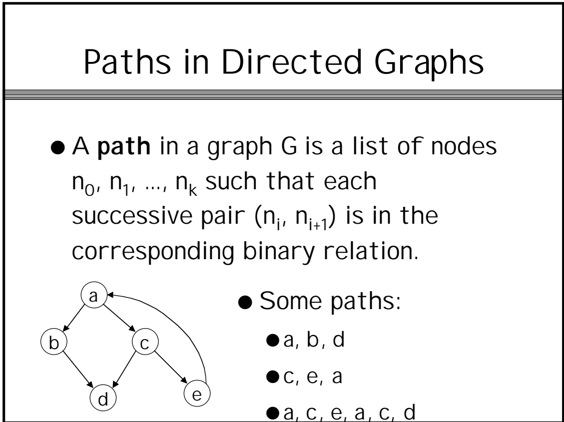
Implementing Trees as Lists

Definition of *Tree*

- There are many different varieties of trees.
- We discuss only some of them.
- Use your knowledge of these to generalize to other varieties.
- We will base our definition on ***paths*** and related concepts.

Paths in Directed Graphs

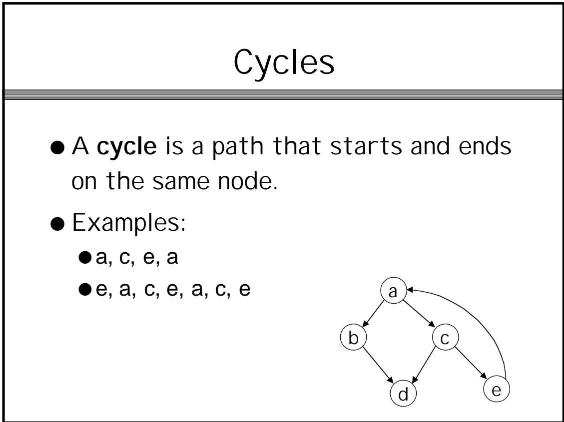
- A **path** in a graph G is a list of nodes $n_0, n_1, \ldots, n_k$ such that each successive pair (n_i, n_{i+1}) is in the corresponding binary relation.



- Some paths:
  - a, b, d
  - c, e, a
  - a, c, e, a, c, d

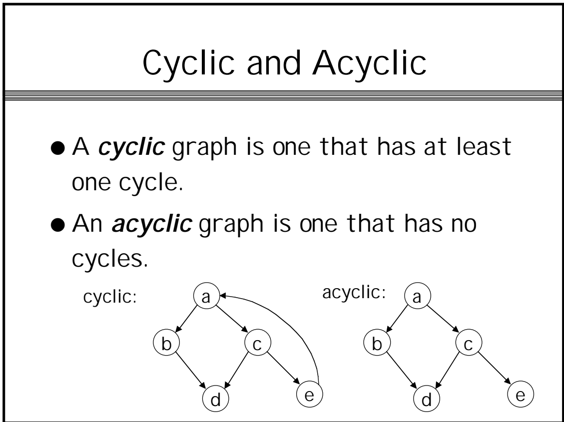
Cycles

- A **cycle** is a path that starts and ends on the same node.
- Examples:
  - a, c, e, a
  - e, a, c, e, a, c, e



Cyclic and Acyclic

- A ***cyclic*** graph is one that has at least one cycle.
- An ***acyclic*** graph is one that has no cycles.

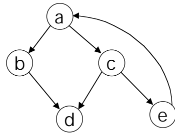


DAGs

- DAG is an acronym for "Directed Acyclic Graph"
- DAG is mainly used because it is more pronounceable than ADG ("Acyclic Directed Graph")

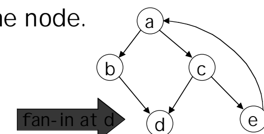
Target Set

- The ***target set*** of a node n is the set of nodes to which there is an arc from n.
  - targets(a) = {b, c}
  - targets(b) = {d}
  - targets(c) = {d, e}
  - targets(d) = { }
  - targets(e) = {a}



Fan-In

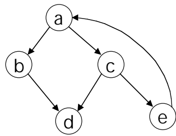
- A directed graph is said to ***fan-in*** at node n if the node is in the target sets of two different nodes.
- A directed graph "**has fan-in**" if it fans in at least one node.



