Equality

We now return to the question of equality: what does it mean when we say two things are equal?

Equality

We now return to the question of equality: what does it mean when we say two things are equal?

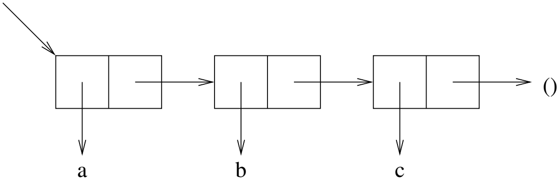
We will have to approach this carefully, starting with the way datastructures are stored in memory

Equality

Lists in Memory

We often draw pairs (also called *cons cells*) as blocks:

`(a b c)` is

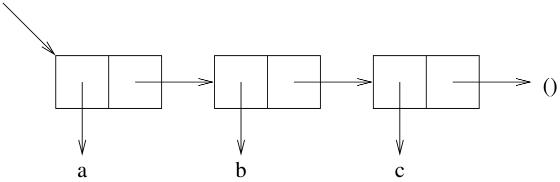


Equality

Lists in Memory

We often draw pairs (also called *cons cells*) as blocks:

`(a b c)` is



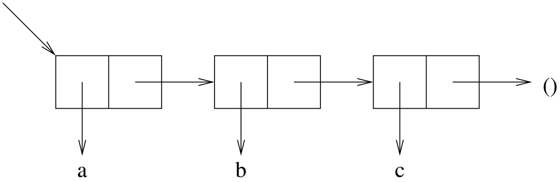
Each cons is a pair of memory locations

Equality

Lists in Memory

We often draw pairs (also called *cons cells*) as blocks:

`(a b c)` is



Each cons is a pair of memory locations

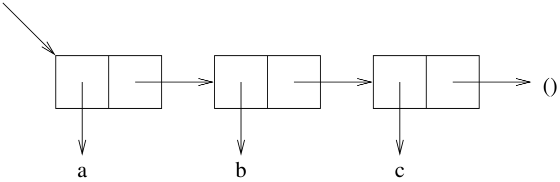
The `car` is a pointer to (i.e., is the memory address of) where the symbol `a` is stored in memory, etc.

Equality

Lists in Memory

We often draw pairs (also called *cons cells*) as blocks:

`(a b c)` is



Each cons is a pair of memory locations

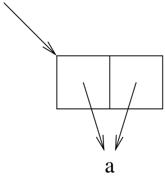
The `car` is a pointer to (i.e., is the memory address of) where the symbol `a` is stored in memory, etc.

A pair really is a pair of pointers in memory

Equality

Lists in Memory

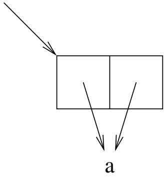
The pair made by `(cons 'a 'a)`, namely `(a . a)` is



Equality

Lists in Memory

The pair made by `(cons 'a 'a)`, namely `(a . a)` is

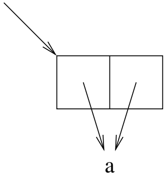


Both parts point to the symbol `a`

Equality

Lists in Memory

The pair made by `(cons 'a 'a)`, namely `(a . a)` is



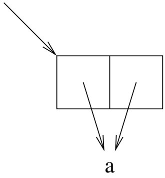
Both parts point to the symbol `a`

In Lisp, a symbol of a given name is unique within the system, unlike strings: there may be several copies of `"cat"` in memory

Equality

Lists in Memory

The pair made by `(cons 'a 'a)`, namely `(a . a)` is



Both parts point to the symbol `a`

In Lisp, a symbol of a given name is unique within the system, unlike strings: there may be several copies of `"cat"` in memory

Uniqueness of symbols is pretty much a defining property of symbols in Lisp

Equality

Aside

Another subtle point:

```
(let ((x 1))
  (let ((x 2))
    ... x ...)
  ... x ...)
```

Regarded as variables (code), the two `x`s are different, and they refer to different memory locations

Equality

Aside

Another subtle point:

```
(let ((x 1))
  (let ((x 2))
    ... x ...)
  ... x ...)
```

Regarded as variables (code), the two `x`s are different, and they refer to different memory locations

You could uniformly replace the inner `x` with, say, `y`, and (name clashes aside) the code does the same thing

Equality

Aside

Another subtle point:

```
(let ((x 1))
  (let ((x 2))
    ... x ...)
  ... x ...)
```

Regarded as variables (code), the two `x`s are different, and they refer to different memory locations

