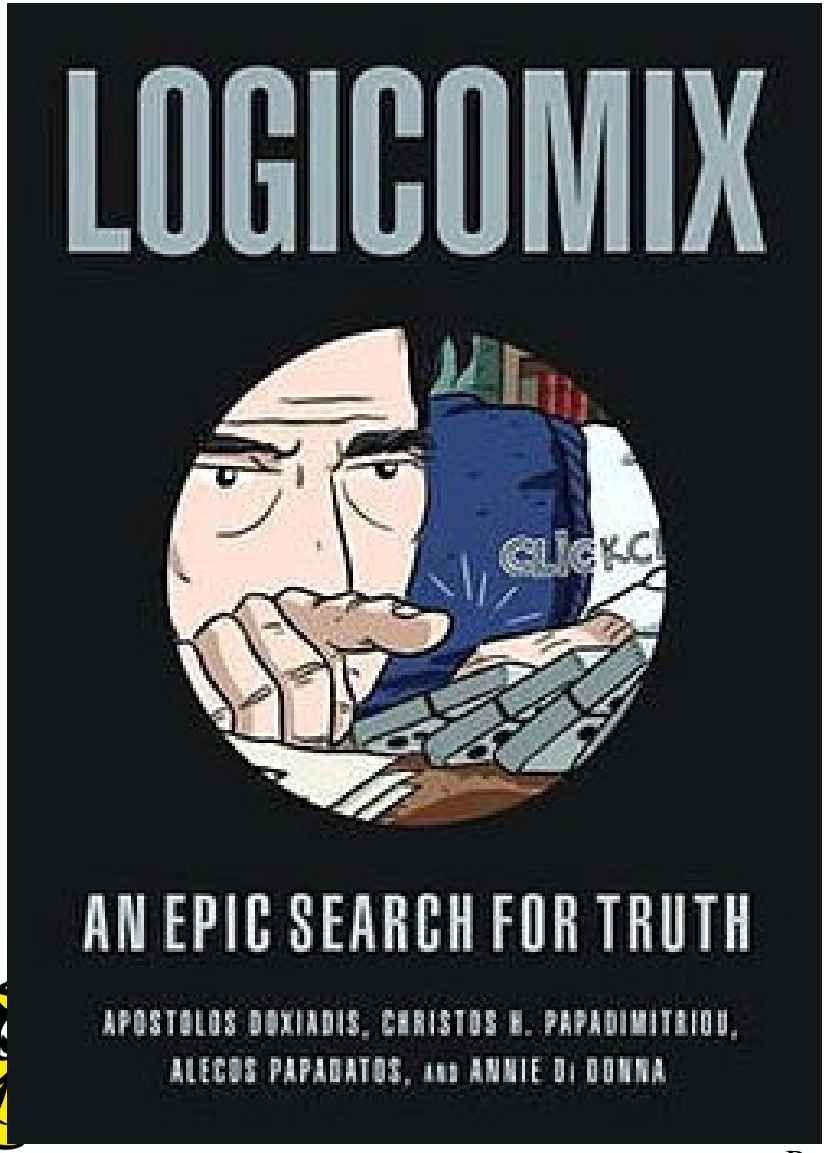
Russell's Paradox

- According to naive set theory every definable collection is a set.
- Let R be the set of all sets that are not members of themselves.

- **Case 1**: R is a member of itself => contradicts definition.
- **Case 2**: R is not a member of itself => it qualifies as a member
- $R = \{x \mid x \notin x\}$, then $R \in R \Leftrightarrow R \notin R$
- Better foundation of set theory needed!