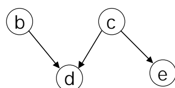
Roots

- A ***root*** of a directed graph is a node that is not in any node's target set.

no roots:



b and c are roots:

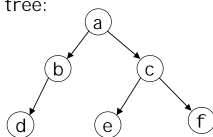


Tree at Last

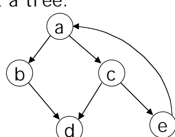
- A ***tree*** is a directed graph such that:
  - The graph is acyclic.
  - There is exactly one root.
  - It has no fan-in.

Tree vs Not

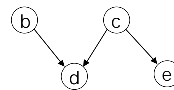
A tree:

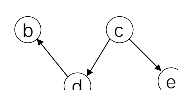


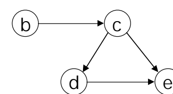
Not a tree:

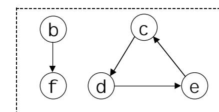


Classify these for Tree-dom

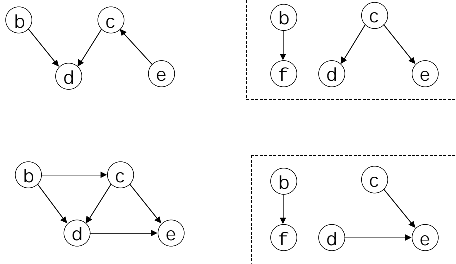






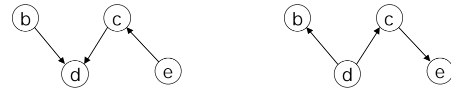


More Graphs to Classify



Reverse Graphs

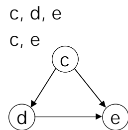
- Some graphs that may look tree-like aren't technically trees unless we consider the **reverse graph** (one with all of the arcs of the original reversed).



Reconvergence, an Alternative

- A **reconvergence** is a pair of *different* paths that start and end, respectively, on the *same* nodes.
- Therefore, a tree can also be characterized as a directed graph that
  - has one root
  - has no cycles
  - has no reconvergences

Reconvergence below:
c, d, e
c, e



Subsets of Three Properties

- **DAG**: acyclic, but
  - may have multiple roots,
  - may have fan-in
- **Forest**: acyclic, and no fan-in but
  - may have multiple roots
- A forest can also be characterized as a **collection of disjoint trees**.

Each tree could be identified with its root.

Adding/Removing Arcs

- Adding arcs to a _______ that is not a tree may make it into a tree.
- Adding arcs to a _______ that is not a tree will never make it into a tree.
- Removing arcs from a _______ that is not a tree may make it into a tree.
- Removing arcs from a _______ that is not a tree will never it into a tree.

Ordered Directed Graphs

- We use the adjective ***ordered*** to indicate that the order of targets of a node matters.
- This property is ***implicit*** with trees much of the time.
- Because we are going to represent trees by lists, we can have ordering for free if we want it.

Representing/Implementing Trees as Lists

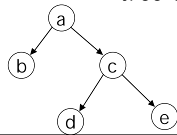
- Every tree can be represented as a list.
- Obvious:
  - Tree is a special kind of directed graph.
  - Every directed graph can be represented as a list of pairs.
- But we want a representation that makes it **clear** that we have a tree.

First Try: Target Sets

- We know that sets can be represented as lists.
- Use a list to represent the target set of each node.
- Associate each node with its list of targets.

Target-Lists Representation

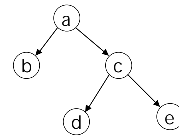
- [[a, [b, c]],
  [b, []],
  [c, [d, e]],
  [d, [ ]],
  [e, [ ]]
  ]



- This works for graphs in general;
  is not limited to trees.
- Doesn't directly show tree-dom.

Nested Target-Lists Representation

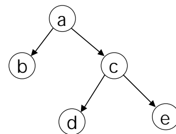
- List the root, followed by the representation of each sub-tree:



  - [a, _____, _____]
  - [a, [b], [c, ___, ____]]
  - [a, [b], [c, [d], [e]]]

Modified Nested Target-Lists Representation

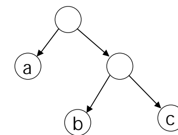
- A leaf is a node with no targets.
- When a sub-tree is a leaf, omit the brackets around it.
  - [a, _____, _____]
  - [a, b, [c, ____, ____]]
  - [a, b, [c, d, e]]



- Less-cluttered appearance (who cares?) but also less uniform.