You could uniformly replace the inner `x` with, say, `y`, and (name clashes aside) the code does the same thing

But regarded as symbols (data), there is just one `x`

Equality

Aside

Another subtle point:

```
(let ((x 1))
  (let ((x 2))
    ... x ...)
  ... x ...)
```

Regarded as variables (code), the two `x`s are different, and they refer to different memory locations

You could uniformly replace the inner `x` with, say, `y`, and (name clashes aside) the code does the same thing

But regarded as symbols (data), there is just one `x`

It's a matter of which properties you are thinking about

Equality

Lists in Memory

The locations of the cons pairs can be anywhere in memory

Equality

Lists in Memory

The locations of the cons pairs can be anywhere in memory

The successive pairs in a list need not be next to each other in memory and quite likely are not

Equality

Lists in Memory

The locations of the cons pairs can be anywhere in memory

The successive pairs in a list need not be next to each other in memory and quite likely are not

In `(1 2 3)`, i.e., `(1 . (2 . (3 . ())))` the cons cell `(1 . ...)` has no particular placement in memory relative to the cons cell `(2 . ...)`

Equality

The locations of the cons pairs can be anywhere in memory

The successive pairs in a list need not be next to each other in memory and quite likely are not

In `(1 2 3)`, i.e., `(1 . (2 . (3 . ())))` the cons cell `(1 . ...)` has no particular placement in memory relative to the cons cell `(2 . ...)`

Some implementations may even have the `car` and `cdr` parts in entirely separate areas of memory

Equality

Lists in Memory

The locations of the cons pairs can be anywhere in memory

The successive pairs in a list need not be next to each other in memory and quite likely are not

In `(1 2 3)`, i.e., `(1 . (2 . (3 . ())))` the cons cell `(1 . ...)` has no particular placement in memory relative to the cons cell `(2 . ...)`

Some implementations may even have the `car` and `cdr` parts in entirely separate areas of memory

It doesn't really matter and the Lisp system deals with it: you never see memory locations in Lisp (unless. . .)

Equality

Lists in Memory

Each call to the function `cons` will return a *newly allocated* pair that is somewhere in memory, but nowhere in particular

Each call to the function `cons` will return a *newly allocated* pair that is somewhere in memory, but nowhere in particular

This is one of the defining properties of `cons`: it guarantees always to allocate a *new* pair

Equality

Lists in Memory

Each call to the function `cons` will return a *newly allocated* pair that is somewhere in memory, but nowhere in particular

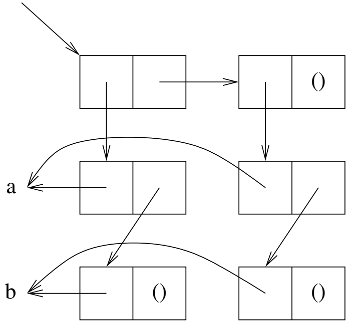
This is one of the defining properties of `cons`: it guarantees always to allocate a *new* pair

And, as a consequence, `list` guarantees to create an all-new list

Equality

Lists in Memory

The list made by `(list (list 'a 'b) (list 'a 'b))` is `((a b) (a b))`

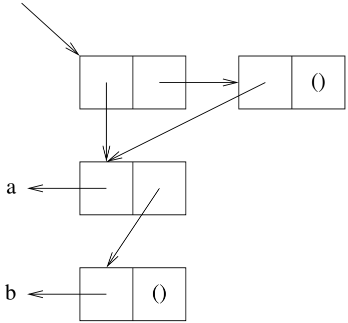


For convenience, I have drawn pointers to `()` as ()

Contrast with the list made in

```
(let ((lab (list 'a 'b)))
   (list lab lab))
```

$\rightarrow$ `((a b) (a b))`

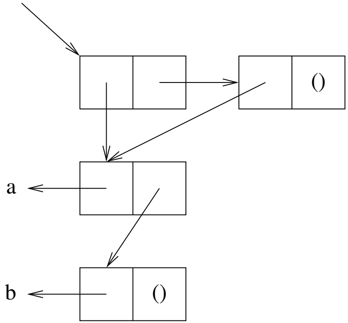


Equality

Lists in Memory

Contrast with the list made in

```
(let ((lab (list 'a 'b)))
   (list lab lab))           → ((a b) (a b))
```



Very different from the previous picture!

