Data on the Heap

Next, lets add support for

05-EGG is up due 5/23

- **Data Structures**

In the process of doing so, we will learn about

- **Heap Allocation**
- **Run-time Tags**

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
```

63

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)

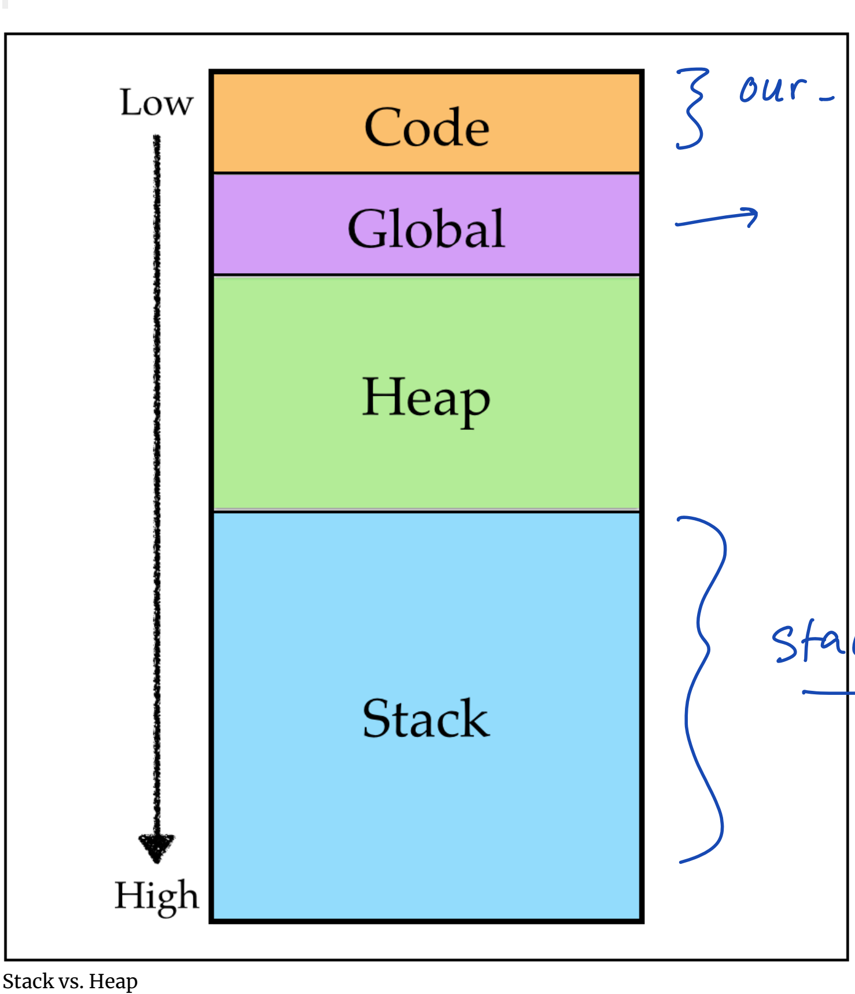
Instead, lets learn how to make **unbounded data structures**

- Lists
- Trees
- …

25 on stack
(1…25) not on stack?

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

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



our code lives

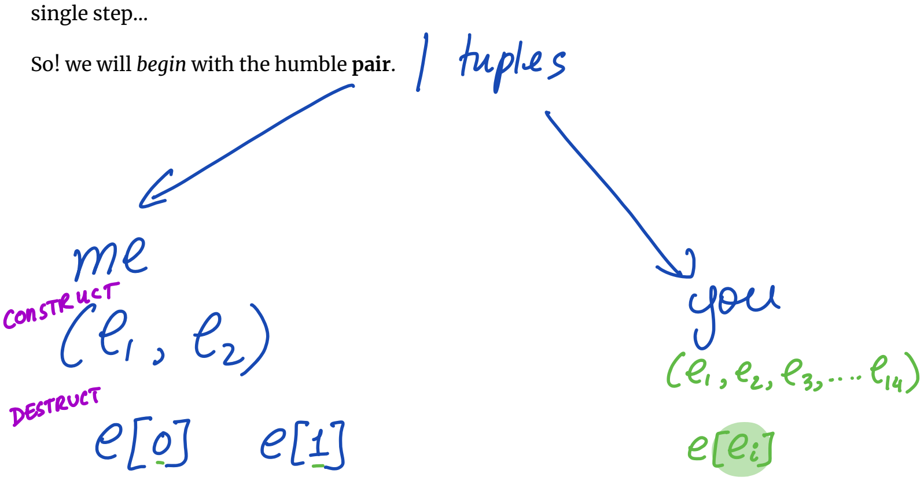
stack frames

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…

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



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

For example,

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

2 3 ⟹ 5

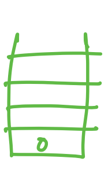
should evaluate to `5`.

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,
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:

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

64 bits
8-bytes

as a **single word** either

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

EXERCISE

What kinds of problems do you think might arise if we represent a pair `(2, 3)` on the *stack* as:

```
|     |
-------
|  30 |
-------
|  20 |
-------
| ... |
-------
```

let t = f(20, 30)
a = t[0]
b = t[1]
in
a + b

NEED size at compile time
① ↓ WHY? do I need size?
② why is this difficult?

(45, (10, (2, 3)))

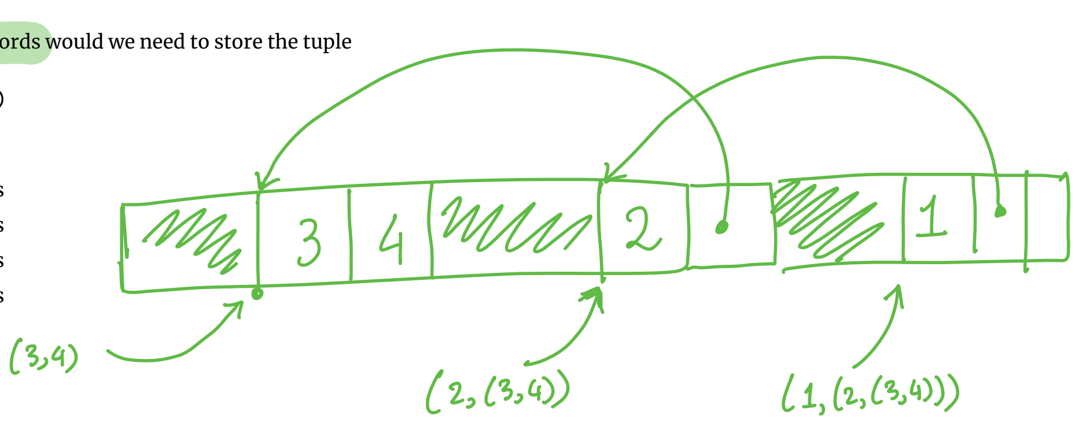
– HOW to even PASS/RETURN tuples (dont fit in RAX)?

QUIZ

How many words would we need to store the tuple

`(3, (4, 5))`

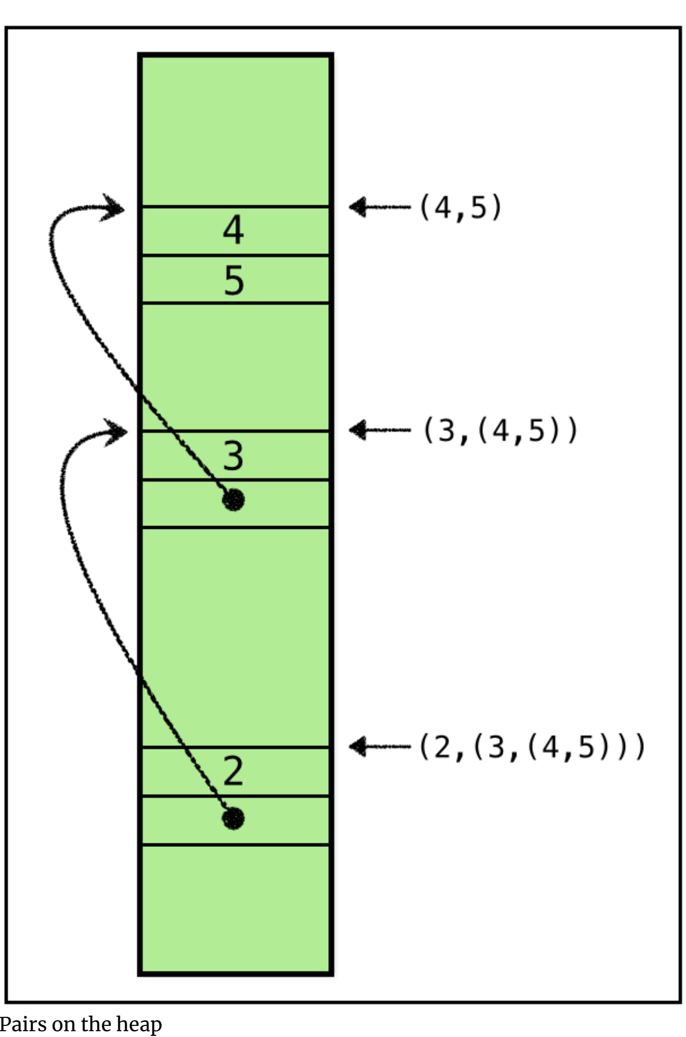
1. 1 word
2. 2 words
3. 3 words
4. 4 words
5. 5 words



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`)?

(4, 5)

Each of the above corresponds to a *different* tuple

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

(4, (…))

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

