CSC324 Lecture 14

15 July 2020

ICFP 2020 programming contest opens today!

The International Conference on Functional Programming holds an annual programming contest, with remarkably interesting problems to solve.

ICFP Contest 2009

University of Kansas

Control a satellite to move between specified orbits and rendezvous with other satellites

ICFP Contest 2014

University of Oxford & Well-Typed LLP

Write AI programs for a Pac-Man-like game in SECD machine instructions and 8-bit machine assembly

ICFP Contest 2016

University of Electro-Communications

Write an AI to solve abstract origami

ICFP Contest 2006

Carnegie Mellon University

Implement a virtual machine that runs a provided OS and crack it using unconventional programming languages

ICFP Contest 2018

Rochester Institute of Technology

Generate nanobot traces to construct, destruct, and reconstruct target 3D objects

ICFP 2020 programming contest opens today!

The International Conference on Functional Programming holds an annual programming contest, with remarkably interesting problems to solve.

(The subreddit is good for reading teams' solutions and writeups)

`https://www.reddit.com/r/icfpcontest/`

`https://icfpcontest2020.github.io/#/`

Today:

- We will discuss a limitation of our current implementation of -<
- We will finish our implementation
- We will introduce the ?- operator and **backtracking search**

Note about defining -< expressions in the editor

Everything behaves correctly in the REPL... But what happens if we put the amb expression inside the editor window (so it's part of lecture13.rkt) and then call next?

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (-< 1 2 3)
1
> (next!)
2
> (next!)
3
> (next!)
'done
> 
```

Note about defining -< expressions in the editor

Calls to (next!) are printing values out two times. WHAT IS GOING ON?

```
(-< 1 2 3)
```

```
Language: racket, with debugging; memory limit: 256 MB.
1
> (next!)
2
2
> (next!)
3
3
> '(what the HECK)
'(what the HECK)
>
```

Note about defining -< expressions in the editor

Intuition: when we run (-< 1 2 3) in the editor, it's actually part of a larger expression that evaluates the call to amb and prints it in the repl!

```
(-< 1 2 3)
```

```
Language: racket, with debugging; memory limit: 256 MB.
1
> (next!)
2
2
> (next!)
3
3
> '(what the HECK)
'(what the HECK)
>
```

```
; what we call
(-< 1 2 3)
```

```
; what we call
(-< 1 2 3)

; what it expands to
(shift k
       (begin (add-choice! (thunk (k (-< 2 3))))
              (k 1)))
```

```racket
; what we call
(-< 1 2 3)

; what it expands to
(shift k
       (begin (add-choice! (thunk (k (-< 2 3))))
              (k 1)))

; what is happening in DrRacket
(print-in-repl (-< 1 2 3))
```

```racket
; what we call
(-< 1 2 3)

; what it expands to
(shift k
       (begin (add-choice! (thunk (k (-< 2 3))))
              (k 1)))

; what is happening in DrRacket
(print-in-repl (-< 1 2 3))

; What is evaluated in DrRacket
(print-in-repl (shift k
                      (begin
                        (add-choice! (thunk (k (-< 2 3))))
                        (k 1))))
```

```racket
; what we call
(-< 1 2 3)

; what it expands to
(shift k
       (begin (add-choice! (thunk (k (-< 2 3))))
              (k 1)))

; what is happening in DrRacket
(print-in-repl (-< 1 2 3))

; What is evaluated in DrRacket
(print-in-repl (shift k
                 (begin
                   (add-choice! (thunk (k (-< 2 3))))
                   (k 1))))

; The shift in -< captures the call
; to print-in-repl in the k continuation!
```

```racket
; Drop anything outside "our code" with a reset...
(reset (-< 1 2 3))
```

Language: racket, with debugging; memory limit: 256 MB.

```
1
> (next!)
2
> (next!)
3
>
```

Last time, we got this far...

```racket
#lang racket
(require racket/control)

(define choices (void))
(define (set-choices! val) (set! choices val))

(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; "when there's only candidate, there's only one choice"
     (begin (set-choices! (void))
            expr)]

    ; Multiple choices: return the first one and store the amb
    ; that produces all the others in choices.
    [(-< expr1 expr2 ...)
     (shift k (begin (set-choices! (thunk (k (-< expr2 ...))))
                     (k expr1)))]))

(define (next!)
  (if (void? choices)
      'done
      (reset (choices))))
```

Remember the goal:

We would like to have multiple amb operators in play, where all the choice points yield the **cartesian products** of the amb's choices.

