# "fer-de-lance" First Class Functions Functions as Values Consider the fall Consider the following egg program def f(it): unbound func 'it' it & funear **def** incr(x): x + 1unbound var incr incr & vartnv f(incr)

What will be the result of compiling/running?

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

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

# Flashback: How do we compile incr?

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

```
> rdi
def incr(x):
  x + 1
incr(5)
becomes, for incr
label def incr start:
                            # setup stack frame
  push rbp
  mov rbp, rsp
  mov rax, rdi
                            # grab param
  add rax, 2
                            # incr by 1
                            # undo stack frame
  mov rsp, rbp
  pop rbp
                            # buh-bye
  ret
for the main expression
```

our-code-here:
push 16p etc Call\_def\_incr\_start:

```
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): "lenable passing"

ti(5)

FUNCTION

def incr(x):
    x + 1

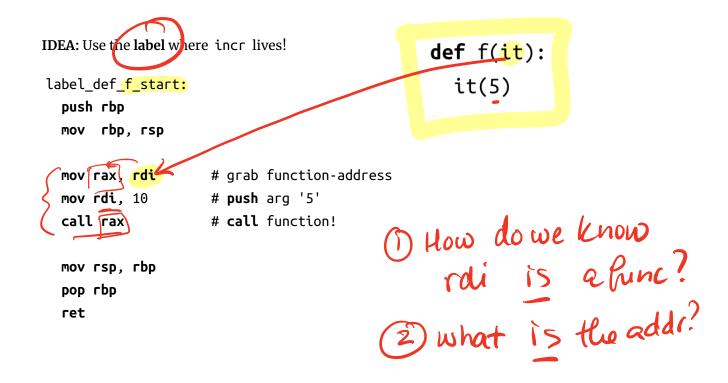
AS = A PAIAM

f(incr)

incr \longrightarrow "label"

it
```

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

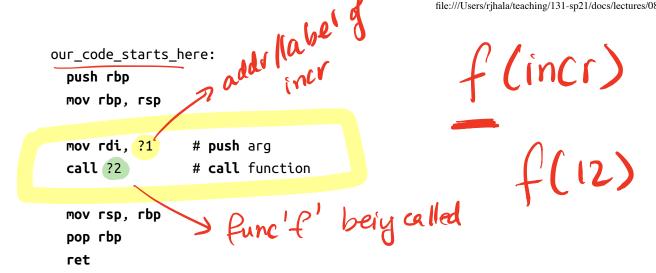


# How to pass the value of the parameter?

So now the main expression

f(incr)

can be compiled to:



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

	?1	?2
A	<pre>label_def_incr_start</pre>	label_def_f_start
В	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, Ladd mov rdi, 10 call roxx mov rax, rdi add rax, rsi gibbersh

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?

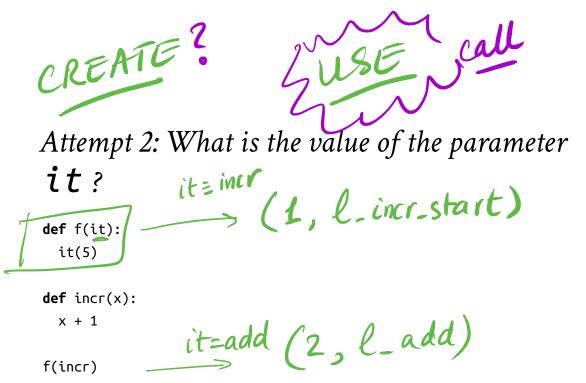
- 1. Where does the run-time check happen?
- 2. What information is needed for the check?