Representation of "Unlabelled" Trees

- In this model, only *leaves* have labels.
- A leaf is represented by its label
- A non-leaf tree is represented by a list of the representations of the targets of the root.
  - [a, _____]
  - [a, [b, c]]



Representing Lists by Ordered Trees

- (This may look "backward" at first.)
- Every list can be represented as an ordered *binary* tree (tree in which each node has at most two targets).
- This corresponds to the "box" storage abstraction, where the data items may themselves be lists.

Representing Lists as Trees

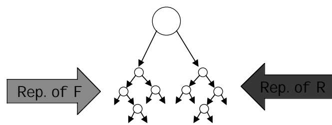
- An atomic item (non-list) is represented by itself.
- The null list is represented as a leaf [].
  A list [First | Rest] is represented by a node with two targets:
  - The **left** target is the representation of First.
  - The **right** target is the representation of Rest.
- Note that ordering of targets is essential.

Representing Lists as Trees

Atom a: a

Empty list []: []

Non-empty list [F | R]:



Representing Lists as Trees

- Matters are actually simpler if we rotate the tree 45°, so that "right" is horizontally right and "left" is down.

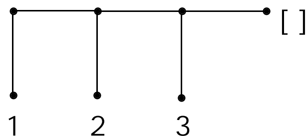
Tree representing the list

Tree representing the rest of the list

Tree representing the first element

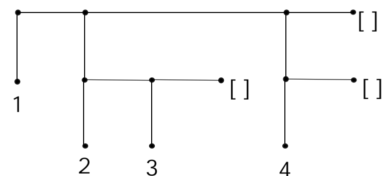
Example: Binary Tree

- Represent as a binary tree: [1, 2, 3]



Example: Binary Tree

- Represent as a binary tree:
  [1, [2, 3], [4]]



Corresponding Box Diagram

[1, [2, 3], [4]]

High-Level Functional Programming

- By ***high-level*** we mean that we are only going to construct functions by composing together (usually powerful) built-in functions.
- We place the construction of functions based on the list dichotomy, for example, under *low-level*.

Some Built-in Functions in rex

- We already saw examples:
  - **length**: returns the length of a list
  - **sort**: returns a sorted version of a list
  - **reverse**: returns the reverse of a list
  - **append**: appends together two lists
- Other functions follow

zip

- **zip** "zips together" two lists:
  - zip([3, 5, 7], [11, 13, 17]) ➪

    [3, 11, 5, 13, 7, 17]

first

- **first** returns the first element of a non-empty list:
  - first([3, 5, 7, 11, 13]) ➪ 3
  - first([[3, 5, 7], 11, 13]) ➪ [3, 5, 7]
- first([ ]) doesn't make sense; it returns an error value
- Be sure that the argument to first is not [ ].

rest

- **rest** returns a list of all but the first element of a non-empty list:
  - rest([3, 5, 7, 11, 13]) ➪ [5, 7, 11, 13]
  - rest([[3, 5, 7], 11, 13]) ➪ [11, 13]
- rest([ ]) doesn't make sense; it returns an error value
- Be sure that the argument to rest is not [ ].

cons

- **cons** creates a list from a first element and another list:
  - cons(3, [5, 7, 11, 13]) ➪ [3, 5, 7, 11, 13]
  - cons([3, 5, 7], [11, 13]) ➪ [[3, 5, 7], 11, 13]
- IMPORTANT: cons is not *append*:
  - append([3, 5, 7], [11, 13]) ➪ [3, 5, 7, 11, 13]

Type Distinction

- Suppose T is some data type
- Let T* mean the type of lists of elements of type T. Here are some **type signatures**:
- cons: $T \times T^* \rightarrow T^*$
- append: $T^* \times T^* \rightarrow T^*$
- first: $T^* \rightarrow T$
- rest: $T^* \rightarrow T^*$
- Here $\times$ means the ***pairing*** of arguments.

