Data on the Heap

num
bool
x char
x double

Next, lets add support for

- **Data Structures**

In the process of doing so, we will learn about

- **Heap Allocation**
- **Run-time Tags**
- High-order Func (Closures)

<env, code>

Creating Heap Data Structures

We have already support for *two* primitive data types

```
data Ty
  = TNumber      -- e.g. 0,1,2,3,...
  | TBoolean     -- e.g. true, false
```

we could add several more of course, e.g.

- `Char`
- `Double` or `Float`

etc. (you should do it!)

However, for all of those, the same principle applies, more or less

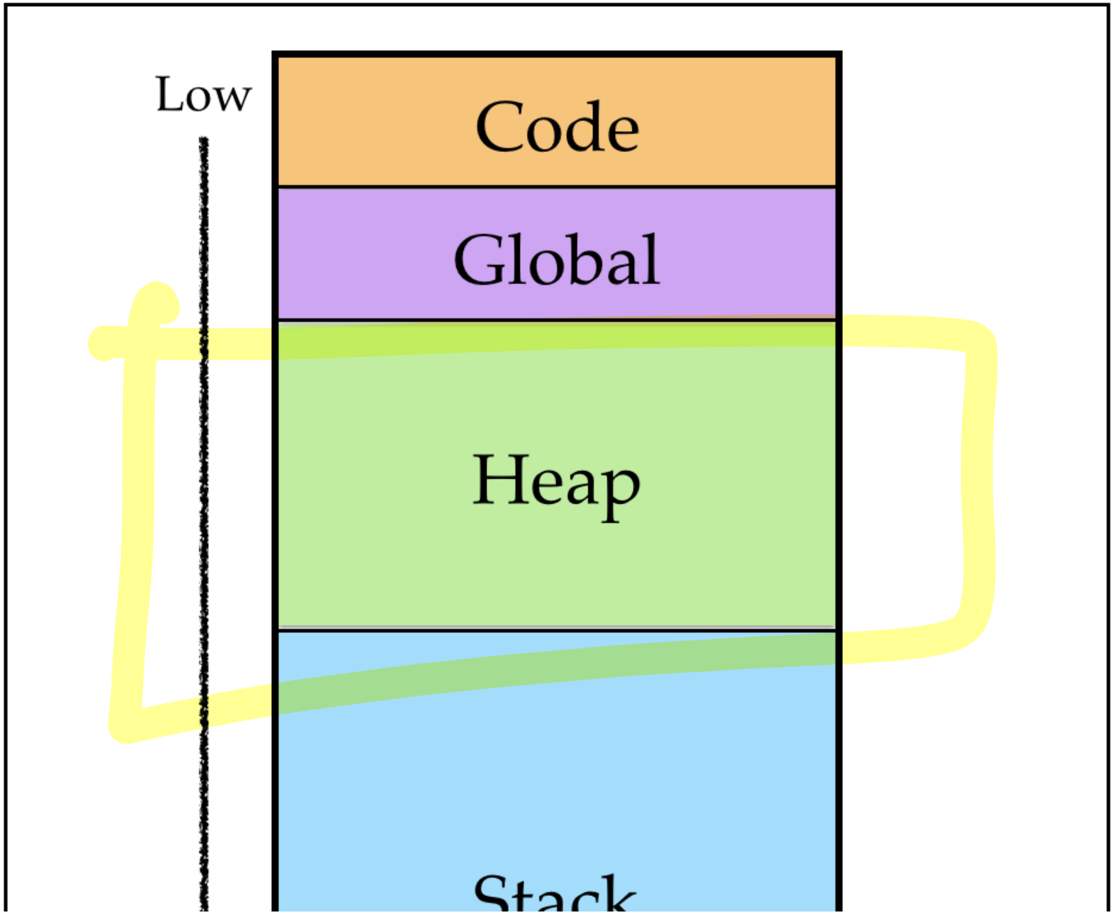
- As long as the data fits into a single word (8-bytes)

Instead, lets learn how to make **unbounded data structures**

- Lists
- Trees
- ...

which require us to put data on the **heap**

> *not just the stack that we've used so far.*



Stack vs. Heap

Pairs

While our *goal* is to get to lists and trees, the journey of a thousand miles begins with a single step…

lec

Assignment

So! we will *begin* with the humble **pair**.

construct

(2, 3)

length=2

(1, (2, 3))

e[0] e[1]

e[0]

(1, 2, 3, 7, (9, 12))

"length" = ?

storage

e[i]

Pairs: Semantics (Behavior)

First, lets ponder what exactly we're trying to achieve.