Equality

Lists in Memory

Understanding the implications of what is going on here is one of the important things in Computer Science

Equality

Lists in Memory

Understanding the implications of what is going on here is one of the important things in Computer Science

Both lists print as `((a b) (a b))` but their structures are very different

Equality

Lists in Memory

Understanding the implications of what is going on here is one of the important things in Computer Science

Both lists print as `((a b) (a b))` but their structures are very different

In the second, the sublists are *shared*: the second sublist is the *same* memory as the first sublist

Equality

Lists in Memory

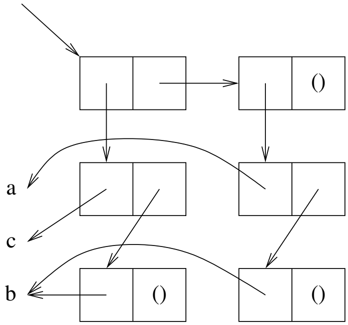
Understanding the implications of what is going on here is one of the important things in Computer Science

Both lists print as `((a b) (a b))` but their structures are very different

In the second, the sublists are *shared*: the second sublist is the *same* memory as the first sublist

In the first, the sublists are separate: the second sublist occupies *different* memory from the first sublist

If we take the first example and somehow update the first sublist to have a `c` instead of the `a` we get

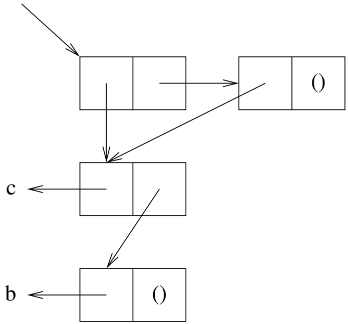


or `((c b) (a b))`

Equality

Lists in Memory

If we take the second example and somehow update the first sublist to have a `c` instead of the `a` we get



or `((c b) (c b))`

Equality

Lists in Memory

As the second sublist **is** the first sublist, updating the first updates them both

Equality

Lists in Memory

As the second sublist **is** the first sublist, updating the first updates them both

But it's not "both" as there's only one

Equality

Lists in Memory

Sometimes we want shared structures: it uses less memory

Equality

Lists in Memory

Sometimes we want shared structures: it uses less memory

Sometimes we want non-shared structures: we can manipulate parts independently

Equality

Lists in Memory

Sometimes we want shared structures: it uses less memory

Sometimes we want non-shared structures: we can manipulate parts independently

Both are useful

Equality

Lists in Memory

Sometimes we want shared structures: it uses less memory

Sometimes we want non-shared structures: we can manipulate parts independently

Both are useful

But we must be aware which we are getting

Equality

Lists in Memory

Sometimes we want shared structures: it uses less memory

Sometimes we want non-shared structures: we can manipulate parts independently

Both are useful

But we must be aware which we are getting

And this applies to all such structures in all languages, not just Lisp

Equality

Lists in Memory

The thing to remember is that `cons` (and therefore `list` and `append` and similar) always allocates new pairs from memory

Equality

Lists in Memory

The thing to remember is that `cons` (and therefore `list` and `append` and similar) always allocates new pairs from memory

So the first example guarantees separate sublists

The thing to remember is that `cons` (and therefore `list` and `append` and similar) always allocates new pairs from memory

So the first example guarantees separate sublists

The second, with `(list lab lab)` we are being explicit about sharing the list `lab`

Exercise. Draw boxes and arrows to explain the differences between

- `(list '(a b) '(c d))`
- `(cons '(a b) '(c d))`
- `(append '(a b) '(c d))`

Each function here makes new cons cells: they *do not modify existing cons cells*

Also: the results from `append` shares the second argument, but makes a new copy of the first argument (Exercise: why?). This makes `append` a very expensive operation if the first argument is a long list

Equality

Though different in memory, the two variants of `((a b) (a b))` are the “same” in some sense

Equality

Though different in memory, the two variants of `((a b) (a b))` are the "same" in some sense

They certainly print the same

Equality

Though different in memory, the two variants of `((a b) (a b))` are the "same" in some sense

They certainly print the same

Sometimes we want to say they are the same, sometimes not

Equality

Though different in memory, the two variants of `((a b) (a b))` are the “same” in some sense

They certainly print the same

Sometimes we want to say they are the same, sometimes not

So Lisp provides two (and more) tests of equality of objects

Equality

Though different in memory, the two variants of `((a b) (a b))` are the "same" in some sense

They certainly print the same

Sometimes we want to say they are the same, sometimes not

So Lisp provides two (and more) tests of equality of objects

It is rare that other languages are even aware of this issue, leading to all kinds of bugs from programmers using them

Equality

The question is: what do we mean by equality?

