FDL "fer-de-lance"

First Class Functions

↳ Assign is up due just before final

funEnv varEnv

Functions as Values

Consider the following `egg` program

```
def f(it):
  it(5)

def incr(x):
  x + 1

f(incr)
```

unbound func 'it'

it ∉ funenv

unbound var 'incr'

incr ∉ varEnv

What will be the result of compiling/running?

on "diamond"

We have functions, but they are *second-class* entities in our languages: they don't have the same *abilities* as other values.

So, we get multiple error messages:

```
Errors found!
tests/input/hof.diamond:(2:3)-(4:1): Function 'it' is not defined

         2|    it(5)
              ^^

tests/input/hof.diamond:7:3-7: Unbound variable 'incr'

         7|  f(incr)
              ^^^^
```

This is because the `Env` only holds

- parameters, and
- let-bound variables

and **not** function definitions.

Functions as Values

But for the many reasons we saw in CSE 130 – we *want* to treat functions like values. For example, if you run the above in Python you get:

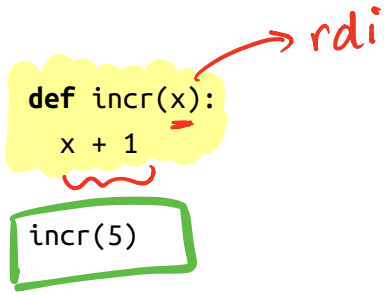
```
>>> def f(it): return it(5)

>>> def incr(x): return x + 1

>>> f(incr)
6
```

Flashback: How do we compile `incr`*?*

We compile each function down into a sequence of instructions corresponding to its body.



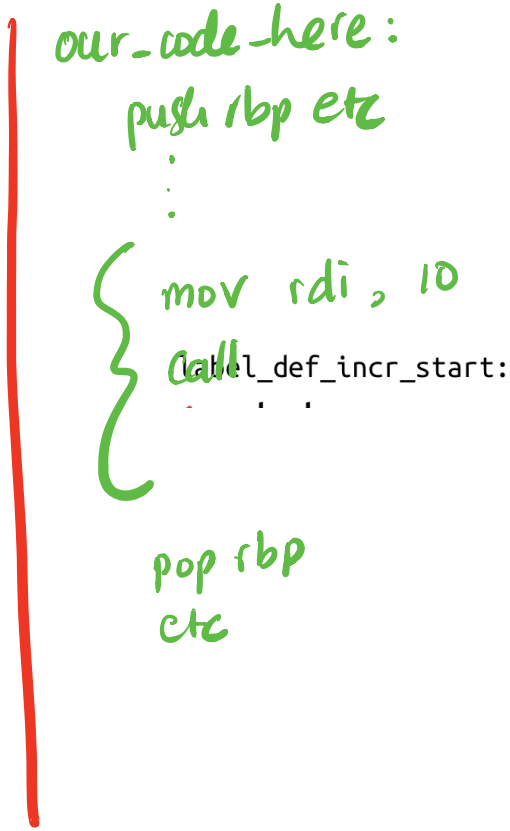
becomes, for `incr`

```
label_def_incr_start:
  push rbp                  # setup stack frame
  mov rbp, rsp

  mov rax, rdi              # grab param
  add rax, 2                # incr by 1

  mov rsp, rbp              # undo stack frame
  pop rbp
  ret                       # buh-bye
```

for the main expression



```
our_code_starts_here:
  push rbp
  mov rbp, rsp

  mov  rdi 10                 # push arg '5'
  call label_def_incr_start   # call function

  mov rsp, rbp
  pop  rbp
  ret
```

What is the value of a function?

So now, lets take a step back. Suppose we want to compile

```
def f(it):
  it(5)

def incr(x):
  x + 1

f(incr)
```

"enable passing FUNCTION as a param"

incr ⟼ "label"

it ⟼

Attempt 1: What is the value of the parameter `it` *?*

IDEA: Use the **label** where `incr` lives!

```
label_def_f_start:
  push rbp
  mov  rbp, rsp

  mov rax, rdi        # grab function-address
  mov rdi, 10         # push arg '5'
  call rax            # call function!

  mov rsp, rbp
  pop rbp
  ret
```

```
def f(it):
  it(5)
```

(1) How do we know rdi is a func?

(2) what is the addr?

How to pass the value of the parameter ?

So now the `main` expression

```
f(incr)
```

can be compiled to:

```
our_code_starts_here:
  push rbp
  mov rbp, rsp

  mov rdi, ?1      # push arg
  call ?2          # call function

  mov rsp, rbp
  pop rbp
  ret
```

addr(label of incr)

f(incr)

f(12)

func 'f' being called

QUIZ: What are suitable terms for `?1` and `?2` ?

