# *Branches and Binary Operators*

# *BOA: Branches and Binary Operators*

Next, lets add

- Branches ( `if` -expressions)
- Binary Operators ( `+` , `-` , etc.)

In the process of doing so, we will learn about

- **Intermediate Forms** → tags
- **Normalization** → ANF

COBRA = BOA + TYPES / RUNTIME CHECK + PRINTING

$(2+3)+(4-5)+6$

# *Binary Operations*

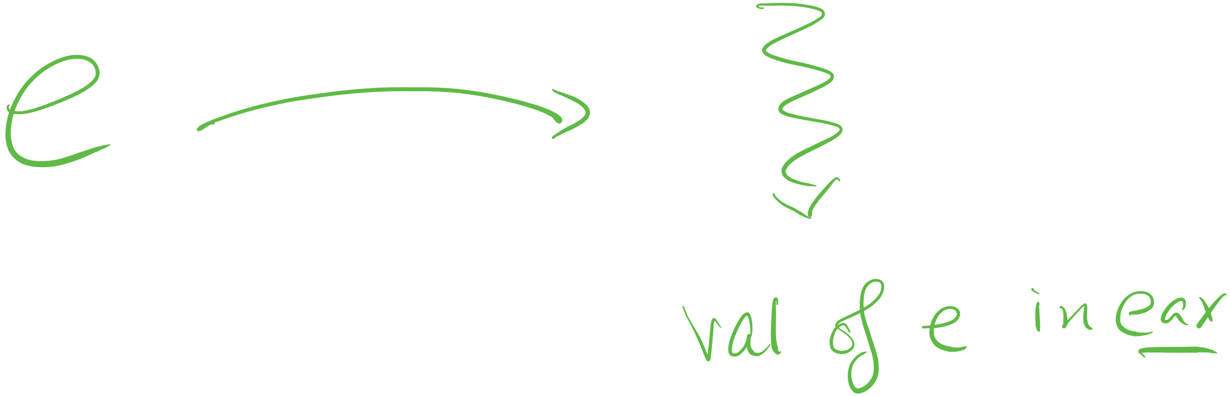
You know the drill.

1. Build intuition with **examples**,
2. Model problem with **types**,
3. Implement with **type-transforming-functions**,
4. Validate with **tests**.

# *Compiling Binary Operations*

Lets look at some expressions and figure out how they would get compiled.

- Recall: We want the result to be in `eax` after the instructions finish.




# *QUIZ*

What is the assembly corresponding to `33 - 10` ?

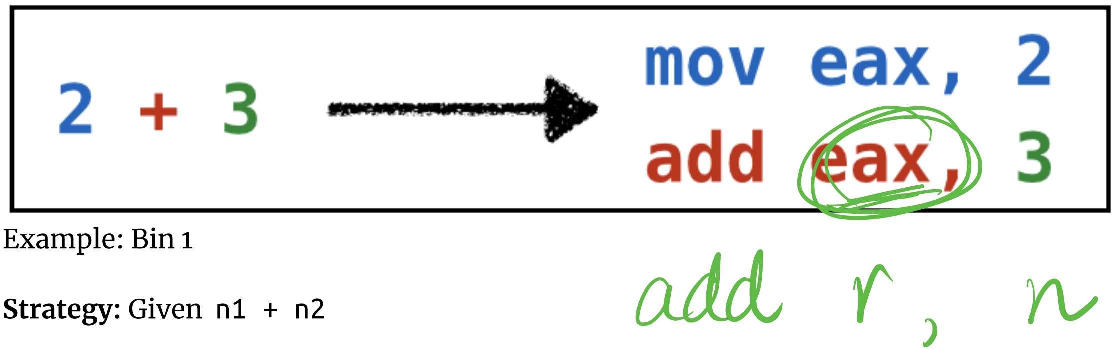
```
?1 eax, ?2
?3 eax, ?4
```

- **A.** `?1 = sub`, `?2 = 33`, `?3 = mov`, `?4 = 10`
- **B.** `?1 = mov`, `?2 = 33`, `?3 = sub`, `?4 = 10`
- **C.** `?1 = sub`, `?2 = 10`, `?3 = mov`, `?4 = 33`

- **D.** `?1 = mov`, `?2 = 10`, `?3 = sub`, `?4 = 33`

## *Example: Bin1*

Lets start with some easy ones. The source:




Example: Bin 1

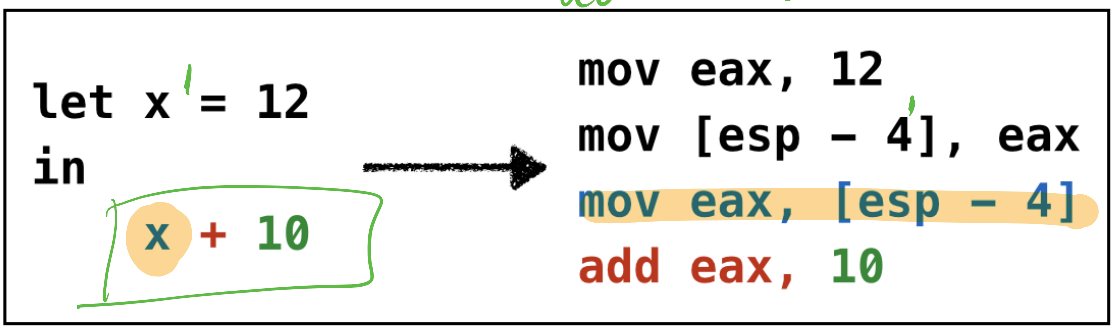
**Strategy:** Given `n1 + n2`

- Move `n1` into `eax`,

- Add `n2` to `eax`.

# *Example: Bin2*

What if the first operand is a variable?



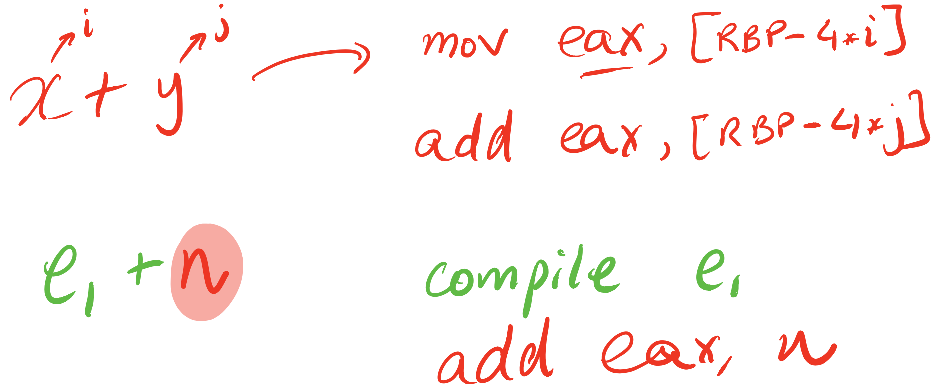

Example: Bin 2

var
const1 + const2
stick in eax
add using const2

Simple, just copy the variable off the stack into `eax`

**Strategy:** Given `x + n`

- Move `x` (from stack) into `eax`,
- Add `n` to `eax`.




## *Example: Bin3*

Same thing works if the second operand is a variable.

```
let x = 12          mov eax, 12
                    mov [esp - 4], eax
  , y = 18   --->   mov eax, 18
in
```

| | |
|---|---|
| `in` | `mov [esp - 8], eax` |
| `x + y` | `mov eax, [esp - 4]` |
| | `add eax, [esp - 8]` |

Example: Bin 3

Strategy: Given `x + n`

- Move `x` (from stack) into `eax`,
- Add `n` to `eax`.

## *QUIZ*

What is the assembly corresponding to `(10 + 20) * 30` ?

mov eax, 10
add eax, 20
mul eax, 30

```
mov eax, 10
?1  eax, ?2
?3  eax, ?4
```

- **A.** `?1 = add`, `?2 = 30`, `?3 = mul`, `?4 = 20`
- **B.** `?1 = mul`, `?2 = 30`, `?3 = add`, `?4 = 20`
- **C.** `?1 = add`, `?2 = 20`, `?3 = mul`, `?4 = 30`
- **D.** `?1 = mul`, `?2 = 20`, `?3 = add`, `?4 = 30`

# *Second Operand is Constant*

In general, to compile `e + n` we can do

```
   compile e
++              -- result of e is in eax
   [add eax, n]
```