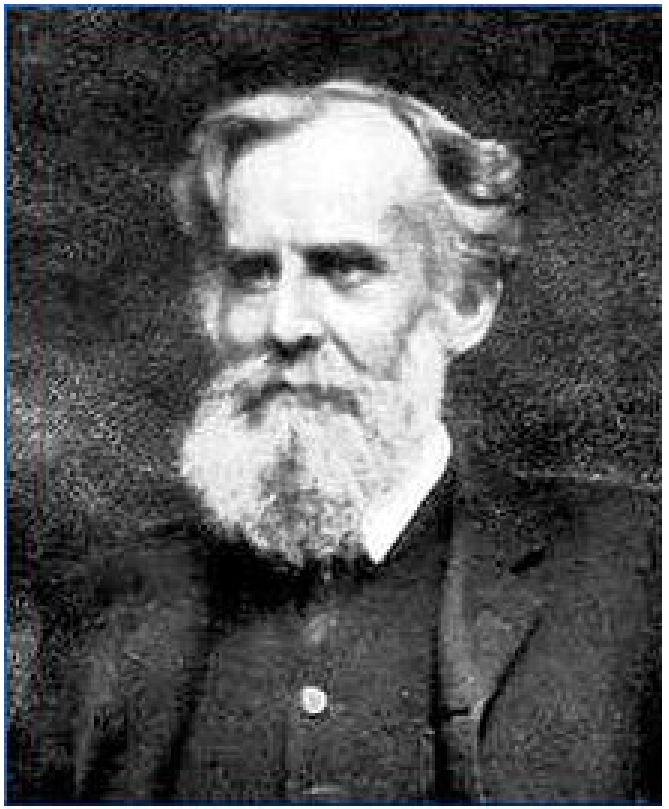


Logicomix

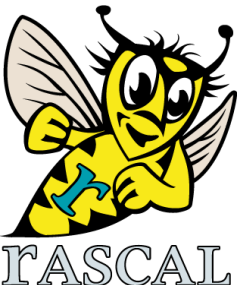


- Highly recommended (and entertaining) history of foundations of logic and mathematics

John Venn (1834-1923)

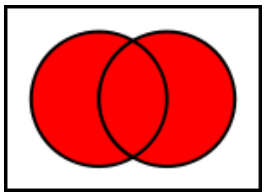


- British philosopher and logician
- Inventor of Venn diagrams

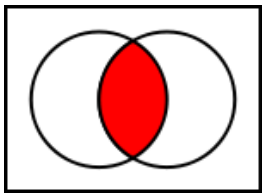


Venn Diagrams

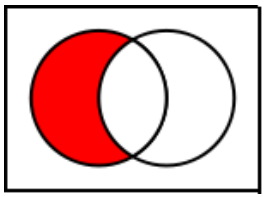
- Union: A + B

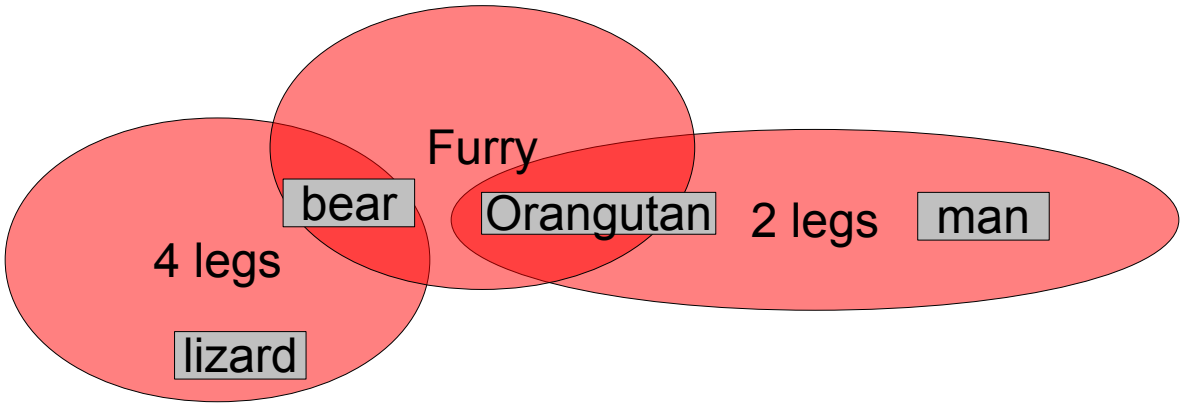


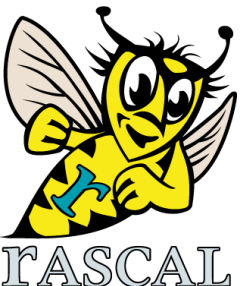
- Intersection: A & B



- Difference: A - B





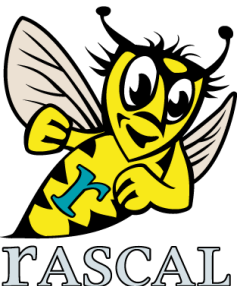


Some formal properties

- A + A = A (idempotent)
- A + B = B + A (commutative)
- (A + B) + C = A + (B + C) (associative)