Tagging Pointers

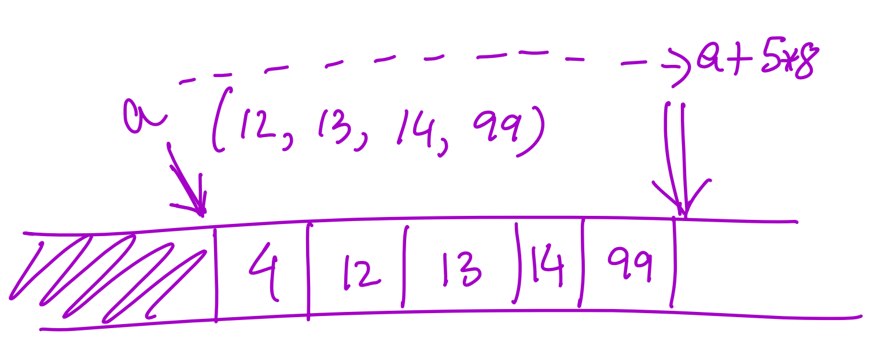
As you might have guessed, we can extend our tagging mechanism to account for *pointers*.

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

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.)



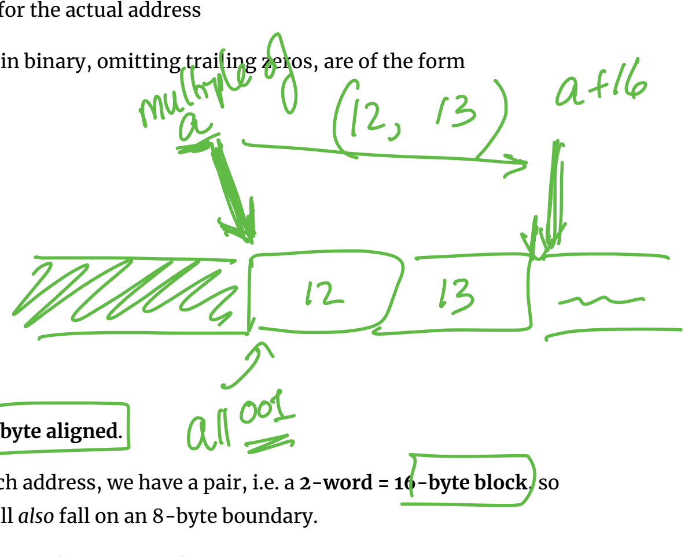
Address Alignment

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.

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

let t = (12, 13)
in

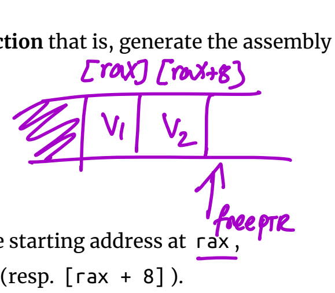
To repr tuple do you store
(A) START pointer ONLY
(B) START pointer + END POINTER
(C) START + NUM BLOCKS
(D) START + SIZE-IN-TUPLE

2. Construction

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