# *Example: Bin4*

But what if we have *nested* expressions

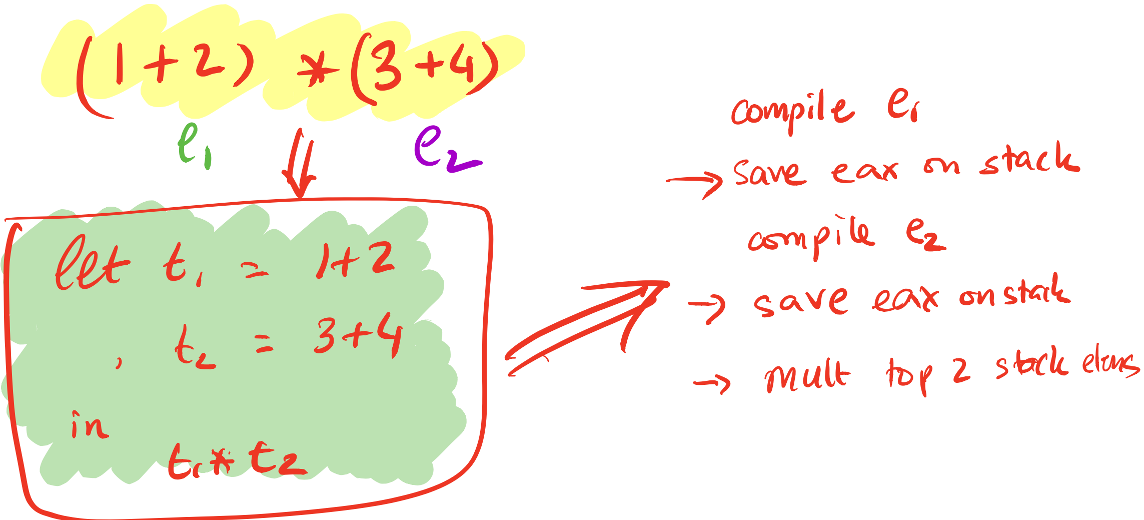
```
(1 + 2) * (3 + 4)
```

- Can compile `1 + 2` with result in `eax` ...
- .. but then need to *reuse* `eax` for `3 + 4`

Need to **save** `1 + 2` somewhere!

*Idea: How about use another register for* `3 + 4`*?*

But then what about `(1 + 2) * (3 + 4) * (5 + 6)` ? * In general, may need to *save* more sub-expressions than we have registers.




# *Idea: Immediate Expressions*

Why were `1 + 2` and `x + y` so easy to compile but `(1 + 2) * (3 + 4)` not?

As `1` and `x` are **immediate expressions**: their values don't require any computation!

- Either a **constant**, or,
- **variable** whose value is on the stack.

# *Idea: Administrative Normal Form (ANF)*

An expression is in **Administrative Normal Form (ANF)**

> *if all **primitive operations** have **immediate** arguments.*

**Primitive Operations:** Those whose values we *need* for computation to proceed.

- `v1 + v2`
- `v1 - v2`
- `v1 * v2`

# *QUIZ*

Is the following expression in ANF?

```
(1 + 2) * (4 - 3)
```

**A.** Yes, its ANF.

**B.** Nope, its not, because of `+`

**C.** Nope, its not, because of `*`

**D.** Nope, its not, because of `-`

**E.** Huh, WTF is ANF?

# *Conversion to ANF*

So, the below is *not* in ANF as `*` has *non-immediate* arguments

```
(1 + 2) * (3 + 4)
```

However, note the following variant *is* in ANF

```
let t1 = 1 + 2
  , t2 = 3 + 4
in
    t1 * t2
```

```
mov eax, 1
add eax, 2
? mov [RBP-4], eax
mov eax, 3
add eax, 4
? mov [RBP-8], eax
mul eax, ? RBP-4
mul eax, RBP-8
```

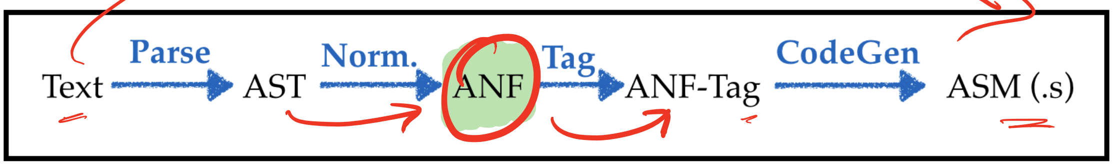
How can we compile the above code?

```
; TODO in class
```

# *Binary Operations: Strategy*

We can convert *any* expression to ANF

- By adding "temporary" variables for sub-expressions




Compiler Pipeline with ANF

- **Step 1**: Compiling ANF into Assembly
- **Step 2:** Converting Expressions into ANF