We want to enrich our language with *two* new constructs:

- **Constructing** pairs, with a new expression of the form `(e0, e1)` where `e0` and `e1` are expressions.
- **Accessing** pairs, with new expressions of the form `e[0]` and `e[1]` which

e[0] e[1]

evaluate to the first and second element of the tuple `e` respectively.

For example,

```
let t = (2, 3) in
  t[0] + t[1]
```

should evaluate to `5`.

Strategy

Next, lets informally develop a strategy for extending our language with pairs, implementing the above semantics. We need to work out strategies for:

1. **Representing** pairs in the machine's memory,

$(e_0, e_1) \longrightarrow$ <asm>

e[0] → (asm)

2. **Constructing** pairs (i.e. implementing `(e0, e1)` in assembly),
3. **Accessing** pairs (i.e. implementing `e[0]` and `e[1]` in assembly).

1. Representation

Recall that we represent all values: (05-cobra.md/#option-2-use-a-tag-bit)

- `Number` like `0` , `1` , `2` …
- `Boolean` like `true` , `false`

64

as a **~~single~~ word** either

- 8 bytes on the stack, or
- a single register `rax` , `rbx` etc.

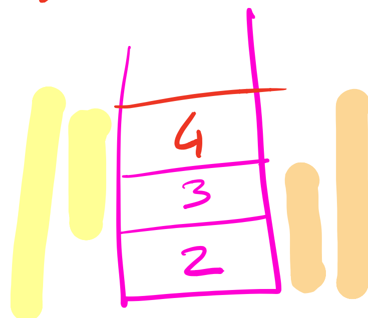
EXERCISE

What kinds of problems do you think might arise if we represent a pair $\boxed{\texttt{(2, 3)}}$ on the *stack* as:

```
|       |
-------
|   3   |
-------
|   2   |
-------
|  ...  |
-------
```

let t = ((2, 3), 4) in

⋮



t = ((2, 3), 4)

t = (2, (3, 4))

t[0][0]

1, 2, 3, 4, 5

cons(1, cons(2, cons(3, cons(4, nil()))))

(1, (2, (3, (4, false))))

(1,2) 3

(e0, e1)

e[0] , e[1]

def tail(l):
  l[1]

def isNil(l):
  l == False

def nil():
  false

def cons(h,t):
  (h,t)

build list

def range(lo, hi):
  if lo<hi:
    cons(lo, range(lo+1, hi))
  else:
    nil()

def length(l):
  if isNil(l):
    0
  else:
    1+length(tail(l))

l = range(0, 100)
length(l)

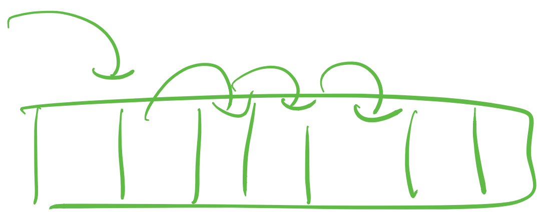
QUIZ

How many words would we need to store the tuple

(3, (4, 5))

