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**
- **Normalization**

Branches

Lets start first with branches (conditionals).

We will stick to our recipe of:

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

Examples

First, lets look at some examples of what we mean by branches.

- For now, lets treat `0` as "false" and non-zero as "true"

Example: `If1`

```
if 10:
  22
else:
  sub1(0)
```

- Since `10` is *not* `0` we evaluate the “then” case to get `22`

Example: `If2`

```
if sub(1):
  22
else:
  sub1(0)
```

- Since `sub(1)` *is* `0` we evaluate the "else" case to get `-1`

QUIZ: *`If3`*

`if-else` is also an *expression* so we can nest them:

What should the following evaluate to?

```
let x = if sub(1):
          22
        else:
          sub1(0)
in
  if x:
    add1(x)
  else:
    999
```

- **A.** `999`
- **B.** `0`
- **C.** `1`
- **D.** `1000`
- **E.** `-1`

Control Flow in Assembly

To compile branches, we will use **labels**, **comparisons** and **jumps**

Labels

```
our_code_label:
  ...
```

Labels are *"landmarks"* - from which execution (control-flow) can be *started*, or - to which it can be *diverted*

Comparisons

```
cmp a1, a2
```

- Perform a (numeric) **comparison** between the values `a1` and `a2`, and
- Store the result in a special **processor flag**

Jumps

```
jmp LABEL     # jump unconditionally (i.e. always)
je  LABEL     # jump if previous comparison result was EQUAL
jne LABEL     # jump if previous comparison result was NOT-EQUAL
```

Use the result of the **flag** set by the most recent `cmp` * To *continue execution* from the given `LABEL`

QUIZ

Which of the following is a valid x86 encoding of

```
if 10:
  22
else
  33
```

A B C D

A

```
  mov eax, 10
  cmp eax, 0
  je  if_false
if_true:
  mov eax, 22
  jmp if_exit
if_false:
  mov eax, 33
if_exit:
```

B

```
  mov eax, 10
  cmp eax, 0
  je  if_false
if_true:
  mov eax, 22

if_false:
  mov eax, 33
if_exit:
```

C

```
  mov eax, 10
  cmp eax, 0
  je  if_true
if_true:
  mov eax, 22
  jmp if_exit
if_false:
  mov eax, 33
if_exit:
```

D

```
  mov eax, 10
  cmp eax, 0
  je  if_true
if_true:
  mov eax, 22

if_false:
  mov eax, 33
if_exit:
```

QUIZ: Compiling if-else

Strategy

To compile an expression of the form

```
if eCond:
  eThen
else:
  eElse
```

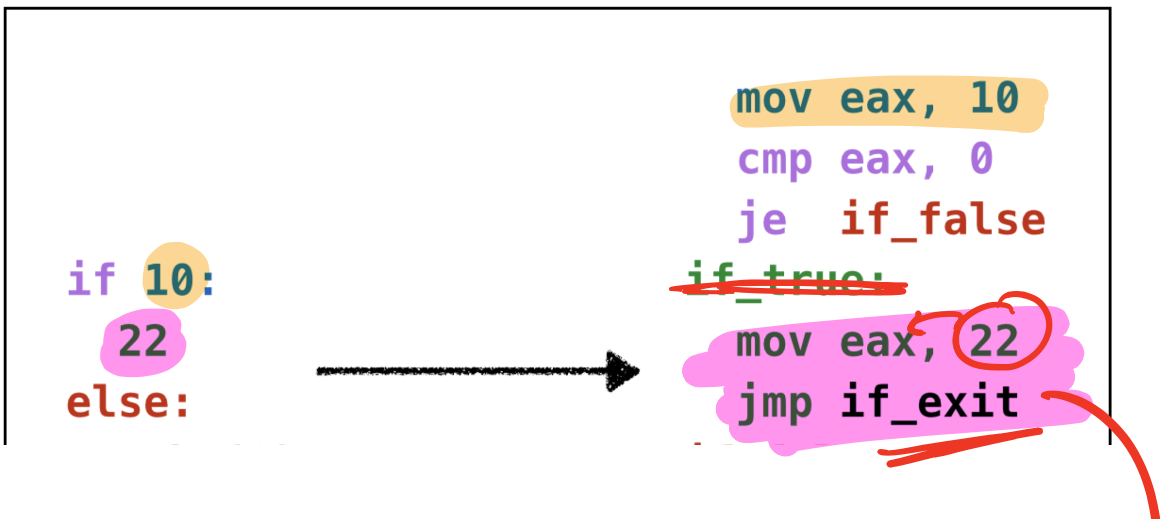
We will:

1. Compile `eCond`
2. Compare the result (in `eax`) against `0`
3. Jump if the result *is zero* to a **special** `"IfFalse"` label
    - At which we will evaluate `eElse` ,
    - Ending with a special `"IfExit"` label.
4. (Otherwise) continue to evaluate `eTrue`
    - And then jump (unconditionally) to the `"IfExit"` label.

Example: If-Expressions to `Asm`

Lets see how our strategy works by example:

Example: if1

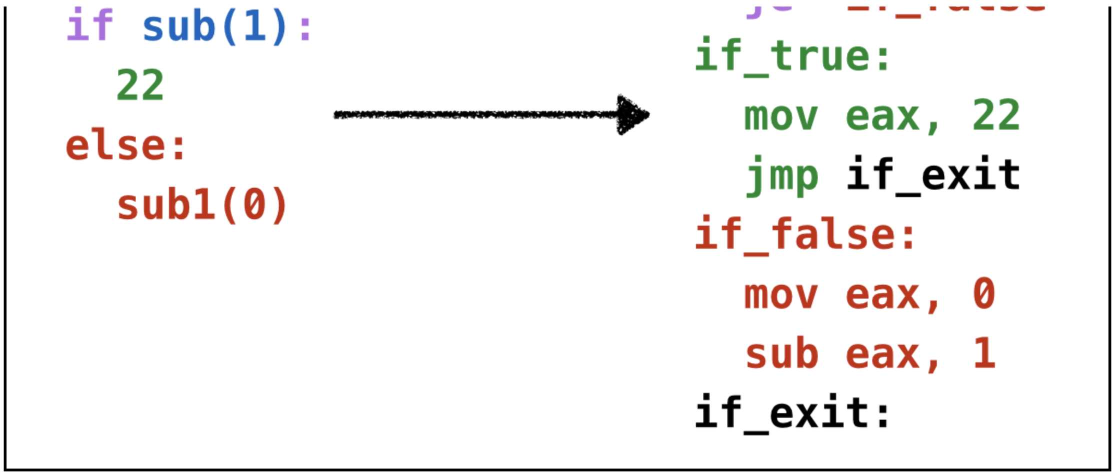


```
sub1(0)                         if_false:
                                  mov eax, 0
                                  sub eax, 1
                                if_exit:
```

Example: if1

Example: if2

```
                                mov eax, 1
                                sub eax, 1
                                cmp eax, 0
                                je  if_false
```



Example: if2

Example: if3

```
let x = if 10:
          22
        else:
          0
in
  if x:
    55
  else:
    999
```