	?1	?2
A	label_def_incr_start	label_def_f_start
B	label_def_f_start	label_def_incr_start
C	label_def_f_start	label_def_f_start
D	label_def_incr_start	label_def_incr_start

Strategy Progression

1. **Representation** = `Start-Label`

```
- **Problem:** How to do run-time checks of valid args?
```

Yay, that was easy! How should the following

behave?

```
def f(it):
  it(5)

def add(x, y):
  x + y

f(add)
```

```
mov rax, L_add
mov rdi, 10
call rax

mov rax, rdi
add rax, rsi
gibberish
```

Lets see what Python does:

```
>>> def f(it): return it(5)
>>> def add(x,y): return x + y
>>> f(add)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in f
TypeError: add() takes exactly 2 arguments (1 given)
```

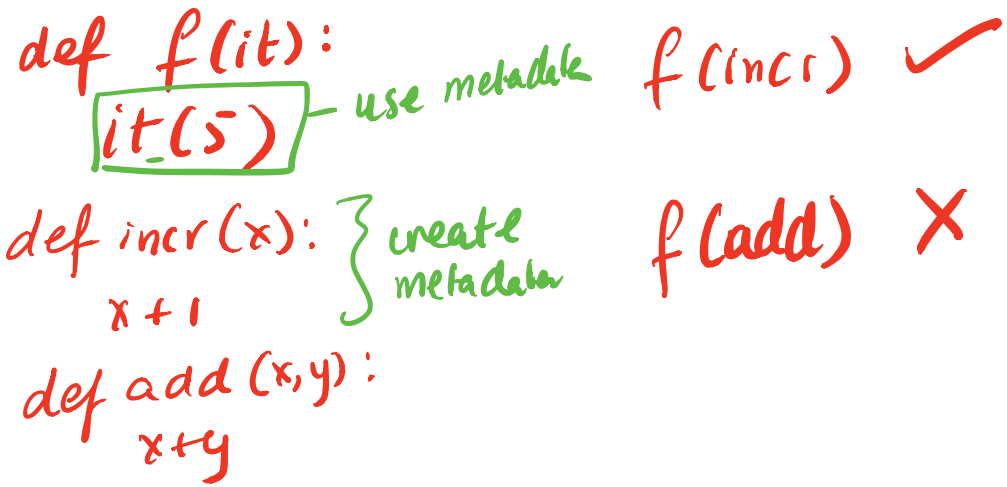
Problem: Ensure Valid Number of Arguments?

How to make sure

- `f(incr)` **succeeds**, but
- `f(add)` **fails**

With **proper run-time error**?

1. **Where** does the run-time check happen?
2. **What** information is needed for the check?



Key: Need to *also* store the function's **arity**

- The **number of arguments** required by the function

Strategy Progression

1. **Representation** = `Start-Label`
    - **Problem:** How to do run-time checks of valid args?
2. **Representation** = `(Arity, Start-Label)`

Attempt 2: What is the value of the parameter **`it`** *?*

```
def f(it):
  it(5)

def incr(x):
  x + 1

f(incr)
```

it = incr → (1, l_incr_start)

it = add (2, l_add)

Tags
000 num
001 tuple
111 bool
101 fun-tuple

IDEA: Represent a *function* with a **tuple of**

```
{arity, function_start_label}
```

1. check e is function
2. check args is correct
3. more args into reg/stack
4. call the function

We can now compile a call

```
e(x1,...,xn)
```

via the following strategy:

it (5)

rax ← <e>

1. assert (rax & 0x111 == 0x101)
2. assert (len xs == rax[0])
3. as before
4. call (rax[1])

1. **Evaluate** the tuple `e`
2. **Check** that `e[0]` is equal to `n` (else arity mismatch error)
3. **Call** the function at `e[1]`

Example

Lets see how we would compile this

```
def f(it):
  it(5)

def incr(x):
  x + 1

f(incr)
```

We need to map **the variable**

`incr`

to **the tuple**

`(1, label_def_incr_start)`

let f = λit → it(5)
, incr = λx → x+1
in
f(incr)

1. create "tuple" for incr
2. name 'incr'

put the func-tuples IN the var env

But **where** will we store this information?

Strategy Progression

1. **Representation** = `Start-Label`
    - **Problem:** How to do run-time checks of valid args?
2. **Representation** = `(Arity, Start-Label)`
    - **Problem:** How to map function **names** to tuples?
3. **Lambda Terms** Make functions just another expression!

Attempt 3: Lambda Terms

So far, we could only define functions at **top-level**

- First-class functions are like *any* other expression,
- Can define a function, wherever you have any other expression.