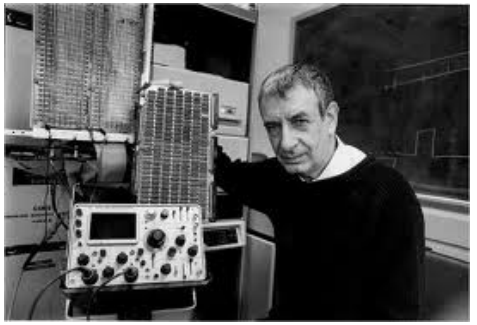
Exercise:

- Which of these properties hold for lists?



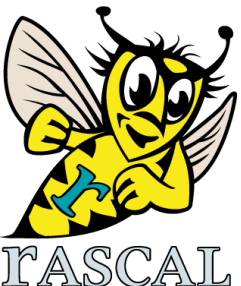
Sets in Programming Languages

- SETL (~1970)



Jack T. Schwartz and Robert B. Dewar

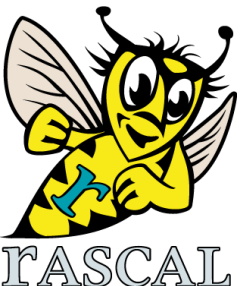
- Influenced ABC, a predecessor of Python, both designed at CWI



Set Operators

Rascal/Expressions/Values/Set

- Insert, Union: +
- Difference: -
- Intersection: &
- Product: *
- In: in, NotIn: notin
- Compare:
  - Equal: ==
  - NotEqual: !=
  - StrictSubSet: <
  - SubSet: <=
  - StrictSuperSet: >
  - SuperSet: >=

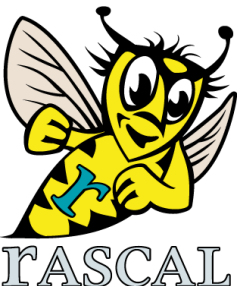


Insert/Union:+

- Insert (element in set):
  - {3,2,4} + 7 gives {3,2,4,7} *Order may differ!*
  - 3 + {2,4,7} gives {3,2,4,7}
  - 3 + {2,3,7} gives {2,3,7}
- Union (two sets):
  - {3, 2} + {4,7} gives {3,2,4,7}
  - {3, 2} + {4,2} gives {3,2,4}

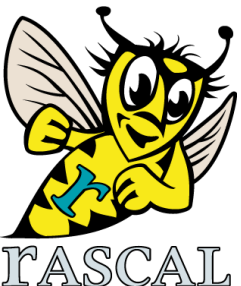
Difference: -

- Difference (of two sets):
  - S1 - S2 returns S1 with each occurrence of an element of S2 removed from it
  - {3,2,4,7} - {6,5,3,4} gives {2,7}
  - {3,2,4,7} - 4 gives {3, 2, 7}



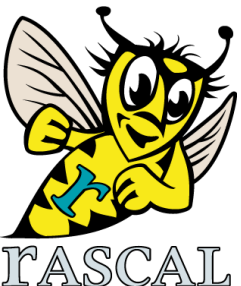
Intersection: &

- Intersection (of two sets):
  - S1 & S2 returns a set with all elements that occur both in S1 and S2
  - {3,2,4,7} & {6,5,3,4} gives {3,4}



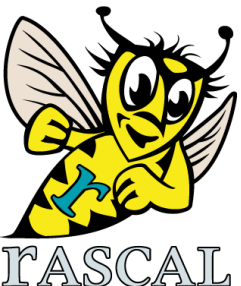
Exercises

- {9,21,13,7} + 13
- 8 + {9,21,13,7}
- {9,21,13,7} & {7,13}
- {9,21,13,7} - {13}
- ({9,21,13,7} - {13}) + ({5,1,3} & {1,2,3})



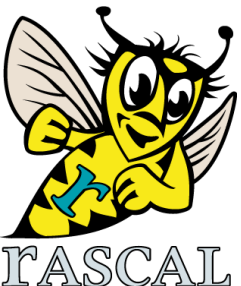
Product: *

- The product of two sets gives a set of tuples for each combination of elements from the two sets:
  - {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"} * toSet([1 .. 12]) gives (similar to lists) ...



