Advanced Stanza Features

Patrick S. Li

Stanza's Technical Pillars

- Stanza is built up from five interacting subsystems:
  - Optional Typing
  - Multimethod Objects
  - Targetable Coroutines
  - Programmatic Macros
  - LoStanza System Language

Stanza's Technical Pillars

- Stanza is built up from five interacting subsystems:
  - Optional Typing
  - Multimethod Objects
  - Targetable Coroutines
  - Programmatic Macros
  - LoStanza System Language

You use these all the time.

Targetable Coroutines

- A Coroutine is like a subroutine:
  - Instead of **return**, you have **yield**.
  - **yield** does the same thing as **return** the first time it is called, but:
    - *Remembers* where it first executed.
    - *Resumes* from where it left off when called again.

Lazy Generation

```
defn count-up-and-down (n:Int) :
  while true :
    for i in 0 to n do :
      println(i)
    for i in n to 0 by -1 do :
      println(i)

count-up-and-down(4)
```

```
0
1
2
3
4
3
2
1
0
1
2
3
4
3
2
1
0
...
```

Lazy Generation

```
defn count-up-and-down (n:Int) :
  while true :
    for i in 0 to n do :
      println(“Counting up”)
      println(i)
    for i in n to 0 by -1 do :
      println(“Counting down”)
      println(i)

count-up-and-down(4)
```

```
Counting up
0
Counting up
1
Counting up
2
Counting up
3
Counting down
4
Counting down
3
Counting down
2
Counting down
1
Counting up
0
...
```

Lazy Generation

```
defn count-up-and-down (n:Int) :
  generate<Int> :
    while true :
      for i in 0 through n do :
        println("Counting up")
        yield(i)
      for i in n through 0 by -1 do :
        println("Counting down")
        yield(i)

val xs = count-up-and-down(4)
println("First one")
println(next(xs))
```

```
First one
Counting up
0
```

Lazy Flattening

```
defn flatten (x) :
  generate :
    let loop (x = x) :
      match(x:List) : do(loop, x)
      else : yield(x)

val xs = flatten(`(0 (1 (2 3)) ((4) 5)))
do(println, take-n(3, xs))
```

```
0
1
2
```

Lazy Flattening

```
defn flatten (x) :
  generate :
    let loop (x = x) :
      match(x:List) : do(loop, x)
      else : yield(x)

val xs = flatten(`(0 (1 (2 3)) ((4) 5)))
val ys = flatten(`((0 1) 2 ((3)) 4 (5)))
println(all?(equal?,
             flatten(xs),
             flatten(ys)))
```

```
true
```

Lazy Flattening

```
defn flatten (x) :
  generate :
    let loop (x = x) :
      match(x:List) : do(loop, x)
      else : yield(x)

val xs = flatten(`(0 (1 (8 3)) ((4) 5)))
val ys = flatten(`((0 1) 2 ((3)) 4 (5)))
println(all?(equal?,
             flatten(xs),
             flatten(ys)))
```

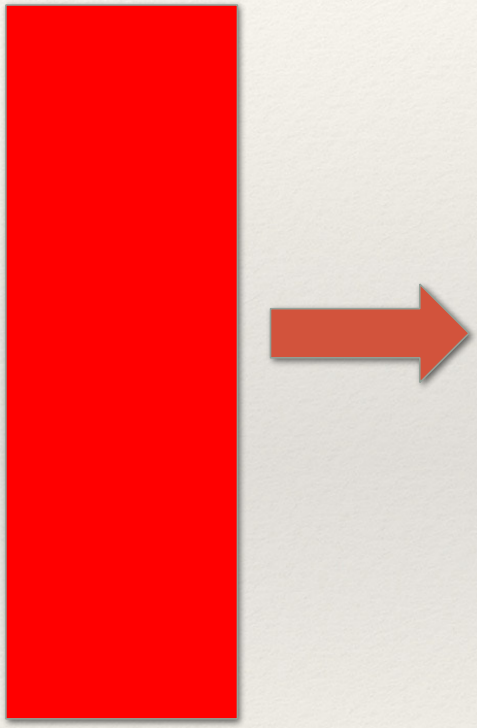
```
false
```

Fundamentals of Animation

```
... state ...

defn tick () :
  ... update state ...

defn draw () :
  ... draw state ...

while true :
  tick()
  draw()
```