Equality

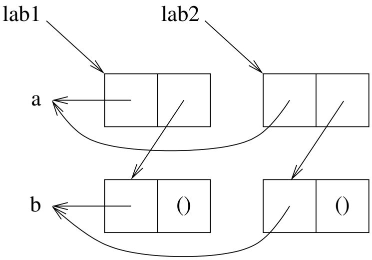
The question is: what do we mean by equality?

Suppose `lab1` and `lab2` have the values created by separate calls `(list 'a 'b)`

```
(let ((lab1 (list 'a 'b))
      (lab2 (list 'a 'b)))
   ...)
```

Equality

So the two lists occupy different chunks of memory



Equality

One kind of equality, *structural equality*, says things are "equal" if they "look the same"

Equality

One kind of equality, *structural equality*, says things are “equal” if they “look the same”

This can be made precise

Equality

One kind of equality, *structural equality*, says things are "equal" if they "look the same"

This can be made precise

In Lisp there is a function named `equal` for this type of equality

Equality

One kind of equality, *structural equality*, says things are “equal” if they “look the same”

This can be made precise

In Lisp there is a function named `equal` for this type of equality

`(equal lab1 lab2)` $\rightarrow$ `t`
`(equal lab1 lab1)` $\rightarrow$ `t`

Equality

Another type of equality is “these two objects are the same object”

Equality

Another type of equality is “these two objects are the same object”

They are the same thing in memory

Equality

Another type of equality is “these two objects are the same object”

They are the same thing in memory

In Lisp there is a function named `eq` for this type of equality

Equality

Another type of equality is “these two objects are the same object”

They are the same thing in memory

In Lisp there is a function named `eq` for this type of equality

```
(eq lab1 lab2) → ()
(eq lab1 lab1) → t
```

Equality

Another type of equality is “these two objects are the same object”

They are the same thing in memory

In Lisp there is a function named `eq` for this type of equality

`(eq lab1 lab2)` $\rightarrow$ `()`
`(eq lab1 lab1)` $\rightarrow$ `t`

All objects are `eq` to themselves (except in Common Lisp…)

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`
3. if they are both numbers, return `t` if they are numerically equal (and same type) else `()`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`
3. if they are both numbers, return `t` if they are numerically equal (and same type) else `()`
4. if they are both strings, return `t` if they contain the same characters else `()`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`
3. if they are both numbers, return `t` if they are numerically equal (and same type) else `()`
4. if they are both strings, return `t` if they contain the same characters else `()`
5. similarly for other datatypes

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`
3. if they are both numbers, return `t` if they are numerically equal (and same type) else `()`
4. if they are both strings, return `t` if they contain the same characters else `()`
5. similarly for other datatypes
6. if they are both pairs, return `t` if both their `car`s are `equal` and their `cdr`s are `equal`

Equality

The `equal` test is roughly as follows. Given two objects `a` and `b`

1. if `(eq a b)` return `t`
2. if they are both symbols, return `t` if they are the same symbol else `()`
3. if they are both numbers, return `t` if they are numerically equal (and same type) else `()`
4. if they are both strings, return `t` if they contain the same characters else `()`
5. similarly for other datatypes
6. if they are both pairs, return `t` if both their `car`s are `equal` and their `cdr`s are `equal`
7. Else return `()`

Equality

In brief, two pairs are `equal` if they are the same pair (`eq`), or both

- their `cars` are `equal`
- and their `cdrs` are `equal`

Equality

`equal` is naturally recursive

Equality

`equal` is naturally recursive

`eq` is a fast memory pointer comparison

Equality

`equal` is naturally recursive

`eq` is a fast memory pointer comparison

`equal` can take a long time on large datastructures

Equality

`equal` is naturally recursive

`eq` is a fast memory pointer comparison

`equal` can take a long time on large datastructures

`eq` is like == in C

Equality

`equal` is naturally recursive

`eq` is a fast memory pointer comparison

`equal` can take a long time on large datastructures

`eq` is like == in C

Whenever you need it, you have to code your own `equal` on datastructures in C

Equality

`equal` is naturally recursive

`eq` is a fast memory pointer comparison

`equal` can take a long time on large datastructures

`eq` is like == in C

Whenever you need it, you have to code your own `equal` on datastructures in C

Though `strcmp` is provided for strings

Equality

And there's more types of equality, mostly for numerical testing

Equality

And there's more types of equality, mostly for numerical testing

- `(eq 1 1.0)` $\rightarrow$ `()` as expected

Equality

And there's more types of equality, mostly for numerical testing

- `(eq 1 1.0)` $\rightarrow$ `()` as expected
- `(equal 1 1.0)` $\rightarrow$ `()` as they are different types