```
> (list (-< 1 2) (-< 'a 'b))
'(1 a)
> (next!)
'(1 b)
> (next!)
'(2 a)
> (next!)
'(2 b)
> (next!)
'done
```

Why doesn't it work for us?

choices:
(void)

```
(list (-< 1 2) (-< 'a 'b))
```

As we haven't evaluated an amb yet, our choices variable is set to the void function.

First, let's identify the redex in this expression. What gets evaluated first?

Why doesn't it work for us?

choices:

```
(void)
```

```
(list (-< 1 2) (-< 'a 'b))
```

OK, so the continuation is…?

Why doesn't it work for us?

```
choices:
(void)
```

```
(list (-< 1 2) (-< 'a 'b))
```

Now let's evaluate the redex.

Why doesn't it work for us?

```
choices:
(void)
```

```
(list
 (shift k
  (begin
    (set-choices! (thunk (k (-< 2))))
    (k 1)))
 (-< 'a 'b))
```

Now we evaluate the two statements in the begin clause...

Why doesn't it work for us?

```
(list
  (shift k
    (begin
      (set-choices! (thunk (k (-< 2))))
      (k 1)))
  (-< 'a 'b))
```

choices:

```
(thunk k (-< 2))
```

k:

```
(λ (x)
  (list x (-< 'a 'b))
```

Now we evaluate the two statements in the begin clause...

Why doesn't it work for us?

choices:

```
(thunk k (-< 2))
```

```
(list
 1
 (-< 'a 'b))
```

Now we have to evaluate the second argument to list, which is also an amb.

Why doesn't it work for us?

```
(list
 1
 (-< 'a 'b))
```

choices:

```
(thunk k (-< 2))
```

Now we have to evaluate the second argument to list, which is also an amb.

Why doesn't it work for us?

```
(list
  1
  (shift k
    (begin
      (set-choices! (thunk (k (-< 'b))))
      (k 'a))))
```

```
choices:
(thunk k (-< 2))
```

Now we have to evaluate the second argument to list, which is also an amb.

Why doesn't it work for us?

```
(list
  1
  (shift k
    (begin
      (set-choices! (thunk (k (-< 'b))))
      (k 'a))))
```

choices:

`(thunk k (-< 'b))`

k:

`(λ (x) (list 1 x))`

Now we have to evaluate the second argument to list, which is also an amb.

Why doesn't it work for us?

choices:

```
(thunk k (-< 'b))
```

```
(list
 1
 'a)
```

Why doesn't it work for us?

```
(list
 1
 'a)
```

choices:

```
(thunk k (-< 'b))
```

what happened to (thunk k (-< 2)), though?

We overwrote it when we mutated choices again! *MUTATION!!!*

Second cut of -<

- Allow choices to store any number of thunks: choices becomes a list

- When we evaluate a -<, we cons the thunk onto the list
- When we get a choice, we pop a thunk off the list.

```
(define choices (box '()))
(define (add-choice! val) (set-box! choices (cons val (unbox choices))))

(define (get-choice!) (let* ([ub (unbox choices)]
                             [val (car ub)])
                        (begin (set-box! choices (cdr ub))
                               val)))

(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; "when there's only candidate, there's only one choice"
     expr]

    ; Multiple choices: return the first one and store the amb
    ; that produces all the others in choices.
    [(-< expr1 expr2 ...)
     (shift k (begin (add-choice! (thunk (k (-< expr2 ...))))
                     (k expr1)))]))

(define (next!)
  (if (empty? (unbox choices)) 'done
      (reset ((get-choice!)))))
```

Why does it work this time?

```
choices:
'()
```

```
(list (-< 1 2) (-< 'a 'b))
```

Let's say we evaluate the the first amb,

Why does it work this time?

choices:

```
(list (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

```
(list 1 (-< 'a 'b))
```

And now it's time to evaluate the second.

Why does it work this time?

choices:

```
(list (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

```
(list 1 (-< 'a 'b))
```

And now it's time to evaluate the second.

Why does it work this time?