# *Types: Source*

Lets add binary primitive operators

```
data Prim2
  = Plus | Minus | Times
```

and use them to extend the source language:

```
data Expr a
  = ...
  | Prim2 Prim2  (Expr a) (Expr a) a
```

So, for example, `2 + 3` would be parsed as:

Prim2 Plus (Const 2) (Const 3)

```
Prim2 Plus (Number 2 ()) (Number 3 ()) ()
```

# *Types: Assembly*

Need to add X86 instructions for primitive arithmetic:

```
data Instruction
  = ...
  | IAdd Arg Arg
  | ISub Arg Arg
  | IMul Arg Arg
```

already there

# *Types: ANF*

We *can* define a separate type for ANF (try it!)

... but ...

*super tedious* as it requires duplicating a bunch of code.

Instead, lets write a *function* that describes **immediate expressions**

```
isImm :: Expr a -> Bool
isImm (Number _ _) = True
isImm (Var    _ _) = True
isImm _            = False
```

We can now think of **immediate** expressions as:

$\{ e : Expr \mid isImm\ e == True \}$

> *The subset of `Expr` such that `isImm` returns `True`*

# QUIZ

Similarly, lets write a function that describes **ANF** expressions

```
isAnf :: Expr a -> Bool
isAnf (Number  _     _) = True
isAnf (Var     _     _) = True
isAnf (Prim2 _ e1 e2 _) = _1
isAnf (If e1 e2 e3   _) = _2
isAnf (Let x e1 e2   _) = _3
```

What should we fill in for `_1` ?

```
{- A -} isAnf e1
{- B -} isAnf e2
{- C -} isAnf e1 && isAnf e2
{- D -} isImm e1 && isImm e2
{- E -} isImm e2
```

## *QUIZ*

Similarly, lets write a function that describes **ANF** expressions

```
isAnf :: Expr a -> Bool
isAnf (Number  _     _) = True
isAnf (Var     _     _) = True
isAnf (Prim1 _ e1 _)    = isAnf e1
isAnf (Prim2 _ e1 e2 _) = isImm e1 && isImm e2
isAnf (If e1 e2 e3   _) = _2       && isANF e2 && isANF e3
isAnf (Let x e1 e2   _) = isANF e1 && isANF e2
```

What should we fill in for `_2` ?

```
{- A -} isAnf e1
{- B -} isImm e1
{- C -} True
{- D -} False
```

We can now think of **ANF** expressions as:

> *The subset of `Expr` such that `isAnf` returns `True`*

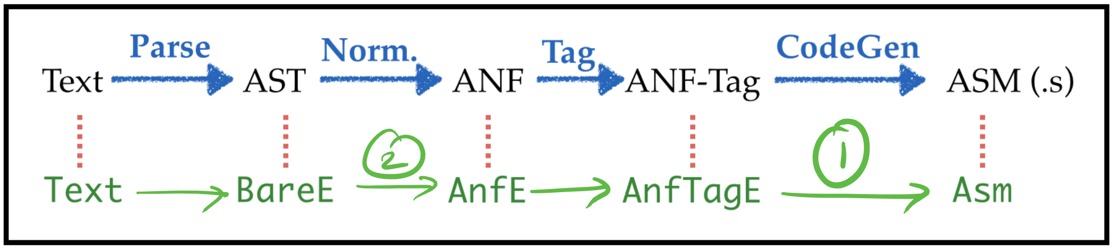
Use the above function to **test** our ANF conversion.

# *Types & Strategy*

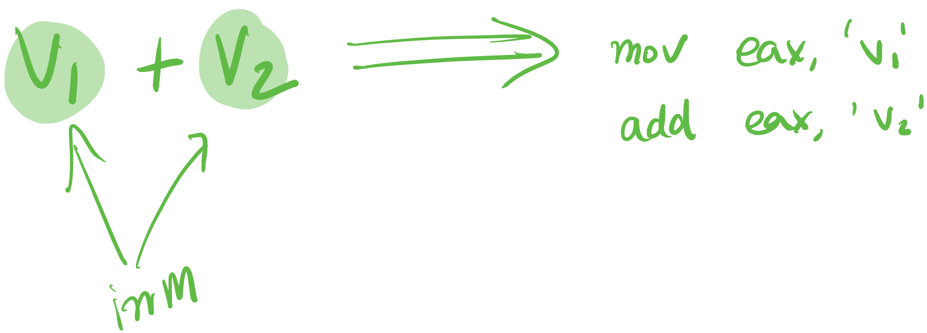
Writing the type aliases:

```haskell
type BareE   = Expr ()
type AnfE    = Expr ()  -- such that isAnf is True
type AnfTagE = Expr Tag -- such that isAnf is True
type ImmTagE = Expr Tag -- such that isImm is True
```

we get the overall pipeline:




Compiler Pipeline with ANF: Types



# *Transforms: Compiling* `AnfTagE` *to* `Asm`


Compiler Pipeline: ANF to ASM

The compilation from ANF is easy, lets recall our examples and strategy:

Strategy: Given `v1 + v2` (where `v1` and `v2` are **immediate expressions**)

- Move `v1` into `eax` ,
- Add `v2` to `eax` .

```
compile :: Env -> TagE -> Asm
compile env (Prim2 o v1 v2)
  = [ IMov      (Reg EAX) (immArg env v1)
    , (prim2 o) (Reg EAX) (immArg env v2)
    ]
```

where we have a helper to find the `Asm` variant of a `Prim2` operation

```
prim2 :: Prim2 -> Arg -> Arg -> Instruction
prim2 Plus  = IAdd
prim2 Minus = ISub
prim2 Times = IMul
```

and another to convert an *immediate expression* to an x86 argument:

```
immArg :: Env -> ImmTag -> Arg
immArg _   (Number n _) = Const n
immArg env (Var    x _) = RegOffset ESP i
  where
    i                   = fromMaybe err (lookup x env)
    err                 = error (printf "Error: Variable '%s' is unbo
und" x)
```

# *QUIZ*

Which of the below are in ANF ?

(A) in ANF

(B) NOT in ANF

```
{- 1 -} 2 + 3 + 4
```

X 2 + (3+4)

```
{- 2 -} let x = 12 in
          x + 1
```

✓

```
{- 3 -} let x = 12
          , y = x + 6
        in
          x + y
```

✓

```
{- 4 -} let x = 12
          , y = 18
          , t = x + y + 1
        in
          if t: 7 else: 9
```

X

- **A.** `1, 2, 3, 4`

- **B.** `1, 2, 3`
- **C.** `2, 3, 4`
- **D.** `1, 2`
- **E.** `2, 3`

# *Transforms: Compiling* `Bare` *to* `Anf`

Next lets focus on **A-Normalization** i.e. transforming expressions into ANF


Compiler Pipeline: Bare to ANF

EXPR ⟶ ANF

defs 1

$e_1 + e_2$

defs 2

(1+2) (3-4)

let $t_1 = 1+2$
$t_2 = 3-4$
in
$t_1 + t_2$

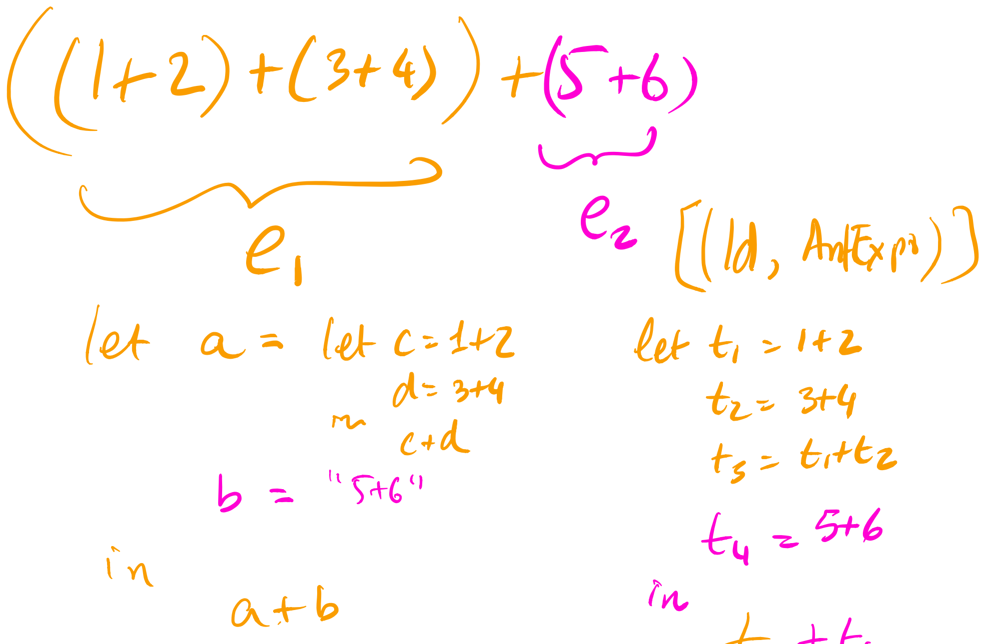
let defs 1
defs 2
in
$t_1 + t_2$

## A-Normalization

We can fill in the base cases easily

```
anf (Number n)      = Number n
anf (Var x)         = Var x
```