assoc

- **assoc** "looks up" a value in an association list.
  - If found, the entire pair is returned.
  - If not found, [ ] is returned.
- assoc("c", [["a", 3], ["b", 5], ["c", 7]]) ➪ ["c", 7]
- assoc("d", [["a", 3], ["b", 5], ["c", 7]]) ➪ [ ]

range

- **range** produces a "range" of numbers
- range(1, 10) ➪ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- There is also a 3-argument version, in which the increment can be specified:
- range(1, 4.5, 0.5) ➪ [1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5]

scale

- **scale** multiplies the values in a list by a common factor
- scale(3, [2, 4, 6, 8]) ➪ [6, 12, 18, 24]

remove_duplicates

- **remove_duplicates** returns a new list with the 2nd, 3rd, ... of any element removed
- remove_duplicates([2, 3, 4, 5, 2, 6, 5, 4]) ➪ [2, 3, 4, 5, 6]

Predicates

- A ***predicate*** is a function that returns one of two values, for purposes of discrimination among arguments.
- In rex, the two values are:
  - 1, for true
  - 0, for false
- Some built-in rex predicates follow

null Predicate

- null tests a list for being empty:
  - null([ ]) ➪ 1
  - null([1]) ➪ 0

member predicate

- **member**(X, L) tells whether or not X occurs in list L
- member(11, [5, 7, 11, 13]) ➪ 1
- member(12, [5, 7, 11, 13]) ➪ 0

even predicate

- **even**(X) tells whether or not X is evenly divisible by 2.
- even(11) ➪ 0
- even(12) ➪ 1
- Note: The argument must be an integer.

odd predicate

- **odd**(X) tells whether or not X divided by 2 has a remainder of 1.
- odd(11) ➪ 1
- odd(12) ➪ 0
- Note: The argument must be an integer.

is_prime predicate

- **is_prime**(X) tells whether or not X is prime (has any even divisors other than itself and 1)
- is_prime(11) ➪ 1
- is_prime(12) ➪ 0
- Note: The argument must be an integer.

"satisfy"

- When an argument value makes a predicate return value 1 (true), the argument is said to **satisfy** the predicate.
- This is useful in constructing sentences where the argument to the predicate is treated as active and the predicate is passive.

"satisfy" Example

- The predicate is_prime is satisfied by each of 2, 3, 5, 7, 11, ...
- It is not satisfied by 4, 6, 8, 9, 10, ...

Higher-Order Functions

- By a higher-order function, we mean one that either:
  - takes a function as an argument, or
  - returns a function as a value
- Examples follow.

map

- **map** is an extremely useful function.
- Its first argument is a function of one argument.
- Its second argument is a list of values of the same type as the argument to the first argument.
- It applies the first argument to all of the elements in the list, giving a list as the result.

map Examples

- map(odd, [2, 3, 4, 5, 6, 7, 8, 9])
  ➪ [0, 1, 0, 1, 0, 1, 0, 1]
- map(is_prime, [2, 3, 4, 5, 6, 7, 8, 9])
  ➪ [1, 1, 0, 1, 0, 1, 0, 0]
- square(X) = X*X;

  map(square, [2, 3, 4, 5, 6, 7, 8, 9])
  ➪ [4, 9, 16, 25, 36, 49, 64, 81]

In rex, we can define functions by equations this way.

Exercise

- Give a type signature for map.
- (Hint: Let T stand for the type of elements in the list.)

3-argument map

- This version of map is defined similarly, but
  - The first argument is a binary (2-argument) function;
  - The 2nd and 3rd arguments are both lists.
- The function argument is applied to pairs of corresponding elements, one from each list.

3-argument map

- map(F, [x_1, x_2, x_3, ..., x_n], [y_1, y_2, y_3, ..., y_n]) ➪ [F(x_1, y_1), F(x_2, y_2), ..., F(x_n, y_n)]
- Examples:
  - map(+, [1, 2, 3], [4, 5, 6]) ➪ [5, 7, 9]
  - map(*, [1, 2, 3], [4, 5, 6]) ➪ [4, 10, 18]
  - map(list, [1, 2, 3], [4, 5, 6]) ➪[[1, 4], [2, 5], [3, 6]]

Exercise

- Give a type signature for the 3-argument map.
- (Note: The lists don't have to have the same type of element as each other.)