```
(list
 1
 (shift k
   (begin
     (add-choice! (thunk (k (-< 'b))))
     (k 'a))))
```

choices:

```
(list (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

```
(list
 1
 (shift k
   (begin
     (add-choice! (thunk (k (-< 'b))))
     (k 'a))))
```

choices:

```
(list (thunk (k2 (-< 'b)))
      (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b)))
```

k2:

```
(λ (x) (list 1 x))
```

```
(list 1 'a)
```

choices:

```
(list (thunk (k2 (-< 'b)))
      (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b)))
```

k2:

```
(λ (x) (list 1 x))
```

Now, when we call `(next!)`, we will pop the thunk at top of the choices stack and evaluate it.

```
(next!)
```

choices:

```
(list (thunk (k2 (-< 'b))
      (thunk (k1 (-< 2)))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k2:

```
(λ (x) (list 1 x)
```

Now, when we call `(next!)`, we will pop the thunk at top of the choices stack and evaluate it.

choices:

```
(list (thunk (k1 (-< 2)))
```

```
(next!)
=> ((thunk (k2 (-< 'b))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k2:

```
(λ (x) (list 1 x)
```

Now, when we call `(next!)`, we will pop the thunk at top of the choices stack and evaluate it.

choices:

```
(list (thunk (k1 (-< 2)))
```

```
(next!)
=> ((thunk (k2 (-< 'b))))
=> (list 1 (-< 'b))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k2:

```
(λ (x) (list 1 x)
```

Now, when we call `(next!)`, we will pop the thunk at top of the choices stack and evaluate it.

choices:

```
(list (thunk (k1 (-< 2)))
```

```
(next!)
=> ((thunk (k2 (-< 'b))))
=> (list 1 (-< 'b))
=> (list 1 'b)
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k2:

```
(λ (x) (list 1 x)
```

```
=> (list 1 'b)
```

choices:

```
(list (thunk (k1 (-< 2)))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k1's body contains an amb expression!
Calling k1 "plants the seeds" to generate
'a and 'b ambiguously all over again!

choices:

```
(list (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

```
=> (list 1 'b)
```

OK, so what happens when we call (next!) yet again?

```
(next!)
```

choices:

```
(list (thunk (k1 (-< 2))))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b)))
```

choices:

'()

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

```
(next!)
=> ((thunk (k1 (-< 2))))
```

choices:

```
'()
```

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

Remember that evaluating an amb with only one choice doesn't push anything onto the stack ("when there's only one candidate, …")

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
=> (k1 2)
```

choices:

```
'()
```

k1:

```
(λ (x)
  (list x (-< 'a 'b)))
```

choices:

```
'()
```

k1:

```
(λ (x)
  (list x (-< 'a 'b)))
```

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
=> (k1 2)
=> (list 2 (-< 'a 'b))
```

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
=> (k1 2)
=> (list 2 (-< 'a 'b))
```

choices:

```
'()
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

Here, we're going to add a new choice point! We understand by now that we are going to: reify the current continuation… (which is what?)

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
=> (k1 2)
=> (list 2 (-< 'a 'b))
```

choices:

```
'()
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k3:

```
(λ (x) (list 2 x)
```

And now we'll push the next thunk, which evaluates the rest of the choice points, onto the stack...

choices:

```
'((thunk (k3 (-< 'b))))
```

```
(next!)
=> ((thunk (k1 (-< 2))))
=> (k1 (-< 2))
=> (k1 2)
=> (list 2 (-< 'a 'b))
```

k1:

```
(λ (x)
  (list x (-< 'a 'b))
```

k3:

```
(λ (x) (list 2 x)
```

And now we evaluate the amb in the redex...

```
(next!)
```

choices:

```
'((thunk (k3 (-< 'b))))
```

k3:

```
(λ (x) (list 2 x)
```

```
(next!)
=> ((thunk (k3 (-< 'b))))
```

choices:

```
'()
```

k3:

```
(λ (x) (list 2 x)
```

```
(next!)
=> ((thunk (k3 (-< 'b))))
=> (k3 (-< 'b))
```

choices:

```
'()
```

k3:

```
(λ (x) (list 2 x))
```

```
(next!)
=> ((thunk (k3 (-< 'b))))
=> (k3 (-< 'b))
=> (k3 'b)
```

choices:

'()

k3:

```
(λ (x) (list 2 x)
```

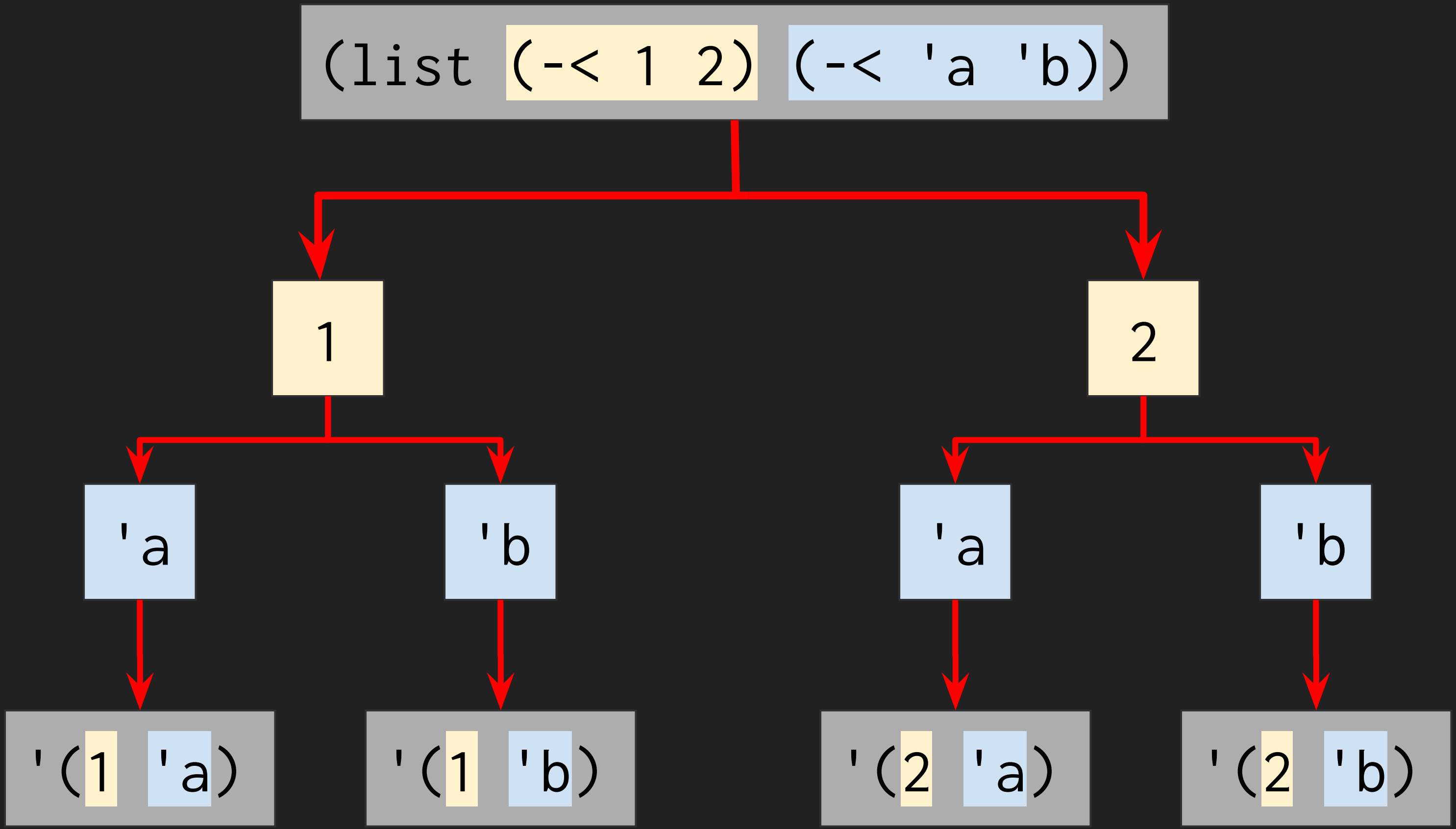
choices:

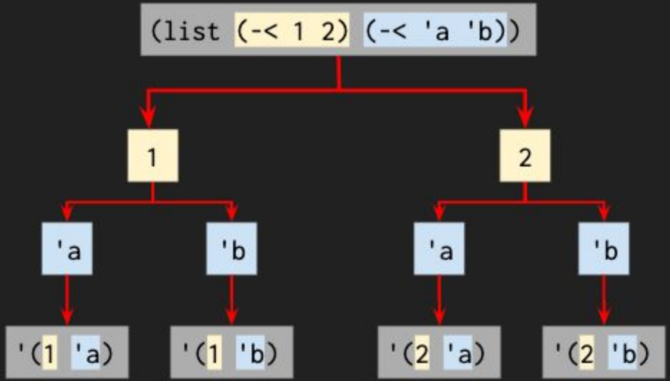
'()

k3:

(λ (x) (list 2 x)

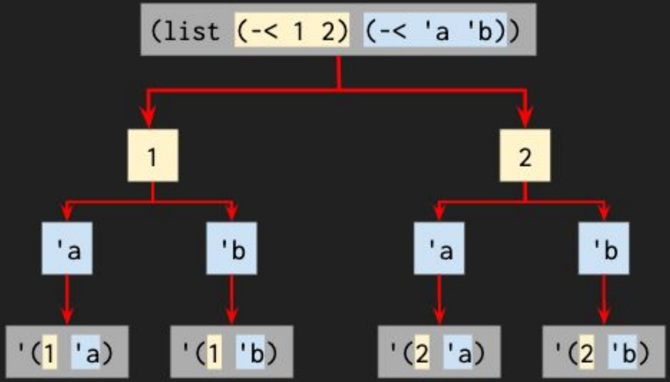
```
(next!)
=> ((thunk (k3 (-< 'b))))
=> (k3 (-< 'b))
=> (k3 'b)
(list 2 'b)
```


(list (-< 1 2) (-< 'a 'b))
1
2
'a
'b
'a
'b
'(1 'a)
'(1 'b)
'(2 'a)
'(2 'b)



Evaluating -< expressions automatically gives us a form of depth-first search through the space of all combinations of choices

Calling (next) **backtracks** as necessary to yield all remaining subsequent expressions.