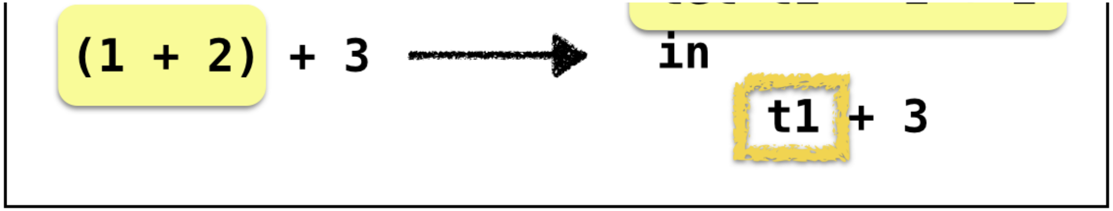
Interesting cases are the binary operations




## *Example: Anf-1*

Left operand is not immediate

```
let t1 = 1 + 2
```




Example: ANF 1

**Key Idea: Helper Function**

```
imm :: BareE -> ([(Id, AnfE)], ImmE)
```

`imm e` returns `([(t1, a1),...,(tn, an)], v)` where

- `ti, ai` are new temporary variables bound to ANF expressions
- `v` is an **immediate value** (either a constant or variable)

Such that `e` is *equivalent to*

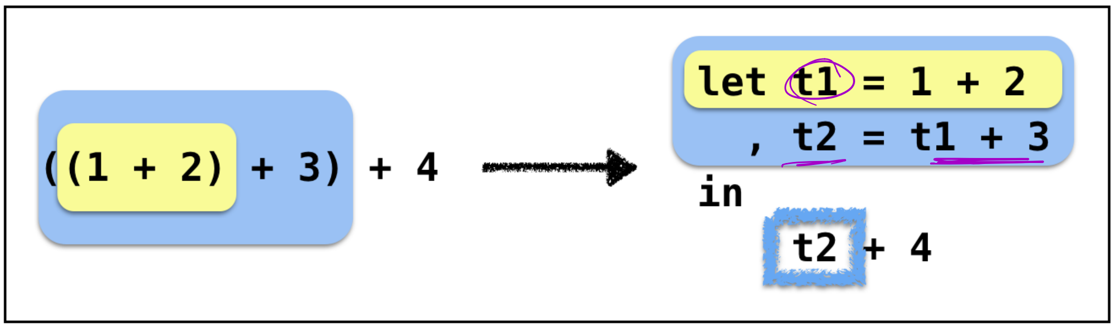
```
let t1 = a1
  , ...
  , tn = an
in
   v
```

Lets look at some more examples.

$e_1 + e_2 \longrightarrow ?$

## *Example: Anf-2*

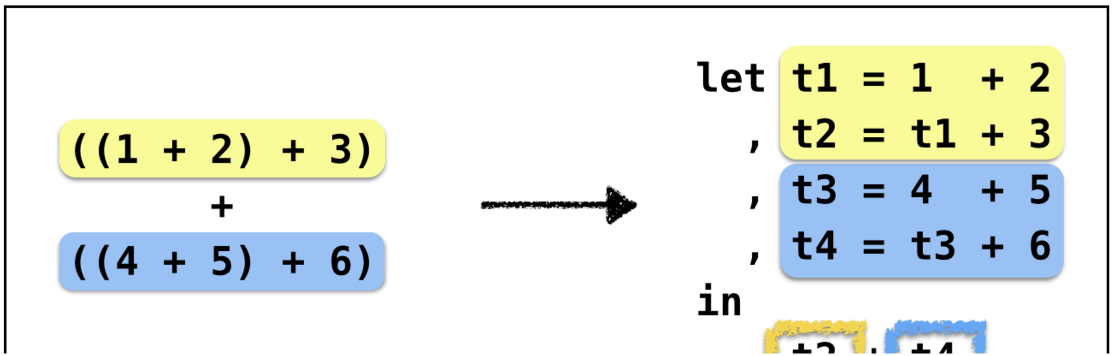
Left operand is not internally immediate