keep

- **keep** has a first argument that is a predicate and a second argument that is a list.
- It returns the list of values that satisfy the first argument.
- keep(odd, [3, 4, 6, 5, 11, 12, 22, 31]) ➪ [3, 5, 11, 31]

drop

- **drop** is like *keep*, except that it returns the list of values that do not satisfy the predicate argument.
- drop(odd, [3, 4, 6, 5, 11, 12, 22, 31]) ➪ [6, 12, 22]
- is_zero(X) = X == 0;
  drop(is_zero, [4, 6, 2, 0, 1, -5, 0]) ➪ [4, 6, 2, 1, -5]

Exercise

- *keep* and *drop* both have the same type signature; what is it?

reduce

- *reduce* takes three arguments:
  - a binary operator, say b, of type $V \times V \rightarrow V$; b should be associative: b(x, b(y, z)) = b(b(x, y), z)
  - a value u of type V
  - a list L = [x_1, x_2, x_3, ..., x_n] of values of type V
- It returns a single value of type V:
  - If L is empty, then the value returned is u.
  - If L is not empty, the value is
    b(...b(b(u, x_1), x_2), x_3), ..., x_n)

Units

- If the first argument of reduce is an algebraic operator, then
- Normally the second argument is the unit for that operator.
- A unit has the property that for any X,
  b(u, X) = b(X, u) = X.
- 0 is the unit for +, 1 is the unit for *,
  [] is the unit for append.

Exercise

- What is an appropriate unit for:
  - max
  - min

reduce Examples

- reduce(+, 0, [6, 7, 8, 9]) ➪ 30
- reduce(*, 1, [6, 7, 8, 9]) ➪ 3024
- reduce(append, [], [[1, 2, 3], [4, 5], [6]])
  ➪ [1, 2, 3, 4, 5, 6]

Function Decomposition

- To **solve problems** using functions, we typically:
  - Express the problem informally as a function, with input and output.
  - Break down the function as a composition of simpler functions.
  - Repeat this process, until we are using only functions that are built-in.

Nim Game Problem

- Nim is a classic game, of which there are many variants. Here is one:
- Several piles of "tokens" are placed on the table. Players take turns removing a non-zero number from just one of the piles. The player who takes the last token wins.

Nim Example

- For brevity, we will represent the piles as a list of the number of tokens in each non-empty pile. Empty piles are not shown.
- Example:
  [3, 6, 9, 2]
  means a pile with 3 tokens, a pile with 6 tokens, one with 9, and one with 2.

Nim Play Example

- player 1: [3, 6, 9, 2] ➡ [3, 1, 7, 2]
- player 2: [3, 1, 7, 2] ➡ [3, 7, 2]
- player 1: [3, 7, 2] ➡ [3, 1, 2]
- player 2: [3, 1, 2] ➡ [1, 2]
- player 1: [1, 2] ➡ [1, 1]
- player 2: [1, 1] ➡ [1]
- player 1: [1] ➡ [] wins

Problem: Construct a good **move** function for Nim

- We want our function to be in the form of a rex definition:

  move(L) =
  some expression using simple rex functions

Strategy?

- What is an appropriate strategy?
- Consider a coarser version of the game, in which a player must take all or none of a pile.
- Then the "strategy" is clear:
  You win if you can leave an *even* number of piles.

Generalizing the Strategy

- An appropriate generalization is to use the idea of a nim_sum of the piles:
  You can win if you can always leave a nim_sum of 0.
- Moreover, if you are given a list with a non-zero nim_sum, you can *always* take from it to leave a nim_sum of 0.

Nim Play Showing nim_sum in Parens

- player 1: [3, 6, 9, 2] (14) ➡ [3, 6, 7, 2] (0)
- player 2: [3, 6, 7, 2] (0) ➡ [3, 7, 2] (6)
- player 1: [3, 7, 2] (6) ➡ [3, 1, 2] (0)
- player 2: [3, 1, 2] (0) ➡ [1, 2] (3)
- player 1: [1, 2] (3) ➡ [1, 1] (0)
- player 2: [1, 1] (0) ➡ [1] (1)
- player 1: [1] (1) ➡ [] (0) wins

The Mysterious nim_sum

- In order to explain nim_sum and why it works, we need to know how to express numbers as **binary numerals**.
- This simply means as a sum of powers of 2. For example,

$9 = 8 + 0 + 0 + 1$
$= 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$

written 1001_2.

nim_sum of two numbers

- The nim_sum of two numbers is the number corresponding to the bit-wise exclusive-OR (xor) of their binary representations.
- Example:

$13 = 1101_2$
$14 = 1110_2$
xor $0011_2 = 3$

nim_sum of a list of numbers

- The nim_sum of a list numbers the reduction of the number in the list by the xor on two numbers:

nim_sum(L) = reduce(xor, 0, L);

- It can easily be seen that:
  - xor is associative
  - 0 is the unit for xor
  - Also, for any z, xor(z, z) = 0.