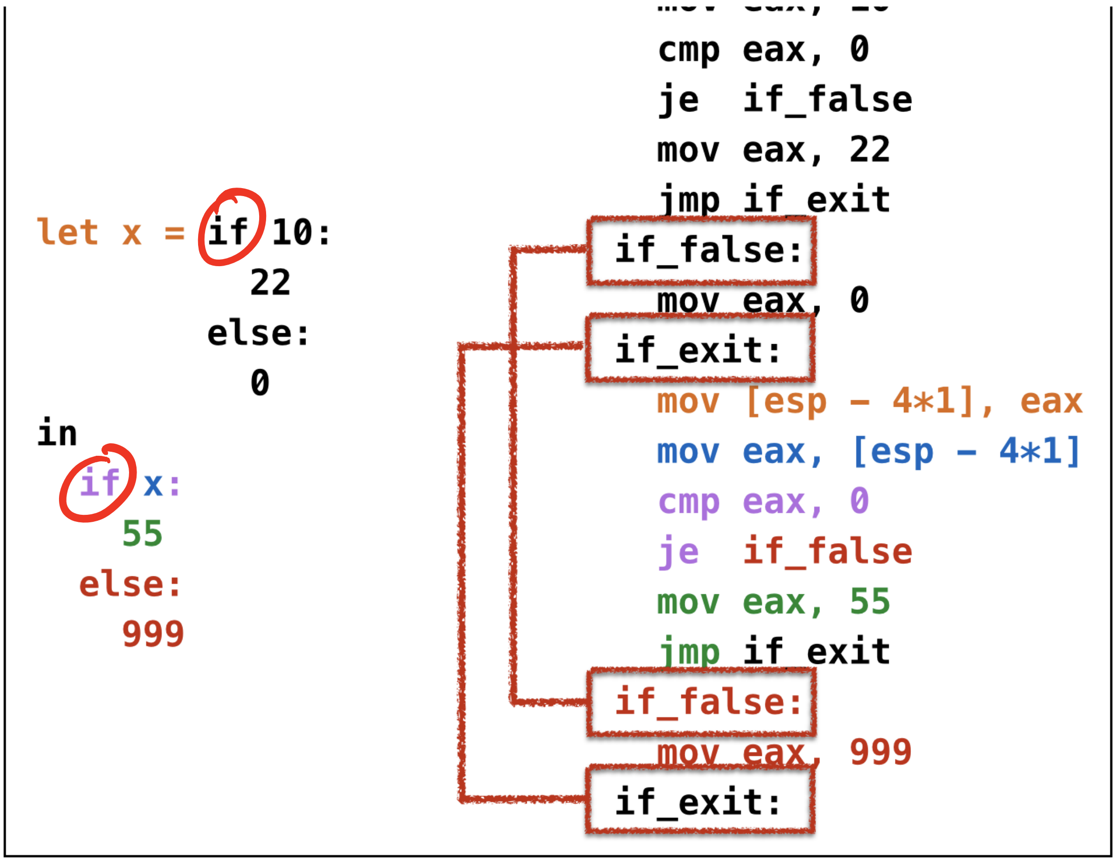
```
    cmp eax, 0
    je  if_false
    mov eax, 22
    jmp if_exit
  if_false:
    mov eax, 0
  if_exit:
    mov [esp - 4*1], eax
    mov eax, [esp - 4*1]
    cmp eax, 0
    je  if_false
    mov eax, 55
    jmp if_exit
  if_false:
    mov eax, 999
  if_exit:
```

Example: if3

Oops, **cannot reuse labels** across if-expressions!

- Can't use same label in two places (invalid assembly)

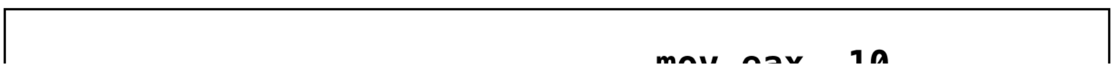
```
    mov eax, 10
```



Example: if3 wrong

Oops, need **distinct labels** for each branch!

- Require **distinct tags** for each `if-else` expression



```
let x = if 10:
          1 22
        else:
          0
in
  if x:
  2 55
  else:
    999
```

```
  mov eax, 10
  cmp eax, 0
  je  if_1_false
  mov eax, 22
  jmp if_1_exit
if_1_false:
  mov eax, 0
if_1_exit:
  mov [esp - 4*1], eax
  mov eax, [esp - 4*1]
  cmp eax, 0
  je  if_2_false
  mov eax, 55
  jmp if_2_exit
if_2_false:
  mov eax, 999
if_2_exit:
```

ICMP → cmp eax, 0
IJeq → je if_2_false
IJmp → jmp if_2_exit
Label → if_2_false:

Example: if3 tagged

Types: Source

Lets modify the *Source Expression* to add `if-else` expressions

```
data Expr a
  = Number Int                        a
  | Add1   (Expr a)                   a
  | Sub1   (Expr a)                   a
  | Let    Id (Expr a) (Expr a)       a
  | Var    Id                         a
  | If     (Expr a) (Expr a) (Expr a) a
```

