Redex:

Program Your Semantics

Robby Findler
Northwestern & PLT

```
_find:
        push  %rbp
        mov   %rsp,%rbp
start:
        cmp   $0,%rsi
        je    false
        cmp   (%rsi),%rdi
        je    true
        push  16(%rsi)
        mov   8(%rsi),%rsi
        call  _find
        pop   %rsi
        cmp   $0,%rax
        jne   true
        jmp   start
true:
        mov   $1,%rax
        pop   %rbp
        retq
false:
        mov   $0,%rax
        pop   %rbp
        retq
```

Do you recognize this function? It is a simple one.

Does it help that I point out that it is recursive?

But that it also has a loop (see the "jmp start")?

```
(struct node (val left right))
(struct leaf (val))

(find (node v1 left right) v2) :=
(or (= v2 v1)
    (find left v2)
    (find right v2))

(find (leaf v)  v)   := #true
(find (leaf v1) v2) := #false
```

How about this function? It is thesame function.

The previousslide was in a language that allexperienced programmers have encountered.This one is in a programming languagethat, while it is similar to others you haveseen I'm sure, it is one I just madeup for this talk.

Much clearer now, right? Functional programmingforever!

```
(node 10 (node 5 (leaf 1) (leaf 7)) (leaf 12))
1

(node 5 (leaf 1) (leaf 7))
1

(leaf 1)
1

#t
```

Even better, if we just stick a printf into beginning of the function then we can even learn a lot about about the behavior of this function, right?

There was no easy way to do this in assembly.

... Lets push this a bit.

```
(interp (? number? n)) := n

(interp (? symbol? x)) := (error 'interp "free var ~s" x)

(interp `(+ ,lhs ,rhs)) := (+ (interp lhs) (interp rhs))

(interp `(λ ,x ,body)) := `(λ ,x ,body)

(interp `(if0 ,tst ,thn ,els))
:= (if (equal? (interp tst) 0)
       (interp thn)
       (interp els))

(interp `(let ([,x ,rhs]) ,body))
:= (interp `((λ ,x ,body) ,rhs))

(interp `(,f ,x)) := (apply (interp f) (interp x))

(apply `(λ ,x ,b) a) := (interp (subst b x a))
(apply val arg) := (error 'interp "non-fun ~s" val)
```

What if we wanted to try to understand the (arguably) most important function that programmers deal with on a day-to-day basis, namely the function that tells you what your programs do?

Here things are getting a big cramped on the slide, but perhaps that is reasonable and you can still see a lot of structure. For example, check out the third line. It says ``the behavior of an addition expression is to evaluate its arguments and then sum them." That seems pretty good.

```
(struct kont (k))
(interp e) := (interp/k e (λ (x) x))

(interp/k (? number? n) k) := (k n)

(interp/k 'call/cc k) := (k 'call/cc)

(interp/k (kont k1) k2) := (k2 (kont k1))

(interp/k (? symbol? x) k) := (k (error 'interp "free var ~s" x))

(interp/k `(+ ,lhs ,rhs) k) := (+ (interp/k
                                    lhs
                                    (λ (l)
                                      (interp/k
                                       rhs
                                       (λ (r)
                                         (k (+ l r)))))))

(interp/k `(λ ,x ,body) k) := (k `(λ ,x ,body))

(interp/k `(if0 ,tst ,thn ,els) k) := (interp/k tst
                                                (λ (v)
                                                  (if (equal? v 0)
                                                      (interp/k thn k)
                                                      (interp/k els k))))

(interp/k `(let ([,x ,rhs]) ,body) k) := (interp/k `((λ ,x ,body) ,rhs) k)

(interp/k `(,f ,x) k) := (interp/k f
                                    (λ (fv)
                                      (interp/k
                                       x
                                       (λ (xv)
                                         (apply fv xv k)))))

(apply `(λ ,x ,b) a k) := (interp/k (subst b x a) k)
(apply `call/cc a k)    := (interp/k `(,a ,(kont k)) k)
(apply (kont k1) a k2)  := (interp/k a k1)
(apply val arg k)       := (error 'interp "non-fun ~s" val)
```

What happens if we add continuations to the mix?

Now we are kind of in a lot of trouble. I doubt the back of the room can even read this slide, but those in the front are probably going to have to make some scratch space somewhere and start thinking really hard about what's going on, even in that addition case.

```
((λ f (f (f 1)))
 (λ x (+ x x)))

(λ f (f (f 1)))

(λ x (+ x x))

((λ x (+ x x))
 ((λ x (+ x x)) 1))

(λ x (+ x x))

((λ x (+ x x)) 1)

(λ x (+ x x))

1

(+ 1 1)

1

1

(+ 2 2)

2

2

4
```

But lets return to the other interpreter and try that trick with sticking a printf into the function. It doesn't really look so good. If you understand interpreters really well, you see what is going on here, but there is information that doesn't show up in the prints; information is that is hidden in the ``call stack'' of the interpreter itself.