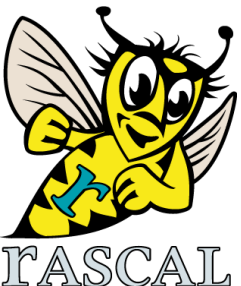
Card Deck!

```
{
  <"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 set:
  - 3 in {6,5,3,4} gives true
  - 7 in {6,5,3,4} gives false
  - 7 notin {6,5,3,4} gives true



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

- {1, 2, 3} == {1, 2, 3} gives true
- {1, 2, 3} == {3,2,1} gives true
- {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

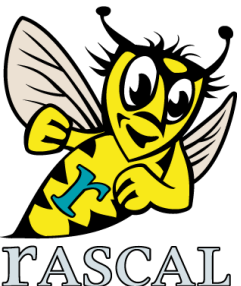
Exercises

- {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"} * toSet([1 .. 12])
- {"a", "b"} * {1, 2}

May be more than one

Set Comprehensions

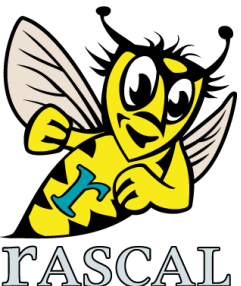
- $\{E_1, \ldots, E_n \mid G_1, \ldots, G_n\}$
- G_i: generators (produce values) or filters (select values)
- E_i: contributions to the resulting set, may use variables introduced in a G_i



Sets of sets

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

{10, {1, 2, 3}, 20} and is of type set[value]

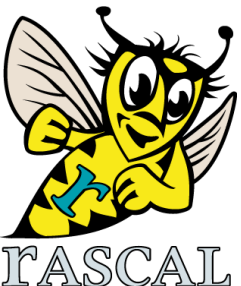


Set Splicing: *

- Set splicing is needed when inserting a set in another set without introducing an extra set level:
  - S = {1, 2, 3};
  - {10, *S, 20} gives {10, 1, 2, 3, 20}
- Set splicing can also be used for contributions to a set comprehension:
  - {*S | S <- {{1,2}, {3, 4, 5}, {6, 7}}} gives {1,2,3,4,5,6,7}

Exercises

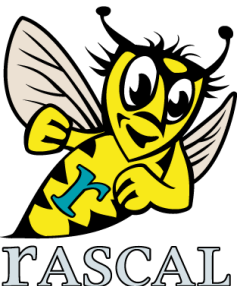
- Given is S = {1, 2, 3}; Also give type of answer
- {S, S}
- {*S, S}
- {*S, *S}
- {*M | M <- [S, S, S]}



Library Functions for Sets (~15)

Rascal/Libraries/Prelude/Set

- size
- min, max
- power
- index
- getOneFrom, takeOneFrom
- mapper

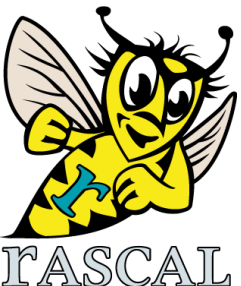


Examples

- Let S = {10, 11, 5}
- size(S) gives 3
- max(S) gives 11
- power(S) gives
  { {10,5}, {11,5}, {11},
  {10}, {5}, {11,10,5},
  {11,10}, {} }
- index(S) gives (11:2, 10:1, 5:0)
- takeOneFrom(S) gives <11, {10,5}>
  Note: an arbitrary element!

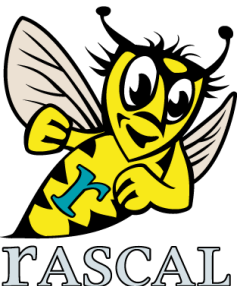
mapper

- Takes a set and a function to be applied to each set 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}



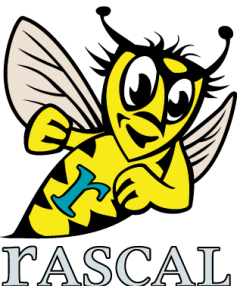
Exercises

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



Set Matching

- Given a set pattern and a set 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



Set Matching