1. ~~1 word~~
2. 2 words
3. 3 words
4. 4 words
5. 5 words

l

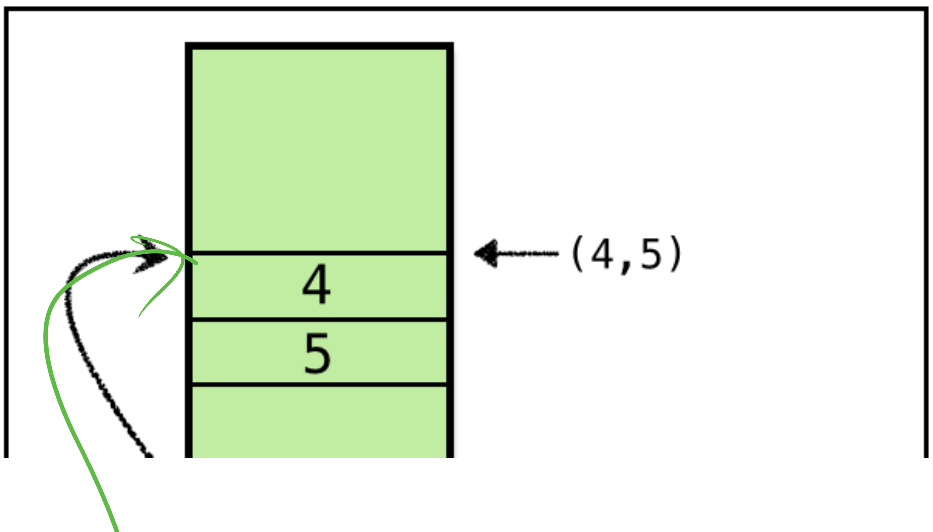


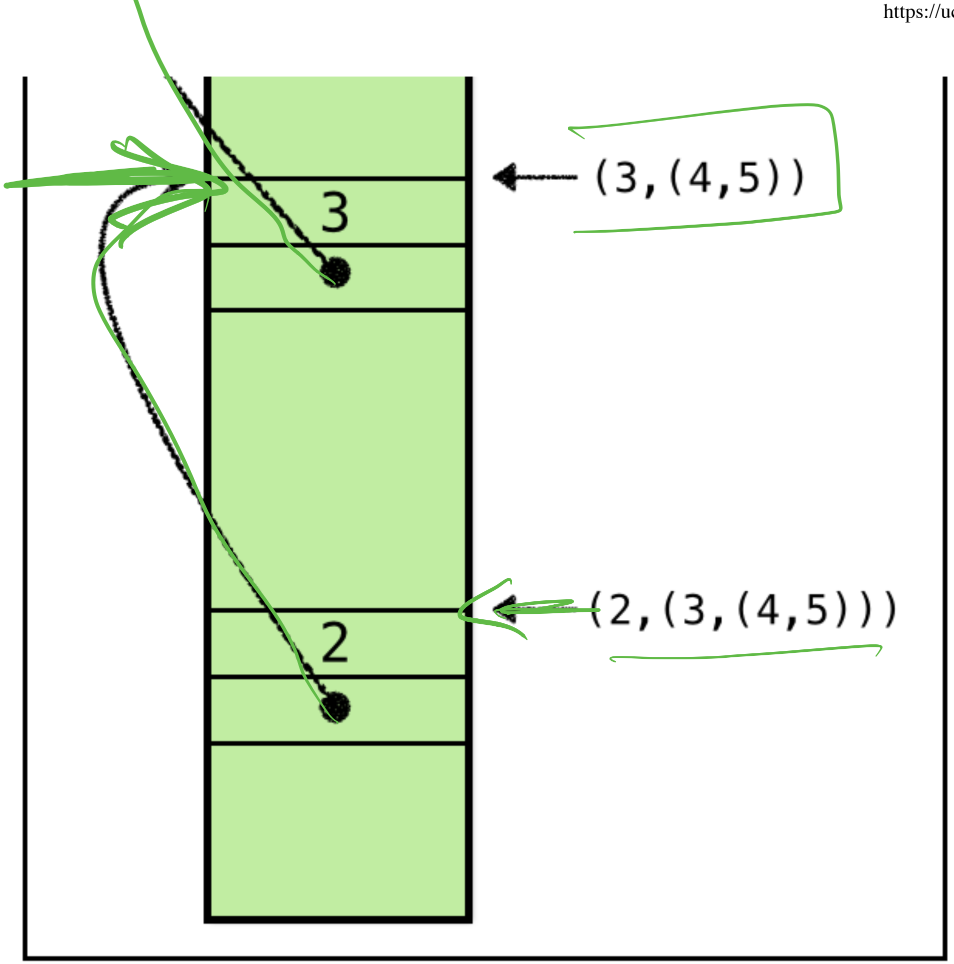
tup

Pointers

> *Every problem in computing can be solved by adding a level of indirection.*

We will **represent a pair** by a **pointer** to a block of **two adjacent words** of memory.