Example: ANF 2

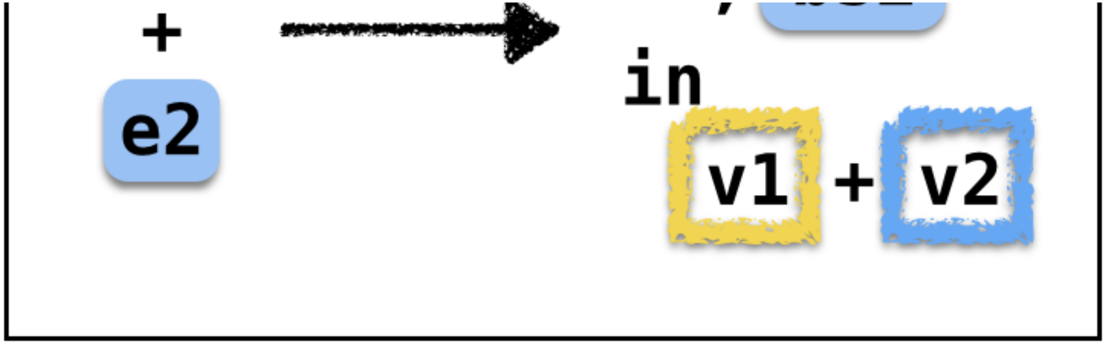
## *Example: Anf-3*

Both operands are not immediate




Example: ANF 3

## *ANF: General Strategy*




ANF Strategy

1. **Invoke** `imm` on both the operands
2. **Concat** the `let` bindings
3. **Apply** the binary operator to the immediate values

# *ANF Implementation: Binary Operations*

Lets implement the above strategy

```
anf (Prim2 o e1 e2) = lets (b1s ++ b2s)
                         (Prim2 o (Var v1) (Var v2))
  where
    (b1s, v1)      = imm e1
    (b2s, v2)      = imm e2
```

```
lets :: [(Id, AnfE)] -> AnfE -> AnfE
lets []          e' = e
lets ((x,e):bs) e' = Let x e (lets bs e')
```

Intuitively, `lets` *stitches* together a bunch of definitions:

```
lets [(x1, e1), (x2, e2), (x3, e3)] e
  ===> Let x1 e1 (Let x2 e2 (Let x3 e3 e))
```

# *ANF Implementation: Let-bindings*

For `Let` just make sure we recursively `anf` the sub-expressions.

```
anf (Let x e1 e2)   = Let x e1' e2'
  where
    e1'             = anf e1
    e2'             = anf e2
```

# *ANF Implementation: Branches*

Same principle applies to `If`

- use `anf` to recursively transform the branches.

```
anf (If e1 e2 e3) = If e1' e2' e3'
  where
    e1'           = anf e1
    e2'           = anf e2
    e3'           = anf e3
```

## *ANF: Making Arguments Immediate via* `imm`

The workhorse is the function

```
imm :: BareE -> ([(Id, AnfE)], ImmE)
```

which creates temporary variables to crunch an arbitrary `Bare` into an *immediate* value.

No need to create an variables if the expression is *already* immediate:

```
imm (Number n l) = ( [], Number n l )
imm (Id     x l) = ( [], Id     x l )
```

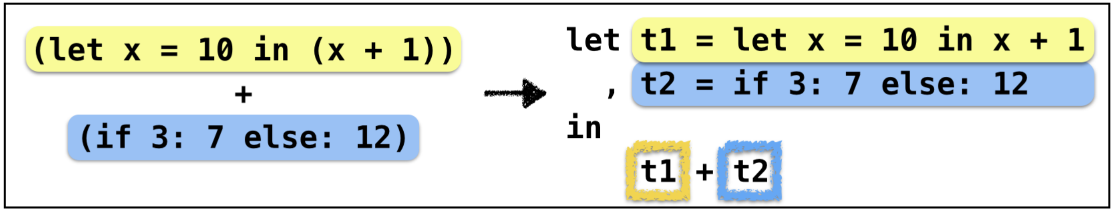
The tricky case is when the expression has a primitive operation:

```
imm (Prim2 o e1 e2) = ( b1s ++ b2s ++ [(t,  Prim2 o v1 v2)]
                      , Id t  )
  t                 = makeFreshVar ()
  (b1s, v1)         = imm e1
  (b2s, v2)         = imm e2
```

Oh, what shall we do when:

```
imm (If e1 e2 e3)   = ???
imm (Let x e1 e2)   = ???
```

Lets look at an example for inspiration.