```
rascal> S = {1, 2, 3, 4};
set[int]: {1,2,3,4}

rascal> {1, int X, 3, 4} := S;
bool: true

rascal> X;
Error: X is undefined

rascal> if({1, int X, 3, 4} := S) println("X = <X>");
X = 2
ok
```

Pattern variable

X is bound but has limited scope

Set Matching

```
rascal> S = {1, 2, 3, 4};
set[int]: {1,2,3,4}

rascal> {int X, 3, 4, 1} := S;
bool: true

rascal> {3, 4, 1,int X} := S;
bool: true

rascal> {4, 3, int X, 1} := S;
bool: true
```

Order does NOT matter, Alll these patterns are equivalent

Set Matching

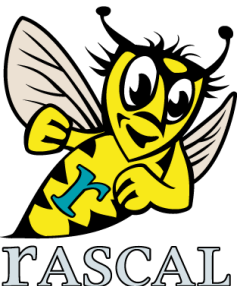
```
rascal> S = {1, 2, 3, 4};
set[int]: {1,2,3,4}

rascal> {set[int] X, 1, 2} := S;
bool: true

rascal> {*X, 1, 2} := S;
bool: true
```

Pattern variable, will be bound to {3, 4}

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



Note

- Set matching is non-unitary (>1 solutions)
- E.g., {*L, *M} := {1, 2} has four solutions:
  - L == {}, M == {1,2}
  - L == {1}, M == {2}
  - L == {2}, M == {1}
  - L == {1,2}, M == {}
- Exercise: how does this compare with

  [*L, *M] := [1, 2]

Iterate over the solutions

```
rascal>for({*L, *M} :=  {1,2,3}) println("<L>, <M>");
{3,2,1}, {}
{3,2}, {1}
{3,1}, {2}
{3}, {2,1}
{2,1}, {3}
{2}, {3,1}
{1}, {3,2}
{}, {3,2,1}
```

Compare with list matching:

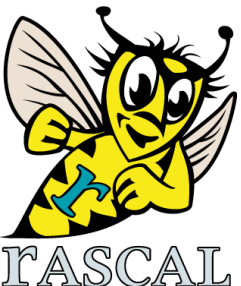
```
rascal>for([*L, *M] :=  [1,2,3]) println("<L>, <M>");
[], [1,2,3]
[1], [2,3]
[1,2], [3]
[1,2,3], []
```

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, set[int] S, 31} := {7, 8, 9, 31, 10}
- {10, *S, 31} := {7, 8, 9, 31, 10}

Applications of Sets



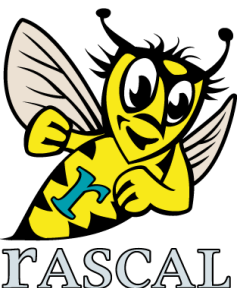
Analyzing Friends (Google Circles)

```
data Person = person(str name, set[str] friends);

public set[Person] circles = {
    person("Alan", {"Bob", "Ed", "Cath", "Dee"}),
    person("Bob", {"Dee", "Ed", "Fred", "Gina"}),
    person("Cath", {"Alan", "Fred"})
};

public void commonFriends(set[Person] persons){
    for(p1 <- persons)
       for(p2 <- persons, p1.name < p2.name){
          c = p1.friends & p2. friends;
          if(size(c) > 0)
             println("<p1.name> and <p2.name> have common friends: <c>");
       }
}
```

```
rascal>commonFriends(circles)
Bob and Cath have common friends: {"Fred"}
Alan and Bob have common friends: {"Ed","Dee"}
```



Recall earlier simplification example