Pairs on the heap

The above shows how the pair `(2, (3, (4, 5)))` and its sub-pairs can be stored in the **heap** using pointers.

`(4, 5)` is stored by adjacent words storing

- `4` and
- `5`

`(3, (4, 5))` is stored by adjacent words storing

- `3` and
- a **pointer** to a heap location storing `(4, 5)`

`(2, (3, (4, 5)))` is stored by adjacent words storing

- `2` and
- a **pointer** to a heap location storing `(3, (4, 5))`.

A Problem: Numbers vs. Pointers?

How will we tell the difference between *numbers* and *pointers*?

That is, how can we tell the difference between

1. the *number* 5 and
2. a *pointer* to a block of memory (with address `5`)?

Each of the above corresponds to a *different* tuple

1. `(4, 5)` or
2. `(4, (...))`.

so its pretty crucial that we have a way of knowing *which* value it is.

t = (1, (2, 3))

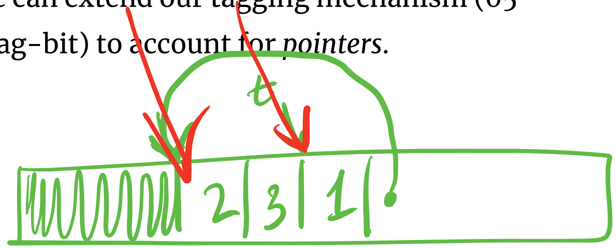
t = 3

t = false

Tagging Pointers

As you might have guessed, we can extend our tagging mechanism (05-cobra.md/#option-2-use-a-tag-bit) to account for *pointers*.

Type	LSB
`number`	xx0
`boolean`	111
`pointer`	001



t = (1, (2,3))

Pointers are 8-byte aligned

That is, for

- `number` the **last bit** will be `0` (as before),
- `boolean` the **last 3 bits** will be `111` (as before), and
- `pointer` the **last 3 bits** will be `001`.

(We have 3-bits worth for tags, so have wiggle room for other primitive types.)

Address Alignment

As we have a **3 bit tag**

- leaving **64 - 3 = 61 bits** for the actual address

So actual addresses, written in binary, omitting trailing zeros, are of the form

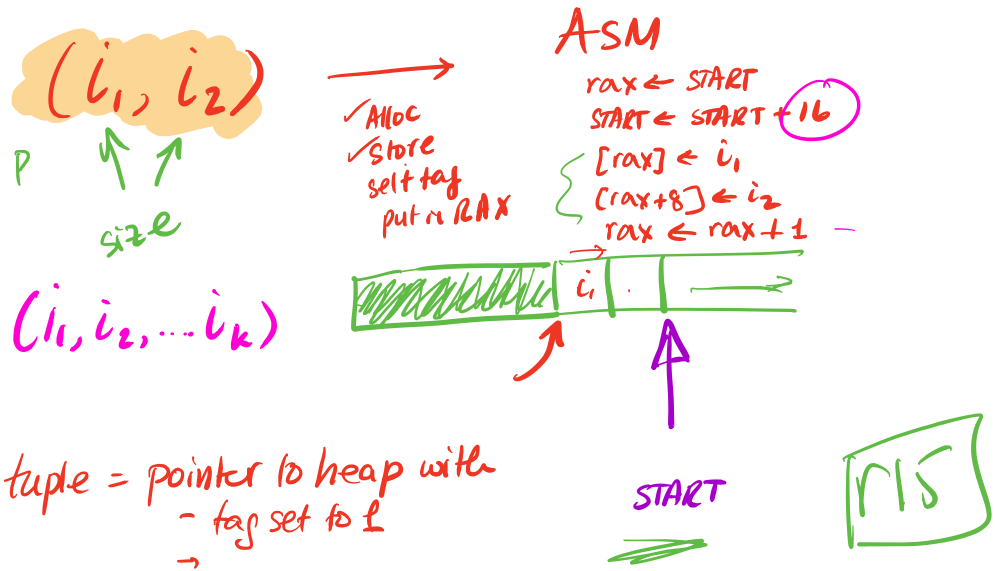
Binary	Decimal
`0b00000000`	0
`0b00001000`	8
`0b00010000`	16
`0b00011000`	24
`0b00100000`	32
...	