Exercise: Check that these nim_sums are correct.

- [3, 6, 9, 2] (14)
- [3, 6, 7, 2] (0)
- [3, 7, 2] (6)
- [3, 1, 2] (0)
- [1, 2] (3)
- [1, 1] (0)
- [1] (1)
- [] (0)

Progress So Far

- We decomposed nim_sum into the use of reduce and xor.
- We now need to show how to compute the move.
- **Claim**: If the nim_sum of a list is not 0, it is always possible to get it to 0 by modifying just one pile.

Proof of Claim

- Suppose nim_sum(L) = s, where s != 0.
- Clearly xor(s, nim_sum(L)) = 0, by the third property of xor.
- We need to show that there is a valid move that diminishes some pile by xor'ing with s.

Example

- [3, 6, 9, 2]:

```
      0 0 1 1
      0 1 1 0
      1 0 0 1
      0 0 1 0
xor  ---------
      1 1 1 0
```

- With which pile can we xor 1110 to diminish the size of the pile?

Example

- [3, 6, 9, 2]:

```
      0 0 1 1
      0 1 1 0
      1 0 0 1
      0 0 1 0
xor  ---------
      1 1 1 0
```

- The only pile that qualifies is 1001. xor(1001, 1110) = 0111.
- So we want to change pile 9 to 7 (by taking 2).

Example

- [3, 6, 7, 2]:

```
      0 0 1 1
      0 1 1 0
      0 1 1 1
      0 0 1 0
xor  ---------
      0 0 0 0
```

Generalization

- If the nim_sum is non-zero then there is **always** a pile to which the nim_sum can be xor'ed to achieve a smaller pile.
- **Rationale**: The highest-order bit in the nim_sum must have come from one of the piles (it is not the result of a "carry", for example). When we xor the nim_sum to that very pile, this bit becomes 0, meaning that the resulting pile is necessarily smaller.

Completing the move function

- Compute the nim_sum s of the list of piles.
- Assume for now that s is not 0.
- Find a pile p such that xor(p, s) < p, (where < means *numerically* less than) and replace it with xor(p, s).
- Remove any empty pile.

Technical Issues

- To remove any empty piles, we can use drop:
  - isZero(p) = p == 0;
  - removeEmpty(L) = drop(isZero, L);

Technical Issues

- The second function alluded to defies analysis using any function we have seen so far. Let's call it ***magic***:
  magic(L, s) = return a new list similar to L, except that the ***first*** element, p, where xor(p, s) < p, is ***replaced*** with xor(p, s).
- There are various ways to implement magic, which we will show later.

Illustration of *magic*

- nim_sum([3, 6, 9, 2]) = 14
  $3 = 0\,0\,1\,1_2$;
  $6 = 0\,1\,1\,0_2$;
  $9 = 1\,0\,0\,1_2$;
  $2 = 0\,0\,1\,0_2$;
  $14 = 1\,1\,1\,0_2$
- xor(9, 14) = 7 < 9
- So magic([3, 6, 9, 2], 14) = [3, 6, 7, 2]
- nim_sum([3, 6, 7, 2]) = 0

nim *move* function decomposed

- move(L) =
  removeEmpty(magic(L, nim_sum(L)));
- nim_sum(L) =
  reduce(xor, 0, L);
- removeEmpty(L)=
  drop(isZero, L);

Alternate version (one big expression)

- move(L) =
  drop(isZero,
  magic(L,
  reduce(xor, 0, L)));
- The prior, decomposed, version may be preferable for its **documentation value**.
- (We still need to deal later with the case that the nim_sum is 0.)

Anonymous Functions

- Sometimes it may be regarded as inconvenient to **name** functions such as isZero.
- Another problem arises when we want to **fix** one or more arguments to a function, leaving the remainder to vary.
- Both are solved by *anonymous* functions.