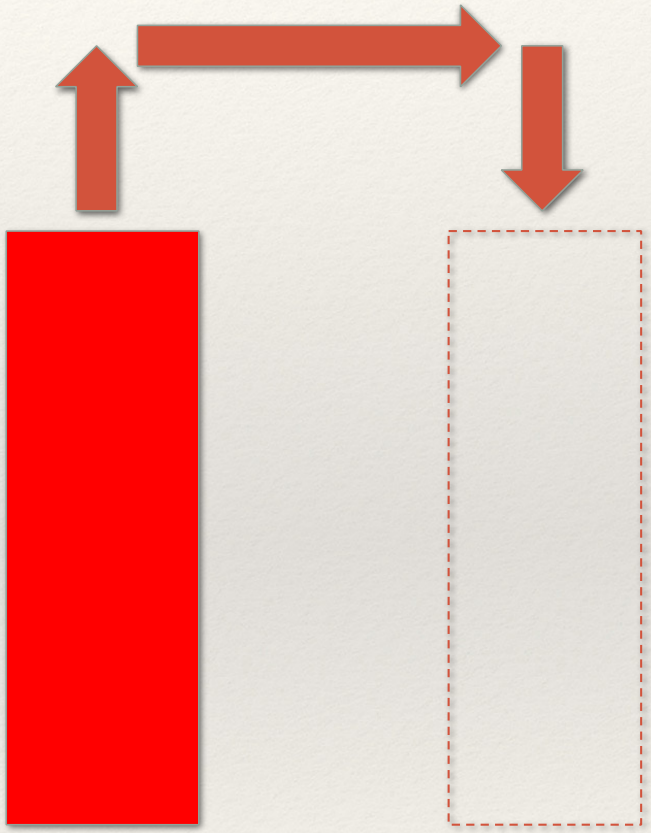
Fundamentals of Animation

```
var x = 0
var y = 0

defn tick () :
  x = x + 1

defn draw () :
  draw-box(x, y)

while true :
  tick()
  draw()
```



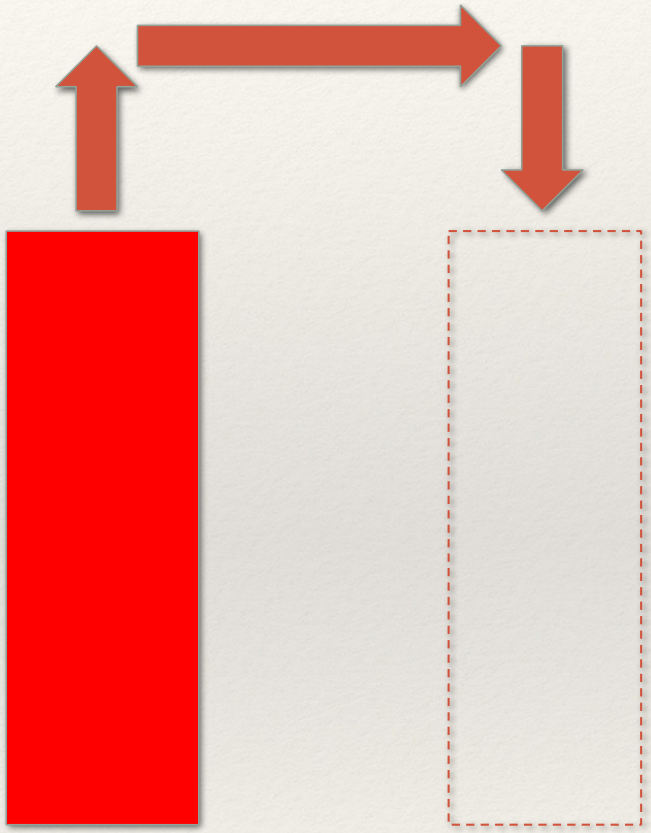
Fundamentals of Animation

```
var x = 0
var y = 0
var action = "UP"

defn tick () :
  if action == "UP" :
    y = y + 1
    if y > 5 :
      action = "RIGHT"
  else if action == "RIGHT" :
    x = x + 1
    if x > 10 :
      action = "DOWN"
  else if action == "DOWN" :
    y = y - 1
    if y <= 0 :
      action = "STOP"
```