That is, the addresses are **8-byte aligned**.

Which is great because at each address, we have a pair, i.e. a **2-word = 16-byte block**, so the *next* allocated address will *also* fall on an 8-byte boundary.

START

- But ... what if we had 3-tuples? or 5-tuples? ...



2. Construction

Next, lets look at how to implement pair **construction** that is, generate the assembly for expressions like:

```
(e1, e2)
```

To **construct** a pair `(e1, e2)` we

1. **Allocate** a new 2-word block, and getting the starting address at `rax` ,
2. **Copy** the value of `e1` (resp. `e2`) into `[rax]` (resp. `[rax + 8]`).

3. **Tag** the last bit of `rax` with `1`.

The resulting `eax` is the **value of the pair**

- The *last step* ensures that the value carries the proper tag.

ANF will ensure that `e1` and `e2` are immediate expressions (04-boa.md/#idea-immediate-expressions)

- will make the second step above straightforward.

EXERCISE How will we do ANF conversion for `(e1, e2)`?

Allocating Addresses

Lets use a **global** register `r15` to maintain the address of the **next free block** on the heap.

Every time we need a *new* block, we will:

1. Copy the current `r15` into `rax`

- Set the last bit to `1` to ensure proper tagging.
- `rax` will be used to fill in the values

2. Increment the value of `r15` by `16`

- Thus *allocating* 8 bytes (= 2 words) at the address in `rax`

Note that addresses stay 8-byte aligned (last 3 bits = 0) if we

- *Start* our blocks at an 8-byte boundary, and
- *Allocate* 16 bytes at a time,

NOTE: Your assignment will have *blocks of varying sizes*

- You will have to *maintain* the 8-byte alignment by *padding*

Example: Allocation

In the figure below, we have

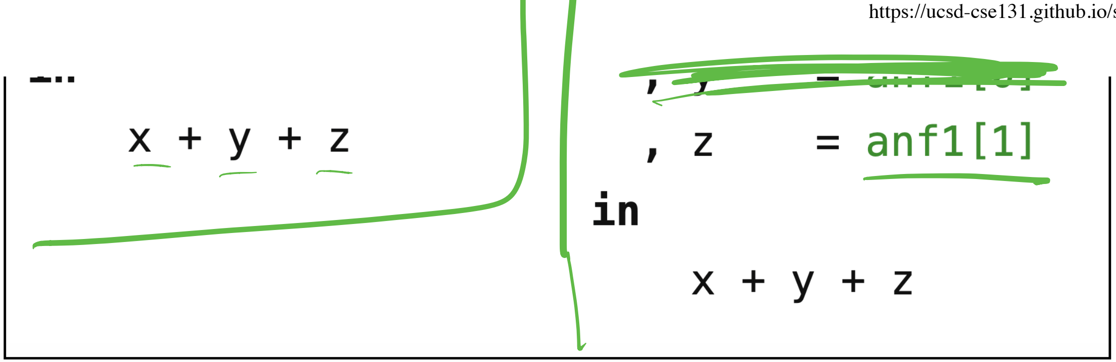
- a source program on the left,
- the ANF equivalent next to it.

Source

```
let p = (3, (4, 5))
  , x = p[0]        =3
  , y = p[1][0]     =4
  , z = p[1][1]     =5
in
```