Polymorphic tags of type `a` for each sub-expression

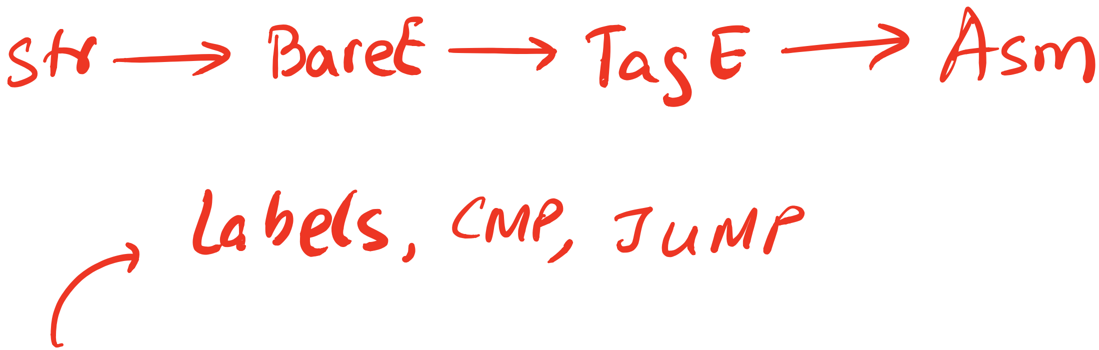
- We can have *different types* of tags
- e.g. Source-Position information for error messages

Lets define a name for `Tag` (just integers).

```
type Tag   = Int
```

We will now use:

```haskell
type BareE = Expr ()      -- AST after parsing
type TagE  = Expr Tag     -- AST with distinct tags
```



Types: Assembly

Now, lets extend the *Assembly* with labels, comparisons and jumps:

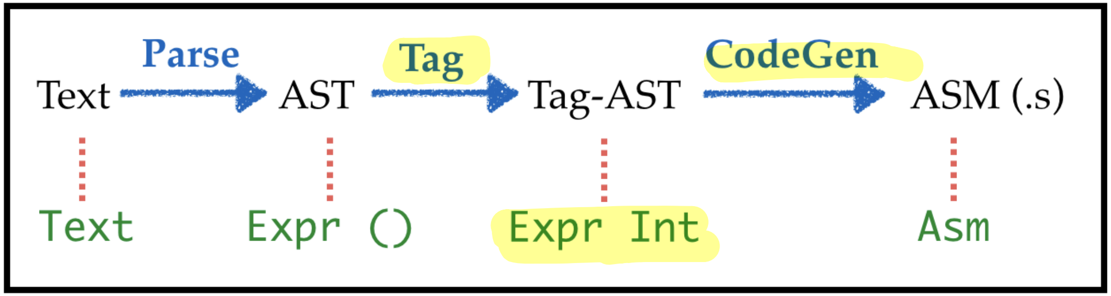
```haskell
data Label
  = BranchFalse Tag
  | BranchExit  Tag
```

```haskell
data Instruction
  = ...
  | ICmp   Arg   Arg   -- Compare two arguments
  | ILabel Label       -- Create a label
  | IJmp   Label       -- Jump always
  | IJe    Label       -- Jump if equal
  | IJne   Label       -- Jump if not-equal
```

Transforms

We can't expect *programmer* to put in tags (yuck.)

- Lets squeeze in a `tagging` transform into our pipeline



Adding Tagging to the Compiler Pipeline

Transforms: Parse

Just as before, but now puts a dummy `()` into each position

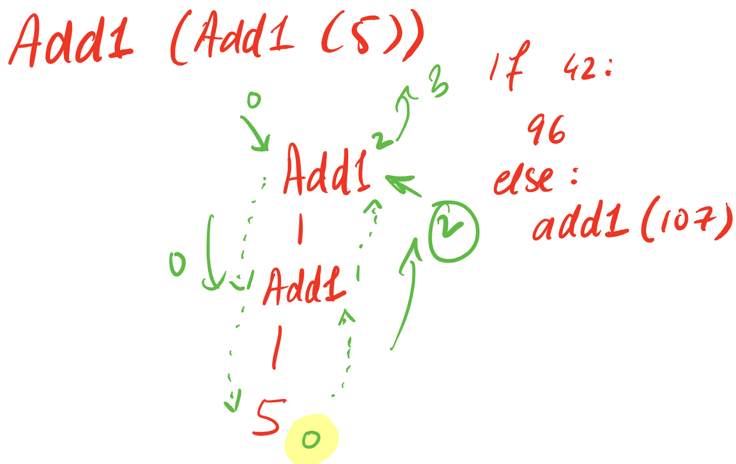
```
λ> let parseStr s = fmap (const ()) (parse "" s)

λ> let e = parseStr "if 1: 22 else: 33"

λ> e
If (Number 1 ()) (Number 22 ()) (Number 33 ()) ()

λ> label e
If (Number 1 ((),0)) (Number 22 ((),1)) (Number 33 ((),2)) ((),3)
```

