Lisp

Skip past closure example

Lisp

Closures

Closures are very useful

```
(defun make-account ()
   (let ((balance 0))
      (list
         (lambda () balance)
         (lambda (n) (setq balance (+ balance n))))))

(let ((acct (make-account)))
  (setq current (car acct))
  (setq deposit (cadr acct)))
```

Adapted from Abelson & Sussman; they use `setq` to destructively update a variable, so this is not pure functional style code

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

`(current)` → `0`

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
```

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
(current) → 10
```

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
(current) → 10
```

The closures `current` and `deposit` have captured the (shared) environment `balance`

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
(current) → 10
```

The closures `current` and `deposit` have captured the (shared) environment `balance`

And the environment is mutable

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
(current) → 10
```

The closures `current` and `deposit` have captured the (shared) environment `balance`

And the environment is mutable

Note that `current` is not referentially transparent

Lisp

Closures

Now we have an account object (`balance`) which we can only manipulate using the accessor closures `current` and `deposit`

```
(current) → 0
(deposit 10) → 10
(current) → 10
```

The closures `current` and `deposit` have captured the (shared) environment `balance`

And the environment is mutable

Note that `current` is not referentially transparent

This is to emphasise that closures and the functional style are separate concepts (though often used together)

Lisp

Closures

And separate accounts have separate balances

```
(let ((acct (make-account)))
  (setq current2 (car acct))
  (setq deposit2 (cadr acct)))
```

```
(deposit2 100) → 100
(current) → 10
(current2) → 100
```

Lisp

Closures

And separate accounts have separate balances

```
(let ((acct (make-account)))
  (setq current2 (car acct))
  (setq deposit2 (cadr acct)))
```

```
(deposit2 100) → 100
(current) → 10
(current2) → 100
```

The closure concept predates object orientation

Lisp

Closures

And separate accounts have separate balances

```
(let ((acct (make-account)))
  (setq current2 (car acct))
  (setq deposit2 (cadr acct)))
```

```
(deposit2 100) → 100
(current) → 10
(current2) → 100
```

The closure concept predates object orientation

See “Structure and Interpretation of Computer Programs” for more on this

Lisp

Closures

Closures are powerful and can be used for all kinds of things

Lisp

Closures

Closures are powerful and can be used for all kinds of things

They are used for capturing information (like objects in OO languages): the example above

Lisp

Closures

Closures are powerful and can be used for all kinds of things

They are used for capturing information (like objects in OO languages): the example above

They can be used for data hiding: the example above

Lisp

Closures

Closures are powerful and can be used for all kinds of things

They are used for capturing information (like objects in OO languages): the example above

They can be used for data hiding: the example above

They can be used to *delay* evaluation: make a closure at some point, then only execute the code later in the knowledge that the code will be executed in the environment of *creation*

Lisp

Closures

```
(let* ((n 0)
       (m 1)
       (lazy (lambda () (foo n m))))
  ...

  (lazy)    ; now call foo
  ...
)
```

Lisp

Closures

Exercise. Look up *thunks* and `delay` and `force` in Scheme

Lisp

Closures

Exercise. Look up *thunks* and `delay` and `force` in Scheme

In the context of functional style programming, closures “do the right thing”

Lisp

Closures

Exercise. Look up *thunks* and `delay` and `force` in Scheme

In the context of functional style programming, closures “do the right thing”

They capture elements of computation

Lisp

Closures

In MacOS X *Snow Leopard* closures are used as a device to structure parallelism

Lisp

Closures

In MacOS X *Snow Leopard* closures are used as a device to structure parallelism

It takes C and extends it with a new construct:

```
int n;
...
x = ^(int m){ printf("n is %d m is %d\n", n, m); };
...
x(4);
```

makes the value of x a closure

Lisp

Closures

In MacOS X *Snow Leopard* closures are used as a device to structure parallelism

It takes C and extends it with a new construct:

```
int n;
...
x = ^(int m){ printf("n is %d m is %d\n", n, m); };
...
x(4);
```

makes the value of `x` a closure

Closures can then be scheduled to run in parallel

General Remark

You have encountered several different kinds of “executable” object:

- function
- closure
- method
- (generic function)

General Remark

You have encountered several different kinds of “executable” object:

- function
- closure
- method
- (generic function)

Make sure you understand the difference between these

General Remark

You have encountered several different kinds of “executable” object:

- function
- closure
- method
- (generic function)

Make sure you understand the difference between these

They are all different kinds of things

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops
- No global state. **Globals variables are bad news in all styles of programming**

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops
- No global state. **Globals variables are bad news in all styles of programming**
- Thus only use local variables

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops
- No global state. **Globals variables are bad news in all styles of programming**
- Thus only use local variables
- No modification of variables, in particular no use of `setq`, `set`, `setf`, `set!`, `push` and so on

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops
- No global state. **Globals variables are bad news in all styles of programming**
- Thus only use local variables
- No modification of variables, in particular no use of `setq`, `set`, `setf`, `set!`, `push` and so on
- Variables don't vary; referential transparency

Functional Style

Perhaps this is a good point to reflect on the functional style: it is characterised by:

- Use of recursion: operate on a small part, do the rest by recursion
- Thus no iterative loops
- No global state. **Globals variables are bad news in all styles of programming**
- Thus only use local variables
- No modification of variables, in particular no use of `setq`, `set`, `setf`, `set!`, `push` and so on
- Variables don't vary; referential transparency
- Use binding of local variables

Functional Style

- No modification of datastructures

Functional Style

- No modification of datastructures
- Make a new one if you want something changed

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency
- Use higher order functions

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency
- Use higher order functions
- Use mapping

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency
- Use higher order functions
- Use mapping
- Separate the traversal of a datastructure from the operations on it

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency
- Use higher order functions
- Use mapping
- Separate the traversal of a datastructure from the operations on it
- Think of datastructures as a whole

Functional Style

- No modification of datastructures
- Make a new one if you want something changed
- This allows safe sharing of datastructures; referential transparency
- Use higher order functions
- Use mapping
- Separate the traversal of a datastructure from the operations on it
- Think of datastructures as a whole
- Keep state in closures

Functional Style

Keeping to the functional style is hard if you have been trained in the procedural style

Functional Style

Keeping to the functional style is hard if you have been trained in the procedural style

But the benefits in the long run are well worth the effort

Functional Style

Keeping to the functional style is hard if you have been trained in the procedural style

But the benefits in the long run are well worth the effort

Even in non-functional languages

Functional Style

Keeping to the functional style is hard if you have been trained in the procedural style

But the benefits in the long run are well worth the effort

Even in non-functional languages

The training in the way you think improves your coding in the procedural and object-oriented styles

Lisp

Garbage Collection

Lisp makes it very easy to make a lots of lists: we need to be a little careful in what we do

Lisp

Garbage Collection

Lisp makes it very easy to make a lots of lists: we need to be a little careful in what we do

In evaluating `(let ((x (list ’a ’b))) 42)` we create a new list `(a b)`, then discard it

Lisp

Garbage Collection

Lisp makes it very easy to make a lots of lists: we need to be a little careful in what we do

In evaluating `(let ((x (list 'a 'b))) 42)` we create a new list `(a b)`, then discard it

As the binding of `x` disappears when we exit the `let`, the list is no longer accessible by our program

Lisp

Garbage Collection

Lisp makes it very easy to make a lots of lists: we need to be a little careful in what we do

In evaluating `(let ((x (list 'a 'b))) 42)` we create a new list `(a b)`, then discard it

As the binding of `x` disappears when we exit the `let`, the list is no longer accessible by our program

The cons cells (pairs) are now garbage occupying memory to no purpose

Lisp

Garbage Collection

We have seen in Java there is a similar problem: objects are often allocated and then dropped (sometimes intentionally)

```
{ foo x = new foo();
  ...
  x = y;
  ...
}
```

This bad code — don’t do this

Or C

```
{ char *x = (char*)malloc(10);
  ...
  x = y;
  ...
}
```

This bad code — don't do this

Lisp

Garbage Collection

The locations of the objects are no longer known by the program: the program can no longer refer to them: they are garbage

Lisp

Garbage Collection

The locations of the objects are no longer known by the program: the program can no longer refer to them: they are garbage

So Lisp pioneered automatic *garbage collection*

Lisp

Garbage Collection

The locations of the objects are no longer known by the program: the program can no longer refer to them: they are garbage

So Lisp pioneered automatic *garbage collection*

This means you can cons without regard to memory use: but you should also be aware there is the associated cost

Lisp

Garbage Collection

The locations of the objects are no longer known by the program: the program can no longer refer to them: they are garbage

So Lisp pioneered automatic *garbage collection*

This means you can cons without regard to memory use: but you should also be aware there is the associated cost

Memory management is never free