For example, how do you get from the ``1'' to the ``(+ 2 2)'' down near the bottom there? That information is missing.

$$
\begin{aligned}
e &::= (e\ e) \mid (\lambda\ x\ e) \mid x \mid (\text{let}\ ([x\ e])\ e) \mid (+\ e\ e) \\
&\quad \mid n \mid (\text{if0}\ e\ e\ e) \\
v &::= (\lambda\ x\ e) \mid n \\
E &::= [] \mid (v\ E) \mid (E\ e) \mid (+\ v\ E) \mid (+\ E\ e) \mid (\text{if0}\ E\ e\ e)
\end{aligned}
$$

$$
\begin{aligned}
E[(\text{let}\ ([x\ e_1])\ e_2)] &\longrightarrow E[((\lambda\ x\ e_2)\ e_1)] \\
E[((\lambda\ x\ e)\ v)] &\longrightarrow E[e\{x{:=}v\}] \\
E[(\text{if0}\ 0\ e_1\ e_2)] &\longrightarrow E[e_1] \\
E[(\text{if0}\ v\ e_1\ e_2)] &\longrightarrow E[e_2] \text{ where } v \neq 0 \\
E[(+\ n_1\ n_2)] &\longrightarrow E[\Sigma[[n_1, n_2]]]
\end{aligned}
$$

Okay, so here is a completely different way to write down that same, continuation-free interpreter.

First thing to note: we've got a bigger font here, so we need less writing to say the same thing.

Second thing to note: this is actually saying something explicitly that was only implicit before, namely that first line is giving us a definition of the language. So this version has more information than the functional version a few slides back did.

Okay, so what is this notation? It is the way that a programming languages semanticist would write down the interpreter, without continuations.

$$
\begin{aligned}
e &::= (e\ e) \mid (\lambda\ x\ e) \mid x \mid (\mathsf{let}\ ([x\ e])\ e) \mid (+\ e\ e) \\
&\quad \mid n \mid (\mathsf{if0}\ e\ e\ e) \mid (\mathsf{abort}\ e) \mid \mathsf{call/cc} \\
v &::= (\lambda\ x\ e) \mid n \mid \mathsf{call/cc} \\
E &::= [] \mid (v\ E) \mid (E\ e) \mid (+\ v\ E) \mid (+\ E\ e) \mid (\mathsf{if0}\ E\ e\ e)
\end{aligned}
$$

$$
\begin{aligned}
E[(\mathsf{let}\ ([x\ e_1])\ e_2)] &\longrightarrow E[((\lambda\ x\ e_2)\ e_1)] \\
E[((\lambda\ x\ e)\ v)] &\longrightarrow E[e\{x{:=}v\}] \\
E[(\mathsf{if0}\ 0\ e_1\ e_2)] &\longrightarrow E[e_1] \\
E[(\mathsf{if0}\ v\ e_1\ e_2)] &\longrightarrow E[e_2] \text{ where } v \neq 0 \\
E[(+\ n_1\ n_2)] &\longrightarrow E[\Sigma[[n_1, n_2]]] \\
E[(\mathsf{call/cc}\ v)] &\longrightarrow E[(v\ (\lambda\ x\ (\mathsf{abort}\ E[x])))] \\
E[(\mathsf{abort}\ e)] &\longrightarrow e
\end{aligned}
$$

And here's the version with continuations. Just a simple change. Why that is will become clear later in the talk, but in general this notation is one that is very composable. Adding new features to the programming language often (but not always) requires only local changes to the semantics.

```
((λ f (f (f 1)))
 (λ x (+ x x)))

((λ x (+ x x))
 ((λ x (+ x x)) 1))

((λ x (+ x x))
 (+ 1 1))

((λ x (+ x x)) 2)

(+ 2 2)

4
```

And this notation also makes it easy to see traces. This is the trace you get from that same expression we saw earlier and now there is no hidden context.

Alright: so what I want to spend the rest of the talk doing is showing you how this too can be a programming language, at a level of abstraction higher than functional programming.

Redex demo

```
#lang racket
(require redex)
(define-language L
  (e (if e e e)
     true
     false
     (+ e ...)
     number)
  (E hole
     (if E e e)
     (+ number ... E e ...)))
```

The code shown in the next few slides is one way the demo might have gone.

