RACKET

Anonymous Procedures
Call-by-Value
Data and Function abstraction
Functions as arguments
Accumulators
Deep Recursion

Good things to know about Racket

- An empty list must be quoted: '()
- To concatenate two lists, use append:

```
(append '(1 2 3) '(a b c)) -> (1 2 3 a b c)
```

- A quick and efficient way to tell whether something is a list:

```
(pair? <arg>)
```

- **`pair?`** Tells whether the object was (or could have been) constructed by cons
- `(list? <arg>)` tells whether something is a proper list.
- As such, it must traverse the entire list.
- To tell whether something is not a pair?:

```
(atom? <arg>)
```

Anonymous Procedures

- Lambda creates a procedure.
- Example of an anonymous procedure:

  (lambda (x) (+ x 1))
- Example of applying an anonymous procedure:

  ((lambda (x) (+ x 1)) 4)

Call-by-Value

- In Racket, arguments are called-by-value.
- This means that in a procedure call, all variables are evaluated, before we call the procedure.
- Example:

```
(define a 3)
(define b 4)
(+ a b)
```

- We know that "+" is a variable bound to the procedure +.
- We see that "a" has the value 3 and "b" has 4.
- We now issue the following procedure call:

  (<procedure +> 3 4)

Call-by-Value

- Notice that we CANNOT do the following (successfully):

```
(define swap
  (lambda (x y)
    (set! temp x)
    (set! x y)
    (set! y temp)))
```

- P.S. Until further notice, you are not allowed to use “set!”

Data Abstraction

- Abstraction is a powerful tool for writing mean and lean code.
- Consider:

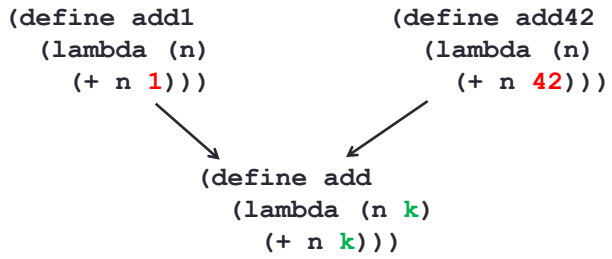
```
(define add1
  (lambda (n)
    (+ n 1)))
```

```
(define add42
  (lambda (n)
    (+ n 42)))
```

- Rather than cluttering up your code with procedures that are just about the same, abstract from the differences and produce one procedure.

Data Abstraction

- Replace the item that is different by a variable and add it as a parameter:



Function Abstraction

- What if two procedures differ not by data but by a function call.
- Example:

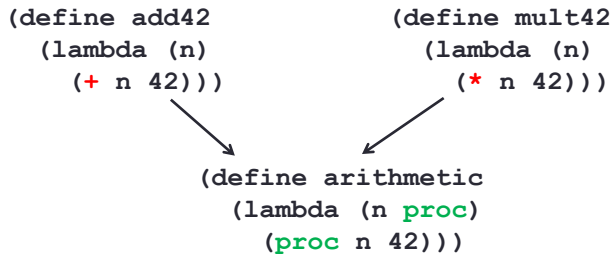
```
(define add42
  (lambda (n)
    (+ n 42)))
```

```
(define mult42
  (lambda (n)
    (* n 42)))
```

- Same process as for data abstraction.

Function Abstraction

- Replace the item that is different by a variable and add it as a parameter:



Procedures as Arguments

- Notice that in Racket, we can (easily) pass a procedure as an argument.
- In Java, we had to include the procedure in a class and then pass in an instance of the class. We called that a functor.
- In Racket, passing in a procedure is seamless.
- Examples:

```
(arithmetic 7 +)
(arithmetic 7 *)
```

- Notice that pass-by-value plays a major role in this.

Procedures as Arguments

- Recall anonymous procedures.
- Here is an example of passing an anonymous procedure as an argument:

  ((lambda (x) (x 3)) (lambda (y) (+ y 1))

Accumulators

- An accumulator is a parameter that can be used to accumulate a result.
- They typically increase the performance of recursive procedures.
- Simple example:

```
(define factorial
  (lambda (n accu)
     (if (= n 0)
          accu
          (factorial (- n 1) (* n accu)))))
```

- Class exercise: Write fibonacci, using accumulators.

Deep Recursion

- So far, we focused on simple or top-level lists.
- List can be nested and they can be nested deeply.
- Consider:

```
'(((a b (c) d) (e (f)) g) h)
```

- Let's write a procedure which counts the number of elements in this nested list.
- Class exercise:
  - Member for nested lists
  - Write a procedure called "flatten" which flattens a nested list.