Equality

And there's more types of equality, mostly for numerical testing

- `(eq 1 1.0)` $\rightarrow$ `()` as expected
- `(equal 1 1.0)` $\rightarrow$ `()` as they are different types
- `(= 1 1.0)` $\rightarrow$ `t` for mathematically the same

Equality

And there's more types of equality, mostly for numerical testing

- `(eq 1 1.0)` $\rightarrow$ `()` as expected
- `(equal 1 1.0)` $\rightarrow$ `()` as they are different types
- `(= 1 1.0)` $\rightarrow$ `t` for mathematically the same
- `(eql 1 1.0)` $\rightarrow$ `()` another equality introduced by Common Lisp to fix a feature when they found some implementations couldn't guarantee that `(eq 1 1)` should be `t`. So, `(eql 1 1)` is guaranteed true, but possibly slower than `eq`. Use `eql` on numbers and characters in Common Lisp

Equality

And there's more types of equality, mostly for numerical testing

- `(eq 1 1.0)` → `()` as expected
- `(equal 1 1.0)` → `()` as they are different types
- `(= 1 1.0)` → `t` for mathematically the same
- `(eql 1 1.0)` → `()` another equality introduced by Common Lisp to fix a feature when they found some implementations couldn't guarantee that `(eq 1 1)` should be `t`. So, `(eql 1 1)` is guaranteed true, but possibly slower than `eq`. Use `eql` on numbers and characters in Common Lisp

Some implementations had `(eq 1023 1023)` true but `(eq 1024 1024)` false

Equality

Note: in the examples above we used

```
(let ((lab1 (list 'a 'b)
      (lab2 (list 'a 'b))) ...)
```

rather than

```
(let ((lab1 '(a b))
      (lab2 '(a b))) ...)
```

Equality

Note: in the examples above we used

```
(let ((lab1 (list 'a 'b)
      (lab2 (list 'a 'b))) ...)
```

rather than

```
(let ((lab1 '(a b))
      (lab2 '(a b))) ...)
```

This is because the lists `'(a b)` are constant and the Lisp interpreter or compiler might spot they are the same and only allocate once and share it

Equality

Note: in the examples above we used

```
(let ((lab1 (list 'a 'b)
      (lab2 (list 'a 'b))) ...)
```

rather than

```
(let ((lab1 '(a b))
      (lab2 '(a b))) ...)
```

This is because the lists `'(a b)` are constant and the Lisp interpreter or compiler might spot they are the same and only allocate once and share it

So `(eq '(a b) '(a b))` could be either `t` or `()`

Equality

Exercise. What might you get from

```
(eq 'cat 'cat)
```

```
(eq "cat" "cat")
```

Exercise. Try

```
(eq '(a b) '(a b))
```

on a few Lisps

Equality

Aside: Equality in Mathematics

In various branches of Mathematics you have to define your own equality

Equality

Aside: Equality in Mathematics

In various branches of Mathematics you have to define your own equality

For example, in arithmetic, we have the standard $1 + 1 = 2$ kind of equality

Equality

Aside: Equality in Mathematics

In various branches of Mathematics you have to define your own equality

For example, in arithmetic, we have the standard $1 + 1 = 2$ kind of equality

For others, we need to define what we want to be an equality and *prove* it has the properties we expect from an equality

Equality

Aside: Equality in Mathematics

In various branches of Mathematics you have to define your own equality

For example, in arithmetic, we have the standard $1 + 1 = 2$ kind of equality

For others, we need to define what we want to be an equality and *prove* it has the properties we expect from an equality

We generally want:

- X equals X; reflexivity
- if X equals Y then Y equals X; symmetry
- if X equals Y and Y equals Z then X equals Z; transitivity

Equality

Aside: Equality in Mathematics

We also want

- if M equals N and $X[M/N]$ is what we get when we replace all occurrences of M by N in X, then X equals $X[M/N]$

Equality

Aside: Equality in Mathematics

We also want

- if M equals N and $X[M/N]$ is what we get when we replace all occurrences of M by N in X, then X equals $X[M/N]$