```
Exp simp(add(con(n), con(m))) = con(n + m);
```

How to simplify expressions like 3 + x + 4:

add(con(3), add(var("x"), con(4)))

add(add(con(3), var("x")), con(4))

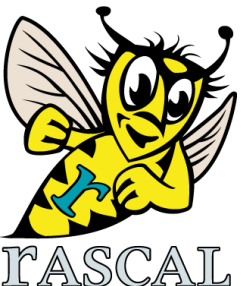
add(add(var("x"), con(3)), con(4))

add(var("x"), add(con(3), con(4)))

add(var("x"), con(7))

General case:
try all permutations of arguments of add

Set matching



A solution using sets

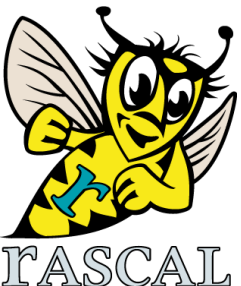
```
Exp simp(add({*e1, con(n), *e2, con(m), *e3})) = add({con(n + m), *e1, *e2, *e3});
Exp simp(add({*e1, con(0),  *e2})) = add({*e1, *e2});
```

Represent the arguments of add as a set.

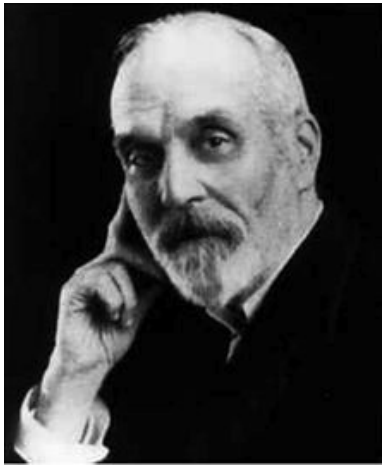
Simplifying 3 + x + 4:

add({con(3), var("x"), con(4)})

add({con(7), var("x")})



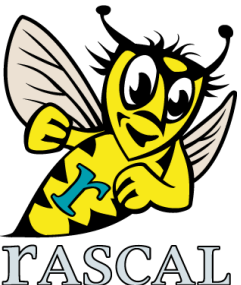
A Famous Puzzle



Henry Dudeney, 1924

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

Associate each letter with a digit so that the addition holds



A Set-based solution

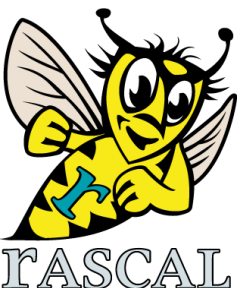
```
  S E N D
+   M O R E
= M O N E Y
```

```
public set[list[int]] sendMoreMoney(){
   ds = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

   return {[S,E,N,D,M,O,R,Y] |
            int S <- ds,
            int E <- ds - {S},
            int N <- ds - {S, E},
            int D <- ds - {S, E, N},
            int M <- ds - {S, E, N, D},
            int O <- ds - {S, E, N, D, M},
            int R <- ds - {S, E, N, D, M, O},
            int Y <- ds - {S, E, N, D, M, O, R},
            S != 0, M != 0,
                        (S * 1000 + E * 100 + N * 10 + D) +
                        (M * 1000 + O * 100 + R * 10 + E) ==
            (M * 10000 + O * 1000 + N * 100 + E * 10 + Y)};
}
```

```
rascal>sendMoreMoney()
set[list[int]]: {[9,5,6,7,1,0,8,2]}
```

1,814,400 Combinations!



Stable Marriage Problem

- Given *N* men and *N* women
- Each person has a list of preferred people of the opposite sex.
- *Marry man and woman, such that there are no two people of opposite sex that would rather marry each other than their current partner.*
- Applications:
  - match medical students with hospitals
  - Match reviewers with submitted articles

Gale/Shapley Algorithm

http://en.wikipedia.org/wiki/Stable_marriage_problem

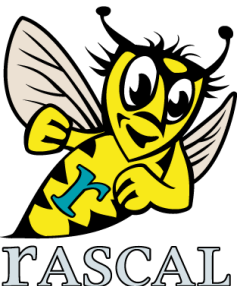
```
function stableMatching {
    Initialize all m    M and w    W to free
    while   free man m who still has a woman w to propose to {
       w = m's highest ranked such woman to whom he has not
           yet proposed
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
           (m, w) become engaged
           m' becomes free
         else
           (m', w) remain engaged
    }
}
```

Males: abe, bob, col, dan, ed, fred, gav, hal, ian, jon

Females: abi, bea, cath, dee, eve, fay, gay, hope, ivy, jan