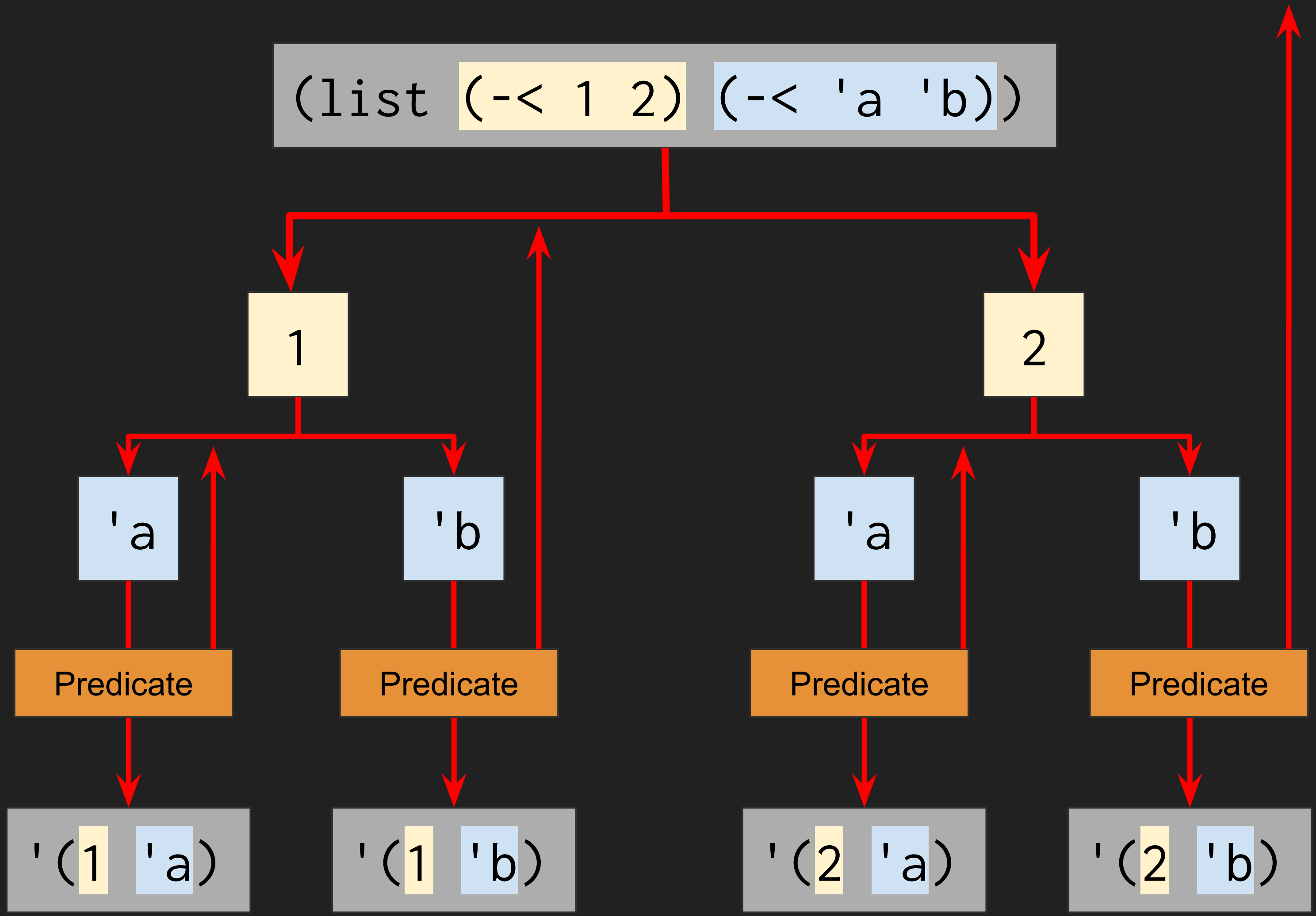


If we had a way of asking questions about each expression we generate, we would have a way of performing **backtrack searching!**

Implementing backtrack search is our first step into the world of **logic programming**. You will explore logic programming more in the second assignment.

Our goal: the query operator

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (?- even? (-< 1 2 3 4))
2
> (next!)
4
> (next!)
'done
>
```



(list (-< 1 2) (-< 'a 'b))
1
2
'a
'b
'a
'b
Predicate
Predicate
Predicate
Predicate
'(1 'a)
'(1 'b)
'(2 'a)
'(2 'b)

Our initial implementation...

```
(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))
```

```
Language: racket, with debugging; memory limit: 256 MB.
> (?- even? (-< 1 2 3 4))
2
> (next!)
4
> (next!)
'done
> |
```

An issue with our implementation...

```
(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))
```

```
> (* 10 (?- even? (-< 1 2 3 4)))
200
> ; shouldn't this have been 20?
```

This suggests an issue with our use of (next!)

```
> (* 10 (?- even? (-< 1 2 3 4)))
200
> ; shouldn't this have been 20?
```

```
> (* 10 (?- odd? (-< 1 2 3 4)))
10
> (next!)
300
```

This suggests an issue with our use of (next!)

Remember that (next!) delimits the continuation with reset because we want to use (next!) in larger expressions, like below here

```
(define (next!)
  (if (empty? choices)
      (shift k 'done)
      (reset ((get-choice!)))))

(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (-< 1 2 3)
1
> (* 10 (next!))
20
>
```

This suggests an issue with our use of (next!)

Also remember that (-< …) captures the entire expression as well, so we can implement expressions like below.

So in our case, we've evaluated the surrounding context with ?- twice! Once from capturing it with -< and again with resetting in (next!).

```
(define (next!)
  (if (empty? choices)
      (shift k 'done)
      (reset ((get-choice!)))))

(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (* 10 (-< 1 2 3 4))
10
>
```

This suggests an issue with our use of (next!)

Notice that we can still use (next!) to get the next filtered element, because the captured continuation uses (backtrack!) to implicitly call (next!) until the next choice satisfies the predicate

```
(define (next!)
  (if (empty? choices)
      (shift k 'done)
      (reset ((get-choice!)))))

(define (backtrack!)
  (shift k (next!)))

(define (?- pred expr)
  (if (pred expr)
      expr
      (backtrack!)))
```

```
Language: racket, with debugging; memory limit: 256 MB.
> (* 10 (?- even? (-< 1 2 3 4)))
20
> (next!)
40
> (next!)
'done
> 
```

Next time

- Examples of backtracking search
- Introduction to type theory