This is substitionality, or substitution of equals by equals

We also want

- if M equals N and $X[M/N]$ is what we get when we replace all occurrences of M by N in X, then X equals $X[M/N]$

This is substitionality, or substitution of equals by equals

In arithmetic, we can replace $1 + 1$ by 2 wherever we see it in an expression, and not affect the value of the expression

Equality

Aside: Equality in Mathematics

In other areas, e.g., Lambda calculus, a model of computation, $1 + 1$ and 2 are different, as it takes a step of computation to get from one to the other

Equality

Aside: Equality in Mathematics

In other areas, e.g., Lambda calculus, a model of computation, $1 + 1$ and 2 are different, as it takes a step of computation to get from one to the other

They say "$1 + 1$ reduces to 2", but maintain they are not "equal"

Equality

Aside: Equality in Mathematics

In other areas, e.g., Lambda calculus, a model of computation, $1 + 1$ and 2 are different, as it takes a step of computation to get from one to the other

They say "$1 + 1$ reduces to 2", but maintain they are not "equal"

In summary: "equal" is a tricky and subtle concept

Equality

Aside: Equality in Mathematics

In other areas, e.g., Lambda calculus, a model of computation, $1 + 1$ and 2 are different, as it takes a step of computation to get from one to the other

They say “$1 + 1$ reduces to 2”, but maintain they are not “equal”

In summary: “equal” is a tricky and subtle concept

Exercise. Convince yourself that Lisp `equal` is an equality is the above sense (reflexive, symmetric, transitive, substitutional)

Exercise. And what about

`(eq l (cons (car l) (cdr l)))`

and

`(equal l (cons (car l) (cdr l)))`

for a list `l`?

Exercise. A related concept is *shallow copy* vs. *deep copy*. Read about this

Equality

Skip to the end…

The following was not covered in lectures

It is not examinable, but is worth reading nevertheless!

Equality

Recursion

Lisp is at its most powerful when we think recursively

Lisp is at its most powerful when we think recursively

```
(defun factorial (n)
  (if (< n 2)
      n
      (* n (factorial (- n 1)))))
```

What happens when we try

```
(defun loop (n)
   (print n)
   (loop (+ n 1)))
```

Equality

Recursion

What happens when we try

```
(defun loop (n)
   (print n)
   (loop (+ n 1)))
```

This should loop “forever”

Equality

Recursion

What happens when we try

```
(defun loop (n)
  (print n)
  (loop (+ n 1)))
```

This should loop “forever”

In many languages and compilers this would loop for a while and then crash

What happens when we try

```
(defun loop (n)
  (print n)
  (loop (+ n 1)))
```

This should loop “forever”

In many languages and compilers this would loop for a while and then crash

This is because each function call takes some stack space and the machine eventually runs out of memory

Equality

Tail Recursion

Except Lisp, of course

Equality
Tail Recursion

Except Lisp, of course

Or, rather, the good Lisps

Except Lisp, of course

Or, rather, the good Lisps

A good Lisp notices that you do not need to save the current function invocation on the stack as you never need to come back

Equality

Tail Recursion

Except Lisp, of course

Or, rather, the good Lisps

A good Lisp notices that you do not need to save the current function invocation on the stack as you never need to come back

So it replaces the function call by a simple jump back to the start of the function `loop`

Equality

Tail Recursion

Except Lisp, of course

Or, rather, the good Lisps

A good Lisp notices that you do not need to save the current function invocation on the stack as you never need to come back

So it replaces the function call by a simple jump back to the start of the function `loop`

```
(defun loop (n)
   (print n)
   increment n
   goto top
```

Equality

Tail Recursion

This can, and does, run forever

Equality

Tail Recursion

This can, and does, run forever

The act of replacing a function call by a jump is called introducing a *tail call*, and by extension we have *tail recursion*

Equality

Tail Recursion

This can, and does, run forever

The act of replacing a function call by a jump is called introducing a *tail call*, and by extension we have *tail recursion*

This is done at the end (tail) of a function when the compiler can deduce you don't ever need to return to that function

Equality

Tail Recursion

This can, and does, run forever

The act of replacing a function call by a jump is called introducing a *tail call*, and by extension we have *tail recursion*

This is done at the end (tail) of a function when the compiler can deduce you don't ever need to return to that function

Good compilers can spot tail calls and do this optimisation

Equality

Tail Recursion

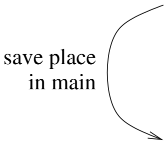
```
...
(foo a)
...
```

```
(defun foo (n)
  ...
  (bar n))
```

```
(defun bar (m)
  ...)
```