```
(e1, e2)
```

(v₁, v₂)



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

- 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` → for n-tuples (n+1)*8

- 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,

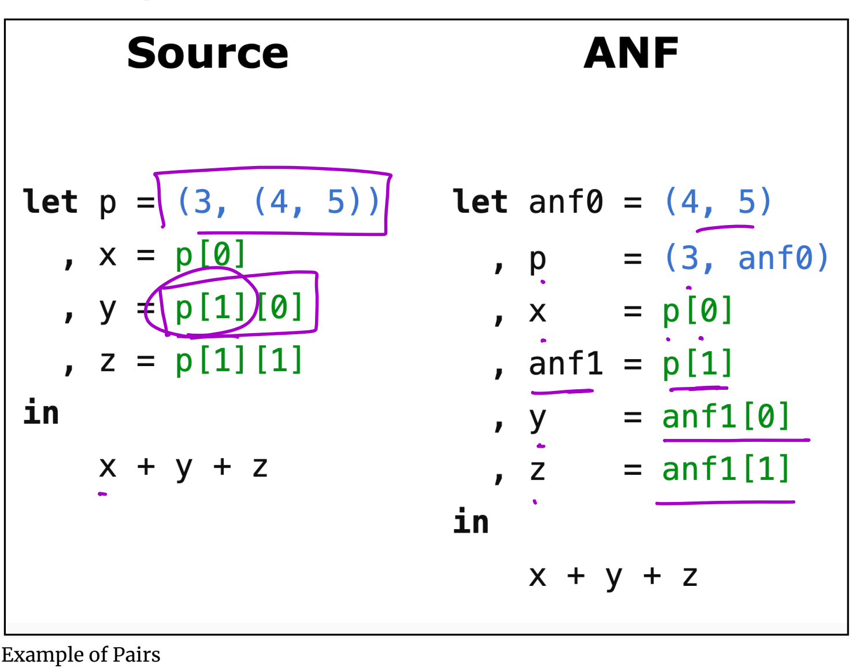
NOTE: Your assignment will have *blocks of varying sizes*

- ~~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

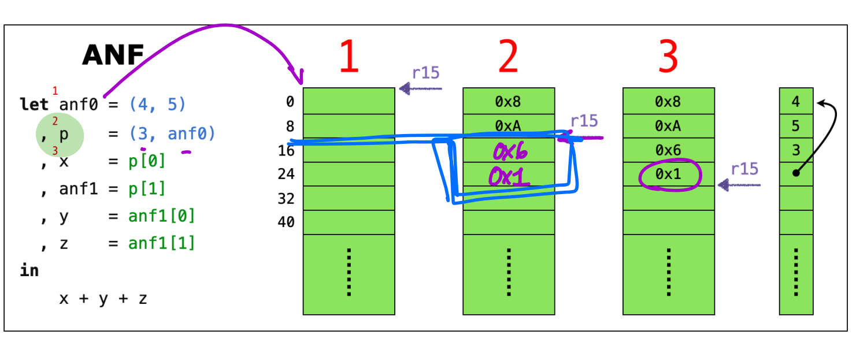
```
let p = (3, (4, 5))
  , x = p[0]
  , y = p[1][0]
  , z = p[1][1]
in
    x + y + z
```

ANF

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

Example of Pairs

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



ANF

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

Allocating Pairs on the Heap

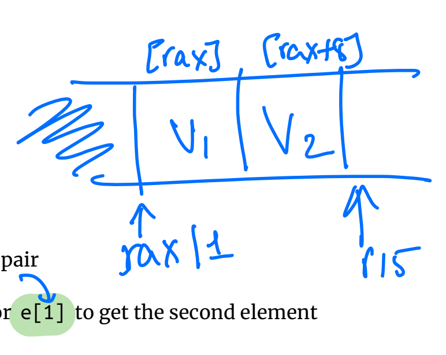
"16" with tag set to 1

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`
2. ~~`(3, (4, 5))`~~ ✗
3. `0x11` ✓
4. `0x9`
5. `0x10`

0x10
0x11



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

e1[e2]

v1[v2]

1. **Check** that immediate value `e` is a pointer
2. **Load** `e` into `rcx`
3. **Remove** the tag bit from `rcx`
4. **Copy** the value in `[rcx]` (resp. `[rcx + 8]`) into ~~`rcx`~~ rax