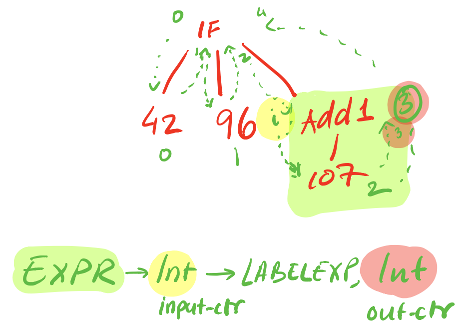
Transforms: Tag

The key work is done by `doTag i e`

1. Recursively walk over the `BareE` named `e` starting tagging at counter `i`

2. Return a pair `(i', e')` of *updated counter* `i'` and tagged expression `e'`





QUIZ

```
doTag :: Int -> BareE -> (Int, TagE)
doTag i (Number n _)     = (i + 1 , Number n i)
doTag i (Var    x _)     = (i + 1 , Var    x i)
doTag i (Let x e1 e2 _) = (_2    , Let x e1' e2' i2)
  where
    (i1, e1')            = doTag i  e1
    (i2, e2')            = doTag _1 e2
```

What expressions shall we fill in for `_1` and `_2` ?

```
{- A -}   _1 = i
          _2 = i + 1

{- B -}   _1 = i
          _2 = i1 + 1

{- C -}   _1 = i
          _2 = i2 + 1

{- D -}   _1 = i1
          _2 = i2 + 1

{- E -}   _1 = i2
          _2 = i1 + 1
```

(**ProTip:** Use `mapAccumL`)

We can now tag the whole program by

- Calling `doTag` with the initial counter (e.g. `0`),
- Throwing away the final counter.

```
tag :: BareE -> TagE
tag e = e'  where  (_, e') = doTag 0 e
```

IF e1 e2 e3 i

compile e1
cmp EAX, 0
jeq "FALSE-LABEL-i"
compile e2
jump "EXIT-LABEL-i"
FALSE-LABEL-i:
compile e3
EXIT-LABEL-i:

Transforms: Code Generation

Now that we have the tags we lets implement our compilation strategy

```
compile env (If eCond eTrue eFalse i)
  =   compile env eCond ++          -- compile `eCond`
    [ ICmp (Reg EAX) (Const 0)      -- compare result to 0
    , IJe (BranchFalse i)           -- if-zero then jump to 'False'-b
lock
    ]
   ++ compile env eTrue  ++         -- code for `True`-block
    [ IJmp   lExit      ]           -- jump to exit (skip `False`-blo
ck!)
   ++
      ILabel (BranchFalse i)        -- start of `False`-block
   : compile env eFalse ++          -- code for `False`-block
    [ ILabel (BranchExit i) ]       -- exit
```

Recap: Branches

- `Tag` each sub-expression,
- Use tag to generate control-flow labels implementing branch.

Lesson: Tagged program representation simplifies compilation…

- Next: another example of how intermediate representations help.