Equality

Tail Recursion



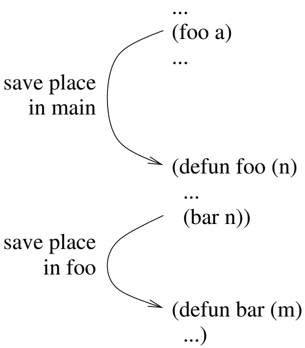

```
...
(foo a)
...

(defun foo (n)
  ...
  (bar n))

(defun bar (m)
  ...)
```

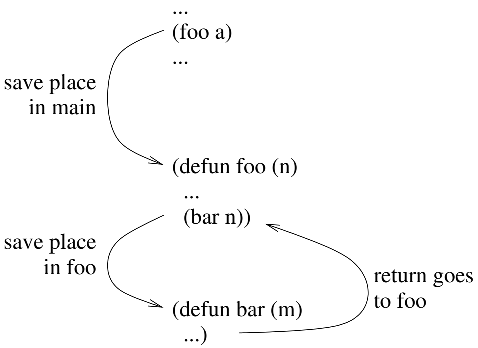
Equality

Tail Recursion



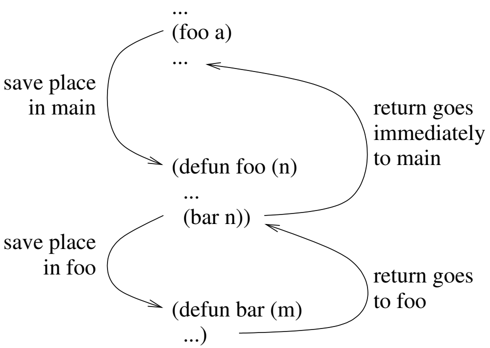
Equality

Tail Recursion



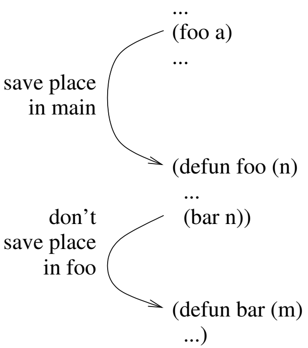
Equality

Tail Recursion



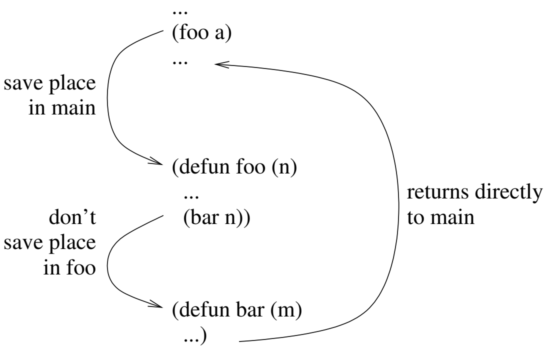
Equality

Tail Recursion



Equality

Tail Recursion



Equality

Tail Recursion

This simple observation allows us to have arbitrary loops, but to write them naturally recursively

Equality

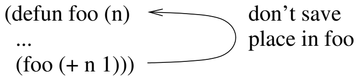
Tail Recursion

This simple observation allows us to have arbitrary loops, but to write them naturally recursively

```
(defun foo (n)
  ...
  (foo (+ n 1)))
```

don't save
place in foo

This simple observation allows us to have arbitrary loops, but to write them naturally recursively



The compiler does clever stuff behind our backs, but compilers are always doing that

```
(defun loop2 (n)
  (loop2 (+ n 1))
  (print n))
```

is not tail recursive, as we need to save where we are before the recursive call to `loop2` to come back and do the `print`

Equality

Tail Recursion

```
(defun loop2 (n)
  (loop2 (+ n 1))
  (print n))
```

is not tail recursive, as we need to save where we are before the recursive call to `loop2` to come back and do the `print`

The call to `loop2` is not in the tail position

Equality

Tail Recursion

```
(defun loop2 (n)
  (loop2 (+ n 1))
  (print n))
```

is not tail recursive, as we need to save where we are before the recursive call to `loop2` to come back and do the `print`

The call to `loop2` is not in the tail position

In reality, we don't actually come back ever

```
(defun loop2 (n)
  (loop2 (+ n 1))
  (print n))
```

is not tail recursive, as we need to save where we are before the recursive call to `loop2` to come back and do the `print`