Key: Need to also store the function's arity

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

# Strategy Progression

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



Tous
000 num
001 tuple
11 ( bool
10 1 funtuple

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

We can now compile a call

via the following strategy:

1. check e is function
2. check args is correct
3. more args into registalk
4. Call the function

1. assert (rax Q OxIII = 6101)

- 2. assert (leu xs == rax[0]) 3. as before
- 4. call (mx[4])

rax < (e)

18 of 60

- 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]

$$\begin{array}{c} \text{Let } f = \text{$\lambda$ it} \Rightarrow \text{$it(5)$} \\ \text{$incr$} = \text{$\lambda$} \Rightarrow \text{$x+1$} \\ \text{$incr$} = \text{$\lambda$} \Rightarrow \text{$x+1$} \\ \text{$incr$} \\ \text{$def f(it): it(5)} \\ \text{$it(5)$} \\ \text{$1.$ $create "hiple" for incr} \\ \text{$def incr(x): } \\ \text{$x+1$} \\ \text{$f(incr)$} \\ \text{$vereath{$x$}} & \text{$incr$} \\ \text{$to the tuple} \\ \text{$incr$} \\ \text{$to the tuple} \\ \text{$(1, label\_def\_incr\_start)} & \text{$incr$} \\ \text{$i$$

20 of 60

But where will we store this information?

# Strategy Progression

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

## 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:

cse131

```
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

cse131 file:///Users/rjhala/teaching/131-sp21/docs/lectures/08-fer-de-lance.html

#### Types:

- 1. Pemove Dof
- 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:

Туре	LSB
number	xx0
boolean	111
pointer	001
function	101

#### In our code:

typeTag :: Ty -> Arg

typeTag TTuple = HexConst 0x00000001

typeTag TClosure = HexConst 0x00000005

typeMask :: Ty -> Arg

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

- padding, etc.
- but with tag 101.

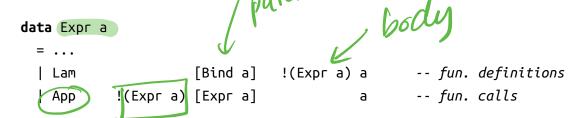
Crucially, we can  ${\bf get}\ 0$  -th, or  $\, {\bf 1}$  -st elements from tuple.

**Question:** Why not use *plain tuples?* 

### Implementation: Types

First, lets look at the new Expr type

• No more Def



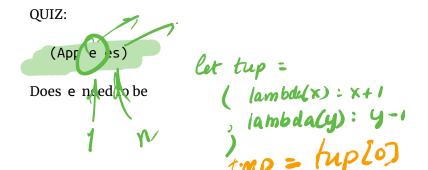
So we represent a **function-definition** as:



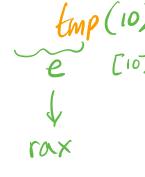
## Transforms: Tag

This is pretty straight forward (do it yourself)

## Transforms: ANF



- 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** 

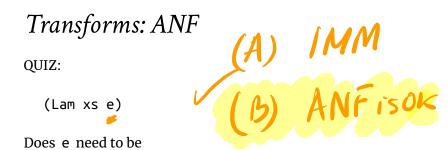
• Need the values to push on stack and make the call happen!

Just like function calls (in diamondback), except

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

cse131

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



- 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!)

cse131

```
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
                           = concatMap . go
           (Boolean {})
           (Number n l) = largeNumberErrors
   go _
   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
                                                       even in scope
   go vEnv (Tuple es _) = ...
   go vEnv (GetItem e1 e2 _) = ...
   go vEnv (App e) es _) = ?1 gos
   go vEnv (Lam xs e _) = ?2 ++ go ?3 e
 • How shall we implement ?1?
 • How shall we implement ?2?
 • How shall we implement ?3?
```

- x is a let-bound variable inside e.
- ullet x is a formal parameter in  $\, ullet$  , OR

A variable  $\,x\,$  is free inside an expression  $\,e\,$  if

• x is **not bound** inside e

For example consider the expression  $\, {\sf e} \,$  :

```
lambda (m):
    let t = m in
        n + t
```

- m, t are bound inside e, but,
- n is free inside e

## Computing Free Variables

Lets write a function to **compute** the set of free variables.

**Question** Why Set?

60 of 60