Example: ANF 4

That is, simply

- `anf` the relevant expressions,
- bind them to a fresh variable.

```
imm e@(If _ _ _)  = immExp e
imm e@(Let _ _ _) = immExp e

immExp :: Expr -> ([(Id, AnfE)], ImmE)
immExp e = ([(t, e')], t)
  where
    e'   = anf e
    t    = makeFreshVar ()
```

# *One last thing: Whats up with `makeFreshVar` ?*

Wait a minute, what is this magic **FRESH** ?

How can we create **distinct** names out of thin air?

(Sorry, no "global variables" in Haskell…)

We will use a counter, but will **pass its value around**

> *Just like* `doTag`

```haskell
anf :: Int -> BareE -> (Int, AnfE)

anf i (Number n l)      = (i, Number n l)

anf i (Id     x l)      = (i, Id     x l)

anf i (Let x e b l)     = (i'', Let x e' b' l)
  where
    (i',  e')           = anf i e
    (i'', b')           = anf i' b

anf i (Prim2 o e1 e2 l) = (i'', lets (b1s ++ b2s) (Prim2 o e1' e2'
l))
  where
    (i' , b1s, e1')     = imm i  e1
    (i'', b2s, e2')     = imm i' e2

anf i (If c e1 e2 l)    = (i''', lets bs  (If c' e1' e2' l))
  where
    (i'  , bs, c')      = imm i   c
    (i'' ,     e1')     = anf i'  e1
    (i''',     e2')     = anf i'' e2
```

and

```haskell
imm :: Int -> AnfE -> (Int, [(Id, AnfE)], ImmE)

imm i (Number n l)        = (i  , [], Number n l)

imm i (Var x l)           = (i  , [], Var x l)

imm i (Prim2 o e1 e2 l) = (i''', bs, Var v l)
  where
    (i'  , b1s, v1)     = imm i  e1
    (i'' , b2s, v2)     = imm i' e2
    (i''', v)           = fresh i''
    bs                  = b1s ++ b2s ++ [(v, Prim2 o v1 v2 l)]

imm i e@(If _ _ _  l)   = immExp i e

imm i e@(Let _ _ _ l)   = immExp i e

immExp :: Int -> BareE -> (Int, [(Id, AnfE)], ImmE)
immExp i e l  = (i'', bs, Var v ())
  where
    (i' , e') = anf i e
    (i'', v)  = fresh i'
    bs        = [(v, e')]
```

where now, the `fresh` function returns a *new counter* and a variable

```
fresh :: Int -> (Int, Id)
fresh n = (n+1, "t" ++ show n)
```

**Note** this is super clunky. There *is* a really slick way to write the above code without the clutter of the `i` but thats too much of a digression, but feel free to look it up yourself (https://cseweb.ucsd.edu/classes/wi12/cse230-a/lectures/monads.html)

# *Recap and Summary*

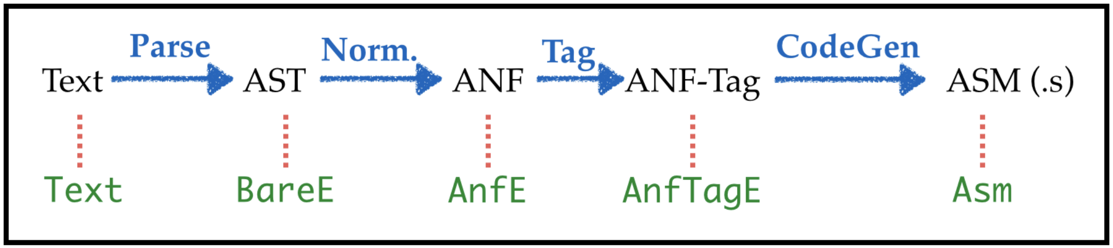
Just created `Boa` with

- Branches ( `if` -expressions)
- Binary Operators ( `+` , `-` , etc.)

In the process of doing so, we will learned about

- **Intermediate Forms**
- **Normalization**

Specifically,




Compiler Pipeline with ANF

---

(https://ucsd-cse131.github.io/sp21/feed.xml)

(https://twitter.com/ranjitjhala)

(https://plus.google.com/u/0/106612421534244742464)

(https://github.com/ucsd-cse131/sp21)