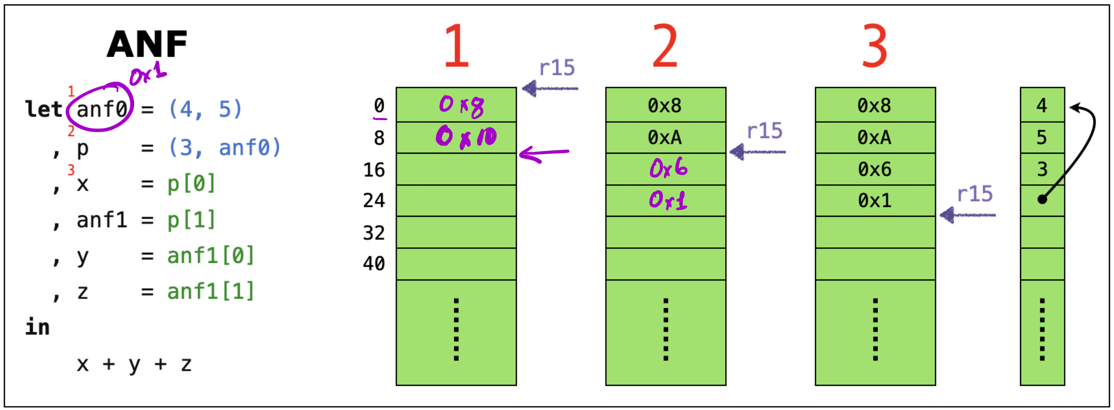
ANF

```
    0
let anf0 = (4, 5)
1
  , p    = (3, anf0)
2
  , x    = p[0]
  , anf1 = p[1]
  y      = anf1[0]
```



Example of Pairs

The figure below shows the how the heap and `r15` evolve at points 1, 2 and 3:



Allocating Pairs on the Heap

QUIZ

In the ANF version, `p` is the *second (local) variable* stored in the stack frame. What *value* gets moved into the *second stack slot* when evaluating the above program?

1. `0x3` → 3
2. `(3, (4, 5))` →
3. `0x11` → 17
4. `0x9` → 9
5. `0x10` → 16

ASM

e[0] → rbx ← ⟨e⟩
rbx ← rbx-1
rax ← [rbx+8·i]

e[1]

3. *Accessing*

Finally, to **access** the elements of a pair

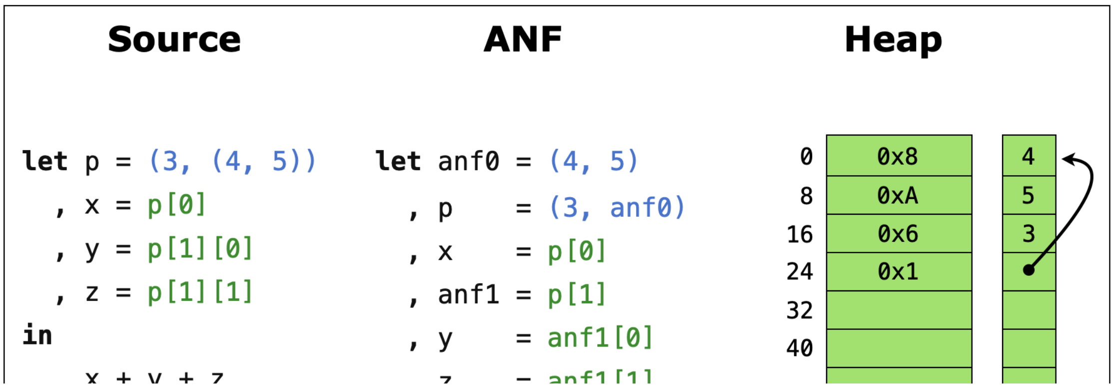
Lets compile `e[0]` to get the first or `e[1]` to get the second element

1. **Check** that immediate value `e` is a pointer
2. **Load** `e` into `rbx`
3. **Remove** the tag bit from `rbx`

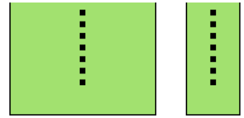
4. **Copy** the value in `[rbx]` (resp. `[rbx + 8]`) into `rbx`.

Example: Access

Here is a snapshot of the heap after the pair(s) are allocated.



```
  in
    x + y + z
```



Allocating Pairs on the Heap

Lets work out how the values corresponding to `x` , `y` and `z` in the example above get stored on the stack frame in the course of evaluation.

Variable	Hex Value	Value
`anf0`	`0x001`	`ptr 0`
`p`	`0x011`	`ptr 16`
`x`	`0x006`	`num 3`
`anf1`	`0x001`	`ptr 0`
`y`	`0x008`	`num 4`
`z`	`0x00A`	`num 5`
`anf2`	`0x00E`	`num 7`
result	`0x018`	`num 12`