Language	**Syntax**
Haskell	`\(x1,...,xn) -> e`
Ocaml	`fun (x1,...,xn) -> e`

Language	Syntax
JS	`(x1,...,xn) => { return e }`
C++	`[&](x1,...,xn){ return e }`

Example: Lambda Terms

We can now replace `def` as:

```
let f    = (lambda (it): it(5))
  , incr = (lambda  (x): x + 1)
in
  f(incr)
```

Implementation

As always, to the details! Lets figure out:

Representation

1. How to store function-tuples

Types:

1. ~~Remove Def~~
2. Add `lambda` to `Expr`

Transforms

1. Update `tag` and `ANF`
2. Update `checker`
3. Update `compile`

Implementation: Representation

Represent `lambda-tuples''` or function-tuples'' via a special tag:

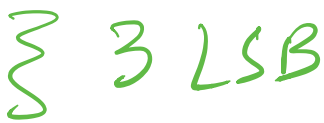
Type	LSB
number	xx0
boolean	111
pointer	001
function	101

In our code:

```
data Ty = ... | TClosure

typeTag :: Ty -> Arg
typeTag TTuple    = HexConst 0x00000001
typeTag TClosure  = HexConst 0x00000005

typeMask :: Ty -> Arg
typeMask TTuple   = HexConst 0x00000007
typeMask TClosure = HexConst 0x00000007
```



So, **Function Values** represented just like a tuples

- padding, ~~FF~~ etc.
- but with tag `101` .

Crucially, we can **get** `0`-th, or `1`-st elements from tuple.

Question: Why not use *plain tuples*?

↳ to ensure e[1] is a func label!

Implementation: Types

First, lets look at the new `Expr` type

- ~~No more `Def`~~

params

body

```
data Expr a
  = ...
  | Lam              [Bind a]   !(Expr a) a      -- fun. definitions
  | App   !(Expr a)  [Expr a]             a      -- fun. calls
```

So we represent a **function-definition** as:

```
Lam [x1,...,xn] e
```

Lam [x1,..., xn] e

and a **function call** as:

```
App e [e1,...,en]
```

App e [e1,... en]

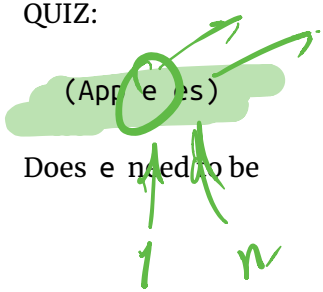
previously ID which we lookup in funEnv

Transforms: Tag

This is pretty straight forward (do it yourself)

Transforms: ANF

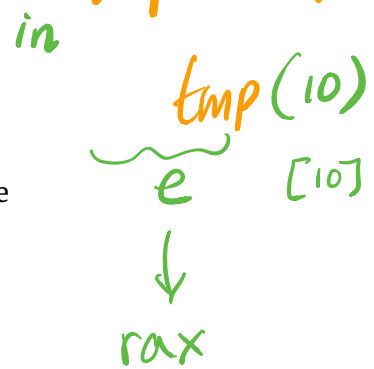
QUIZ:



Does e need to be

```
let tup =
  ( lambda(x) : x+1
  , lambda(y) : y-1
  )
tmp = tup[0]
```

- **A** Immediate
- **B** ANF
- **C** None of the above



Transforms: ANF

QUIZ:

```
(App e es)
```

Do `es` need to be

- **A** Immediate
- **B** ANF
- **C** None of the above

Transforms: ANF

The `App` case, fun + args should be **immediate**

- **Need the values to push on stack and make the call happen!**

Just like function calls (in `diamondback`), except

- Must also handle the **callee-expression** (named `e` below)

```
anf i (App e es)   = (i', stitch bs (App v vs))
  where
    (i', bs, v:vs) = imms i (e:es)
```

Transforms: ANF

QUIZ:

```
(Lam xs e)
```

Does `e` need to be

(A) IMM

(B) ANF is OK

- **A** Immediate
- **B** ANF
- **C** None of the above

Transforms: ANF

The `Lam` case, the body will be **executed** (when called)

- So we just need to make sure its in ANF (like all the code!)

```
anf i (Lam xs e) = (i', Lam xs e')
  where
    (i', e')     = anf i e
```

Transforms: Checker

We just have `Expr` (no `Def`) so there is a single function:

```
wellFormed :: BareExpr -> [UserError]
wellFormed = go emptyEnv
  where
    gos                       = concatMap . go
    go _    (Boolean {})      = ...
    go _    (Number  n     l) = largeNumberErrors      n l
    go vEnv (Id      x     l) = unboundVarErrors  vEnv x l
    go vEnv (Prim1 _ e     _) = ...
    go vEnv (Prim2 _ e1 e2 _) = ...
    go vEnv (If   e1 e2 e3 _) = ...
    go vEnv (Let x e1 e2   _) = ... ++ go vEnv e1 ++ go (addEnv x vEn
v) e2
    go vEnv (Tuple es      _) = ...
    go vEnv (GetItem e1 e2 _) = ...
    go vEnv (App e es      _) = ?1
    go vEnv (Lam xs e      _) = ?2 ++ go ?3 e
```

- How shall we implement `?1` ?
- How shall we implement `?2` ?
- How shall we implement `?3` ?