```
map[str,list[str]] male_preferences = (
   "abe":  ["abi", "eve", "cath", "ivy", "jan", "dee", "fay", "bea", "hope", "gay"],
   "bob":  ["cath", "hope", "abi", "dee", "eve", "fay", "bea", "jan", "ivy", "gay"],
   "col":  ["hope", "eve", "abi", "dee", "bea", "fay", "ivy", "gay", "cath", "jan"],
   "dan":  ["ivy", "fay", "dee", "gay", "hope", "eve", "jan", "bea", "cath", "abi"],
   "ed":   ["jan", "dee", "bea", "cath", "fay", "eve", "abi", "ivy", "hope", "gay"],
   "fred": ["bea", "abi", "dee", "gay", "eve", "ivy", "cath", "jan", "hope", "fay"],
   "gav":  ["gay", "eve", "ivy", "bea", "cath", "abi", "dee", "hope", "jan", "fay"],
   "hal":  ["abi", "eve", "hope", "fay", "ivy", "cath", "jan", "bea", "gay", "dee"],
   "ian":  ["hope", "cath", "dee", "gay", "bea", "abi", "fay", "ivy", "jan", "eve"],
   "jon":  ["abi", "fay", "jan", "gay", "eve", "bea", "dee", "cath", "ivy", "hope"]
   );

map[str, list[str]] female_preferences = (
   "abi":  ["bob", "fred", "jon", "gav", "ian", "abe", "dan", "ed", "col", "hal"],
   "bea":  ["bob", "abe", "col", "fred", "gav", "dan", "ian", "ed", "jon", "hal"],
   "cath": ["fred", "bob", "ed", "gav", "hal", "col", "ian", "abe", "dan", "jon"],
   "dee":  ["fred", "jon", "col", "abe", "ian", "hal", "gav", "dan", "bob", "ed"],
   "eve":  ["jon", "hal", "fred", "dan", "abe", "gav", "col", "ed", "ian", "bob"],
   "fay":  ["bob", "abe", "ed", "ian", "jon", "dan", "fred", "gav", "col", "hal"],
   "gay":  ["jon", "gav", "hal", "fred", "bob", "abe", "col", "ed", "dan", "ian"],
   "hope": ["gav", "jon", "bob", "abe", "ian", "dan", "hal", "ed", "col", "fred"],
   "ivy":  ["ian", "col", "hal", "gav", "fred", "bob", "abe", "ed", "jon", "dan"],
   "jan":  ["ed", "hal", "gav", "abe", "bob", "jon", "col", "ian", "fred", "dan"]
   );
```



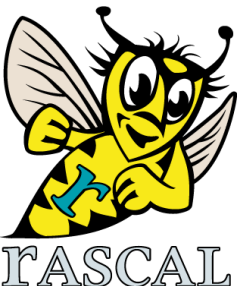
```
function stableMatching {
    Initialize all m    M and w    W to free
    while   free man m who still has a woman w to propose to {
       w = m's highest ranked such woman to whom he has not yet proposed
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
           (m, w) become engaged
           m' becomes free
         else
           (m', w) remain engaged
    }
}
```

```
data ENGAGED = engaged(str man, str woman);

public set[ENGAGED] stableMarriage(){
  engagements = {};
  freeMen = domain(male_preferences);
  while (size(freeMen) > 0){
     <m, freeMen> = takeOneFrom(freeMen);
     w = head(male_preferences[m]);
     male_preferences[m] = tail(male_preferences[m]);
     if({*_, engaged(str m1, w), *_} := engagements){
        if(indexOf(female_preferences[w], m) < indexOf(female_preferences[w], m1)){
           engagements = engagements - {engaged(m1, w)} + {engaged(m, w)};
           freeMen += m1;
         } else {
           freeMen += m;
         }
      } else {
         engagements += {engaged(m, w)};
      }
  }
  return engagements;
}
```

A Solution

```
rascal>stableMarriage()
set[ENGAGED]: {
  engaged("ed","jan"),
  engaged("ian","hope"),
  engaged("bob","cath"),
  engaged("hal","eve"),
  engaged("jon","abi"),
  engaged("gav","gay"),
  engaged("fred","bea"),
  engaged("dan","fay"),
  engaged("abe","ivy"),
  engaged("col","dee")
}
```



Summary

- Sets are an abstraction that is used in many mathematical descriptions of algorithms
- Also true for relations that we will see next
- Most of these algorithms can be easily transcribed to Rascal and be used to use directly or to experiment with