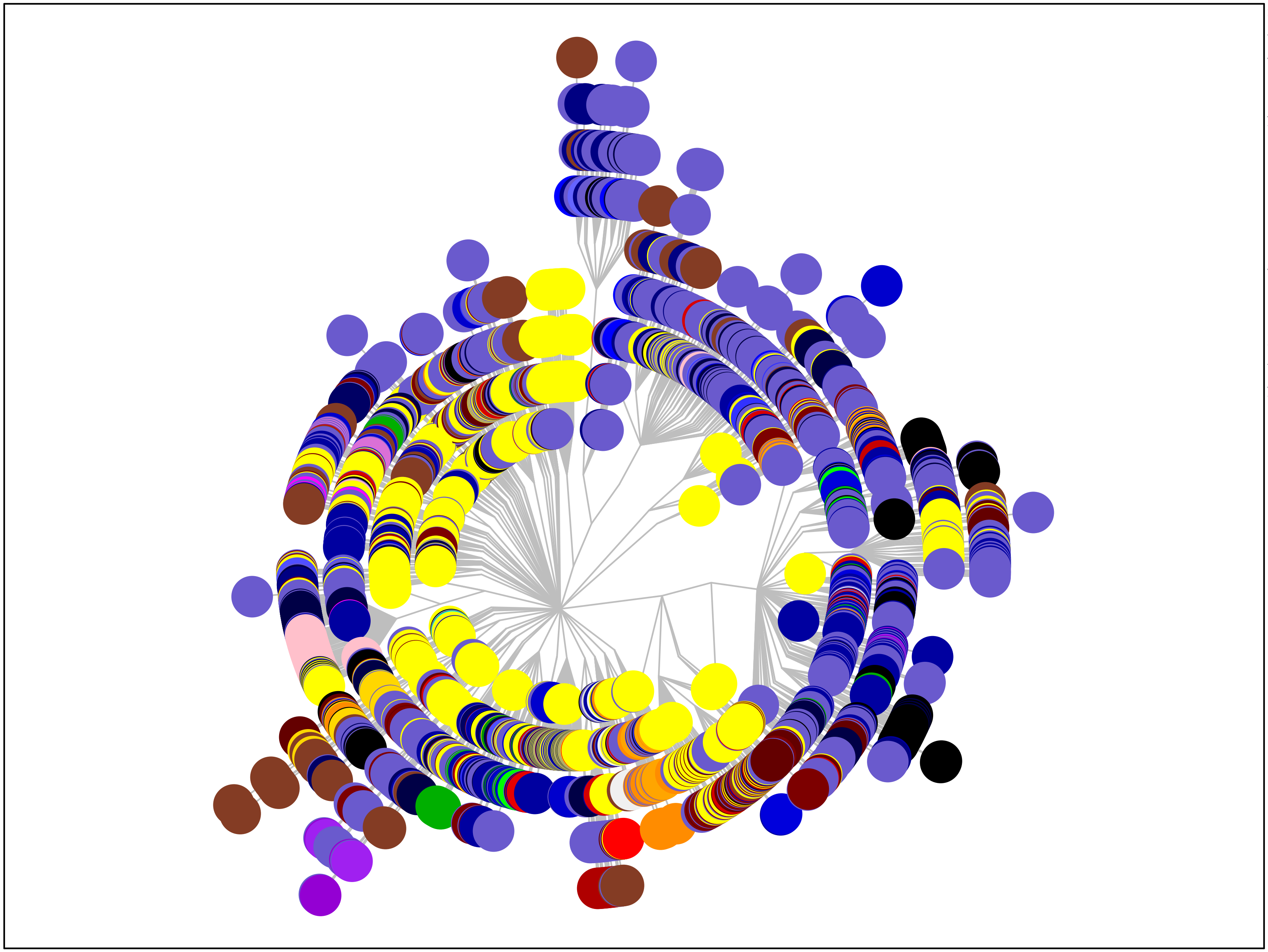
```
(define red
  (reduction-relation
   L
   (--> (in-hole E (if true e1 e2))
        (in-hole E e1))
   (--> (in-hole E (if number e1 e2))
        (in-hole E e1))
   (--> (in-hole E (if false e1 e2))
        (in-hole E e1))
   (--> (in-hole E (+ number1 number2))
        (in-hole E ,(+ (term number1) (term number2))))
   (--> (in-hole E (+)) (in-hole E 0))
   (--> (in-hole E (+ number)) (in-hole E number))
   (--> (in-hole E (+ number1 number2
                      number3 number4 ...))
        (in-hole E (+ (+ number1 number2)
                      number3 number4 ...)))))
```

```
(traces
 red
 (term (+ 1 2 3
          (if false 4 5)
          (if true 6 7))))
```

```
(define (value-or-reduces? e)
  (or (redex-match L true e)
      (redex-match L false e)
      (redex-match L number e)
      (pair? (apply-reduction-relation red e))))
(redex-check
 L
 e
 (value-or-reduces? (term e)))
```

From here to continuations:

- Substitution
- Contexts are continuations (just wrap a λ)



Stepping back for a moment, I'd like toconnect Redex to the larger Racket worldit lives in.

If you've seen a talk about Racket before, you'veprobably seen a version of this graph: it showsa bubble for each file and each bubble is coloredbased on the language it is implemented in.

Racket is a language incubatorwhere we build lots of languagesall the time, as we find we are oftenin the situation described in the beginningof the talk. And as we are language researcherswho work on language semantics we built Redex.And since we give talks, we built a languagefor that. Etc. So, if you have a need fora language, consider Redex: it is proven techfor implementing languages.

Thank you.