The call to `loop2` is not in the tail position

In reality, we don’t actually come back ever

This one would run until it ran out of stack space unless we have a really clever compiler

Equality

Tail Recursion

```
(defun foo (n)
  (print n)
  (bar (+ n 1)))

(defun bar (m)
  (print m)
  (foo (+ m 1)))
```

are mutually tail recursive: the compiler can replace the function call to `bar` by a jump to `bar`; similarly the other way round

Equality

Tail Recursion

```
(defun foo (n)
  (print n)
  (bar (+ n 1)))

(defun bar (m)
  (print m)
  (foo (+ m 1)))
```

are mutually tail recursive: the compiler can replace the function call to `bar` by a jump to `bar`; similarly the other way round

This, too, will run forever (if the compiler spots it)

Equality

Tail Recursion

Tail recursion is supported in some modern C compilers

Equality

Tail Recursion

Tail recursion is supported in some modern C compilers

It's been in most Lisps since the 1960s

Tail recursion is supported in some modern C compilers

It's been in most Lisps since the 1960s

Admittedly, the functional nature makes it easier to analyse and spot tail recursion in Lisp than in a procedural language like C

Equality

Tail Recursion

Tail recursion is supported in some modern C compilers

It's been in most Lisps since the 1960s

Admittedly, the functional nature makes it easier to analyse and spot tail recursion in Lisp than in a procedural language like C

EuLisp (Euscheme): yes. Clisp: interpreted, no; compiled, yes. Scheme: always, since defined in the language specification, Clojure: no, allegedly because Java doesn't but this is not a valid implication

Loops in other languages are replaced by tail recursive calls in the functional style

```
for (i = 0; i < 10; i++) {
  do something
}
```

becomes

```
(defun loopy (i)
  (when (< i 10)
        do something
        (loopy (+ i 1))))

(loopy 0)
```

Or even

```
(labels ((loopy (i)
            (when (< i 10)
              do something
              (loopy (+ i 1)))))
  (loopy 0))
```

Or even

```
(labels ((loopy (i)
            (when (< i 10)
              do something
              (loopy (+ i 1)))))
  (loopy 0))
```

This looks clumsy, but we are trying to force a procedural style (iteration) in Lisp

Or even

```
(labels ((loopy (i)
            (when (< i 10)
              do something
              (loopy (+ i 1)))))
  (loopy 0))
```

This looks clumsy, but we are trying to force a procedural style (iteration) in Lisp

There are better ways to do something to a sequence of objects

Or even

```
(labels ((loopy (i)
           (when (< i 10)
             do something
             (loopy (+ i 1)))))
  (loopy 0))
```

This looks clumsy, but we are trying to force a procedural style (iteration) in Lisp

There are better ways to do something to a sequence of objects

But note within the body of the function `loopy` the variable `i` is never updated; the variable `i` does not vary

It would be quite easy to add a "`for`" form to Lisp (and some Lisps do) that implements

```
(for init test inc body1 body2 ...)
```

as

```
(labels ((loopy ()
             (when test
               body1 body2 ...
               inc
               (loopy))))
  init
  (loopy))
```

But that's not a route we shall follow

Equality

Tail Recursion

Many functions can, with some effort, be converted to tail recursive style

Many functions can, with some effort, be converted to tail recursive style

```
(defun factorial (n)
  (if (< n 2) 1 (* n (factorial (- n 1)))))
```

is not tail recursive

```
(defun fact (n)
  (factaux n 1))

(defun factaux (n sofar)
  (if (< n 2)
      sofar
      (factaux (- n 1) (* n sofar))))
```

is tail recursive

Equality

Tail Recursion

```
(defun fact (n)
  (factaux n 1))

(defun factaux (n sofar)
  (if (< n 2)
      sofar
      (factaux (- n 1) (* n sofar))))
```

is tail recursive

Whether it is worth it is a question you must address in each circumstance

Note: tail recursion is something done by the compiler, but the programmer should be aware it exists to make good use of it

Note: tail recursion is something done by the compiler, but the programmer should be aware it exists to make good use of it

Also, it can make debugging a little harder: the backtrace at an error will not contain the record of the intermediate functions that were tail optimised

Equality

Tail Recursion

Note: tail recursion is something done by the compiler, but the programmer should be aware it exists to make good use of it

Also, it can make debugging a little harder: the backtrace at an error will not contain the record of the intermediate functions that were tail optimised

The loss is worth the gain, though