Anonymous Functions

- Functions have a meaning independent of the names we give them.
- We want a way to use a function without giving it a name.
- Notation:
  (X) => ... some expression ...
  means "the function that, with argument X, returns the value of ... some expression ....

Example

- The function isZero, defined by:

  isZero(X) = X == 0;

  can also be written anonymously:

  (X) => X == 0

  read "the function that, with argument X, returns the value of X == 0".

Precedent

- This notation for talking about a function goes back to (at least) Bourbaki, where the symbol

  ↦

  was used instead of

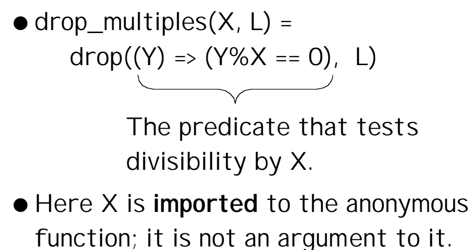
  =>

- Church used the idea extensively, but with a different symbol λ as a *prefix*.

More Anonymous Functions

- (X) => X+5 The function that adds 5
- (X) => X*5 The function that multiplies by 5
- (X) => X*X The function that squares
- (X, Y) => Y/X The function that divides its second argument by its first.

Anonymous Functions with "Imported" Values



- drop_multiples(X, L) =

  drop((Y) => (Y%X == 0), L)

  The predicate that tests divisibility by X.

- Here X is **imported** to the anonymous function; it is not an argument to it.

Exercises

- Give an equation defining *scale* using *map*, where

  scale(F, L) multiplies each element of L by a factor F.

Exercises

- Give an equation defining pairWith, such that

  pairWith(X, L) creates a list in which each element of L is paired with X:

  pairWith(3, [1, 2, 3])

  ➪ [[3, 1], [3, 2], [3, 3]]

Exercises

- Can you give an equation defining pairs, such that
  pairs(L, M) creates a list in which each element of L is paired with each element of M, e.g.
  pairs([1, 2, 3], [4, 5, 6])
  ➪ [[1, 4], [1, 5], [1, 6],
  [2, 4], [2, 5], [2, 6],
  [3, 4], [3, 5], [3, 6]]

find predicate

- find(P, L) returns the longest suffix of L that begins with an element satisfying P.
- Example:
  - find(odd, [2, 4, 6, 7, 9, 10, 12])
    ➪ [7, 9, 10, 12]
- As with map, etc., find is often used with anonymous functions.

An Implementation of magic

- To find the longest suffix of L beginning with a p such that xor(p, s) < p:
  - find(((p)=>xor(p, s)< p), L), for example
  - find(((p)=>xor(p, 14) < p), [3, 6, 9, 2]) ➪ [9, 2]
- Call this suffix M for now.
- We use the first(M) as the value of p.
- We use rest(M) to construct the new list.
- The new list can be formed as
  append(N, cons(xor(first(M), s), rest(M)))
  where N is an appropriate-length prefix of L:
  prefix(length(L)-length(M), L)

An Implementation of magic

- magic(L, s) =
  M = find(((p)=>xor(p, s)<p), L),
  N = prefix(length(L)-length(M), L),
  append(N, cons(xor(first(M), s), rest(M)));

inner definitions, called "equational guards" in book

- rex allows us to use these inner definitions in exactly this form. But if we wanted to, we could just substitute them (at the expense of making the expression really big):
  magic(L, s) =
  append(prefix(length(L)-length(find(((p)=>xor(p, s)<p), L)), L),
  cons(xor(first(find(((p)=>xor(p, s)<p), L)), s),
  rest(find(((p)=>xor(p, s)<p), L))));
- This expression might also be costlier to compute, because the same sub-expressions occur multiple times.

Other versions of magic

- We will show another, perhaps clearer, version in discussing low-level functional programming.
- We also need to show how implement xor, and will do so there as well.

What if the nim_sum is 0?

- In this case, there is no winning strategy; the best that the move function can do is "fake it"; make a move that appears intelligent, without giving too much away.
- For example, it could just take one token from the first pile arbitrarily.
- We can achieve this overall effect with a **conditional expression** as follows:
- move(L) =
  s = reduce(xor, 0, L),
  s == 0 ? removeEmpty(cons(first(L)-1, rest(L)))
  : removeEmpty(magic(L, s));

conditional expression

- removeEmpty(L) = drop(isZero, L);