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

Binary Operations

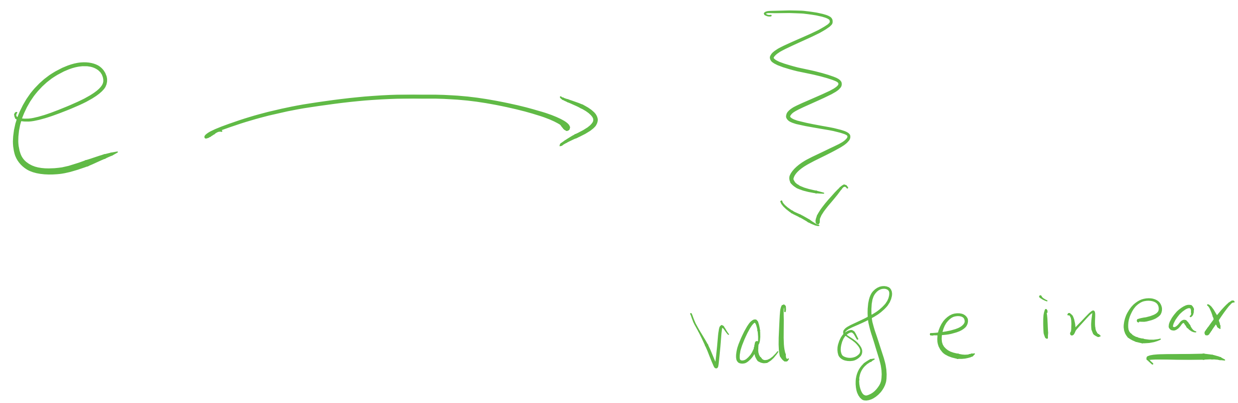
You know the drill.

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
```

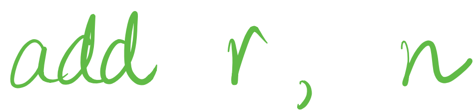
- **A.** `?1 = sub`, `?2 = 33`, `?3 = mov`, `?4 = 10`
- **B.** `?1 = mov`, `?2 = 33`, `?3 = sub`, `?4 = 10`
- **C.** `?1 = sub`, `?2 = 10`, `?3 = mov`, `?4 = 33`

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

Example: Bin1

Lets start with some easy ones. The source:

Example: Bin 1



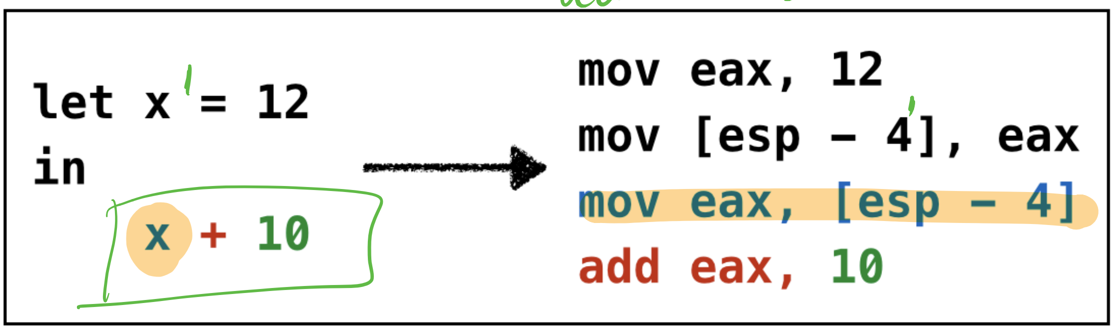
Strategy: Given `n1 + n2`

- Move `n1` into `eax`,

- Add `n2` to `eax`.

Example: Bin2

What if the first operand is a variable?



Example: Bin 2

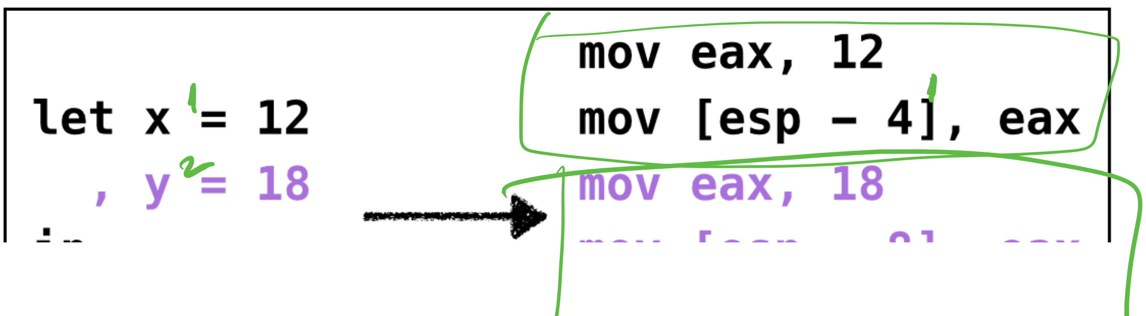
Simple, just copy the variable off the stack into `eax`

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.



```
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`?

02-boa will be due FRI 4/16

```
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