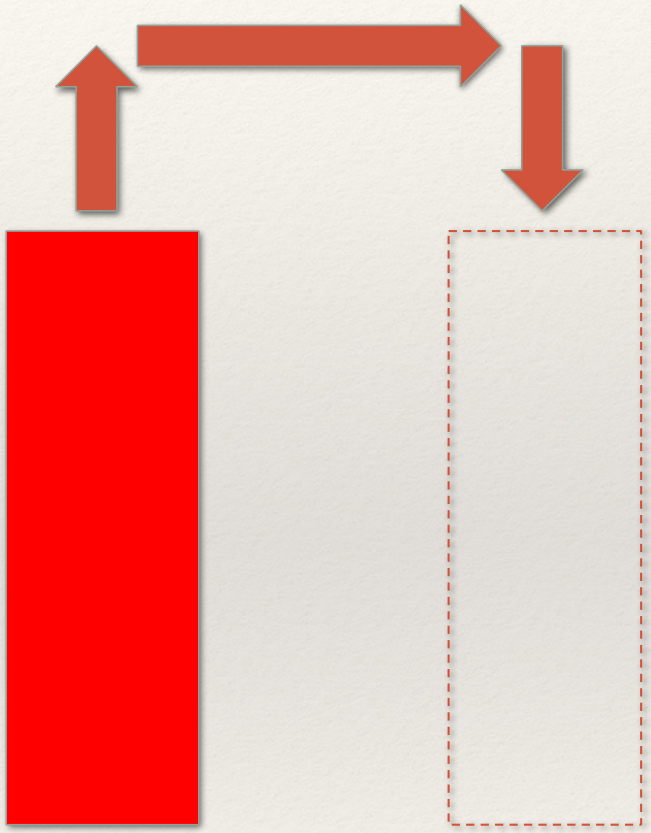
Fundamentals of Animation

```
var x = 0
var y = 0

defn ticking () :
  generate<False> :
    while y <= 5 :
      y = y + 1
      yield(false)
    while x <= 10 :
      x = x + 1
      yield(false)
    while y >= 0 :
      y = y - 1
      yield(false)

val ticks = ticking()
while not empty?(ticks) :
  next(ticks)
  draw()
```



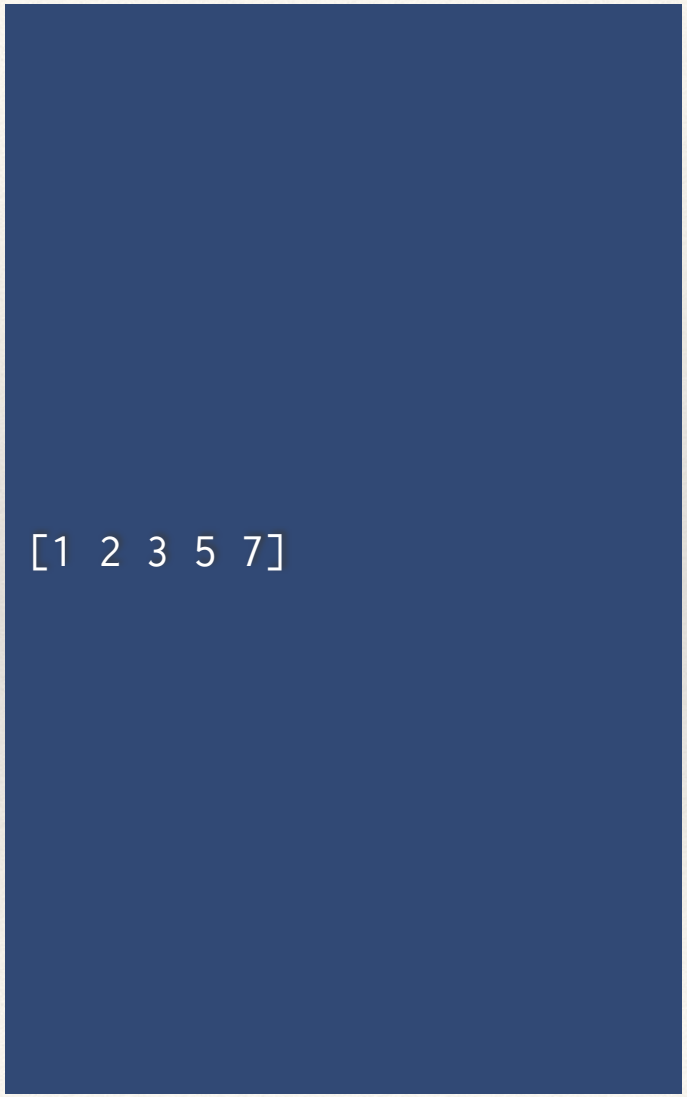
Fundamentals of Animation

```
var x = 0
var y = 0

defn ticking () :
  generate<False> :
    defn slide (dx, dy, n) :
      for i in 0 to n do :
        x = x + dx
        y = y + dy
        yield(false)

    slide(0, 1, 5)
    slide(1, 0, 10)
    slide(0, -1, 5)

val ticks = ticking()
while not empty?(ticks) :
  next(ticks)
  draw()
```



