Closures and metaprogramming

- It can be incredibly useful to write functions that will build and customize other functions for us
- We pass parameters to the writer function, telling it precisely what customizations we want for the to-be-created function
- It builds and returns the function, and we run it as desired
- Writing code that writes code is called *metaprogramming*
- Functions that write customized versions of other functions are often referred to as *closures*

Example scenario: NPC control

- Suppose we have an AI-controlled character in a game
- When the character is generated, it might be given a variety of different skills, traits, equipment, etc
- Suppose we have a function that is run periodically to decide what that character will do next
- This could be a generic function, with many nested if/switch statements saying “if the character has this then pick from the following...”
- Or, when we create the character, we could have our code generate a custom control function just for that one

Example scenario: page display

- Suppose our app has many configurable options related to the display of a web page or other data
- A user typically doesn’t change their own settings too frequently
- To display a page or data we could use a general purpose function that considers all the possible combinations of settings
- Or, whenever the user adjusts their settings we could generate a special streamlined function just for that combination of settings

Starting small...

- Dealing with examples like the previous two can be pretty complex, and we don’t want our code generating buggy code that will run later
- We’ll start with small examples, slowly work up from there
- Basic process will be:
  - Write a function that takes a parameter or two and uses those plus lambda to create and return some new function
  - Call our writer, store the returned function in a variable, and test it using funcall or apply

Function using a name in a msg

- Ultimately, the user wants a function they pass a message to, e.g. prt(msg), but that prints their name and message
- We’ll call the builder function, passing it the user’s name, and the builder will return the appropriate print function

```
(defvar prt (buildPrt “Super Me”))
(funcall prt “ wants to go home”)


Super Me wants to go home
```

Buildprt implementation

```
(defun buildPrt (username)
   (lambda (msg) (format t “~A ~A~%” username msg)))

(defvar prt (buildPrt “Super Me”))
(defvar prt2 (buildPrt “Someone Else”))

(funcall prt “wants to go home”)
Super Me wants to go home
(funcall prt2 “does not want to go home”)
Someone Else does not want to go home
```

Error checks in builder and lambda

- The builder function needs to error check the parameters it is passed, but it also needs to insert appropriate error checking into the function it is building (for that function's parameters)
- Example: building a time dilation calculator
  - our friend travels on a spacecraft travelling at a fixed speed
  - we want a function built for us to computes time passed for us vs for them
  - when we call that function we'll tell it how much time has passed for us, it will return how much time has passed for them
- In order to generate such a function, the builder needs to know the speed of the spacecraft, so will require that as a

Using our timeBuilder

```
; we know bob's spacecraft will be flying 11km/second
(defvar bobTime (timeBuilder 11))
; we feel like it took 1000 days, how long was it for bob?
(funcall bobTime 1000)
```

- Error checking in timeBuilder: check that the speed passed was a positive real number
- Error checking in the constructed function: check that the amount of time passed was a positive real number

timeBuilder

```
(defun timeBuilder (speed)
  (if (and (realp speed) (> speed 0))
     ; valid speed, generate the lambda function
     (lambda (time)
        (if (and (realp time) (> time 0))
           (* time (sqrt (- 1 (/ (* sp sp) 90000000000))))
           (format t “Error: invalid time ~A~%” time)))
     ; invalid speed, return lambda function that prts error
     (lambda (time)
       (format t “Error: invalid speed ~A~%” speed))))
```

Error handling

- Bad speed value in construction:

```
(defvar f (buildTime “foo”))
(funcall f  1000)
Error: invalid speed foo
```

- OK construction, bad time:

```
(defvar f (buildTime 11))
(funcall f “foo”)
Error: invalid time foo
```