Insertion Sort

```
defn insertion-sort (xs) :
  val N = length(xs)
  for i in 1 to N do :
    val insert = xs[i]
    val j = find!({xs[_] > insert}, 0 to i)
    for k in i to j by -1 do :
      xs[k] = xs[k - 1]
    xs[j] = insert
  xs

val xs = to-array<Int>([1 5 3 2 7])
insertion-sort(xs)
println(xs)
```



Insertion Sort

```
defn insertion-sort (xs) :
  val N = length(xs)
  for i in 1 to N do :
    val insert = xs[i]
    val j = find!({xs[_] > insert}, 0 to i)
    for k in i to j by -1 do :
      xs[k] = xs[k - 1]
    xs[j] = insert
  xs

animate(insertion-sort)
```

Selection Sort

```
defn selection-sort (xs) :
  val N = length(xs)
  for s in 0 to (N - 1) do :
    ;Find minimum
    var min-i = s
    var min-v = xs[s]
    for i in (s + 1) to N do :
      if xs[i] < min-v :
        min-i = i
        min-v = xs[i]

    ;Swap
    if min-i != s :
      xs[min-i] = xs[s]
      xs[s] = min-v

animate(selection-sort)
```

Decoupled Animations

```
defn selection-sort (xs) :
  val N = length(xs)
  for s in 0 to (N - 1) do :
    ;Find minimum
    var min-i = s
    var min-v = xs[s]
    for i in (s + 1) to N do :
      if xs[i] < min-v :
        min-i = i
        min-v = xs[i]

    ;Swap
    if min-i != s :
      xs[min-i] = xs[s]
      xs[s] = min-v

machine-animate(selection-sort)
```

Concurrent Animations

```
side-by-side-animate(
  machine-animate{insertion-sort}
  machine-animate{quick-sort})
```

Nested Animations

```
tv-animate(
  machine-animate{insertion-sort}
  animate{quick-sort}
  animate{selection-sort})
```

Game Framework

```
play-game()
```

Programmatic Macros

- Stanza's general mechanism for *syntactic abstraction*.

Please run

`Bleh`

whenever I type

`Blah`

Unless Macro

```
unless x :
  do-this()
  do-that()
```

```
if not x :
  do-this()
  do-that()
```

```
defrule exp4 = (unless ?p:#exp : ?body:#exp) :
  parse-syntax[core / #exp](
    fill-template(`(if not x : y), [
      `x => p
      `y => body]))
```

Switch Macro

```
switch(x) :
  a : body1
  else : body2
```

```
if x == a : body1
else : body2
```

```
defrule exp4 = (switch(?x:#exp) :
                  ?a:#exp : ?b1:#exp
                  else : ?b2:#exp) :
  val template = `(
    if x == a : body1
    else : body2)
  parse-syntax[core / #exp](
    fill-template(template, [
      `x => x
      `a => a
      `body1 => b1
      `body2 => b2]))
```

Internal Languages

```
defresolver resolve-exp (e:IExp, eng:Engine) :

   ;Resolve top level expressions
   resolve te :
      IDefType: {args:+, parent:t, children:te}
      IDef: (type:t, value:e)
      IDefVar: (type:t, value:e)
      IDefn: {targs:+, args:+, a1:t, a2:t, body:e}

   ;Resolve Stanza expressions
   resolve e :
      Let: (def:e, body:e)
      LetRec: (defns:f+, defns:f, body:e)

   ...
```

Start a new scope

New Binders

Follow the "e" resolution rules

Functions are first defined before resolved.

Internal Languages

- Stanza's binder resolution phase is 207 lines using the `defresolver` macro.
- Expands to about 2000 lines of Stanza code.

External Languages

- JitPCB is just a collection of Stanza macros.
- Some other languages written/writing/will write in our lab:
  - Chipper (essentially JitRTL)
  - VMLang (essentially Jit - Virtual Machine)
  - Allure (essentially JitGUI)

Leveraging Talent

- Small team of high-talent programmers. How do you compete?
- Macros allow you to trade off:

  High Quantity Work
  for:
  High Difficulty Work

- Work smarter not harder:
  - Less Code
  - Less Bugs
  - More Consistent
  - Easier to Change
  - Easier to Extend