mov rax, [rcx+8]

"0"

check Ty v1 Tuple

check Index v1 v2

rcx ← v1

mov rax, [rcx+8*...]

Example: Access

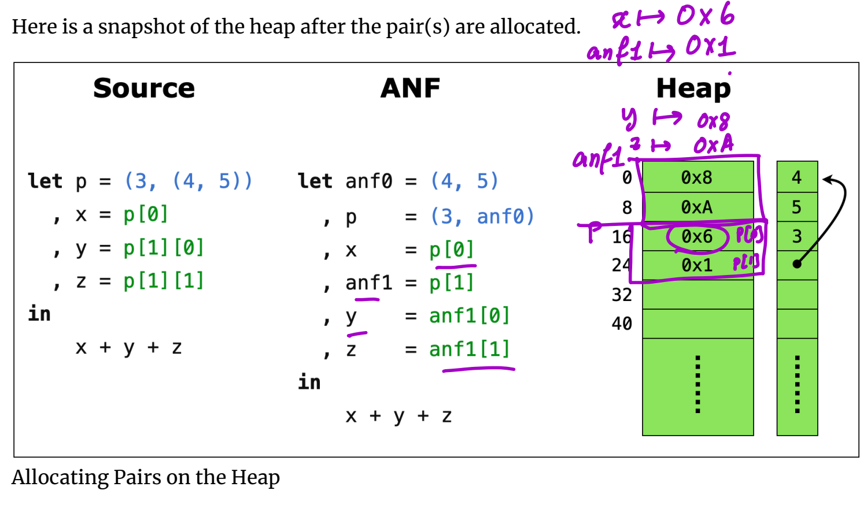
Here is a snapshot of the heap after the pair(s) are allocated.

p ↦ 0x11

x ↦ 0x6

anf1 ↦ 0x1

Can you have cycles in tuples?
- Yes ?
- No ?



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

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

(e1, e2) e[0] e[1]

const. destr

(A)

Plan

Pretty pictures are well and good, time to build stuff!

As usual, lets continue with our recipe:

alloc

1. Run-time → ① r15, ② print tuples
2. Types → Tuple
3. Transforms → (e1, e2) e[0] e[1]

We've already built up intuition of the *strategy* for implementing tuples.

Next, lets look at how to implement each of the above.

(B) Library of lists / trees etc.

Run-Time

We need to extend the run-time (`c-bits/main.c`) in two ways.

allocate heap

print (1, (2,3))

1. **Allocate** a chunk of space on the heap and pass in start address to `our_code`.
2. **Print** pairs properly.

Allocation

The first step is quite easy we can use `calloc` as follows:

```c
int main(int argc, char** argv) {
  int* HEAP = calloc(HEAP_SIZE, sizeof (int));
  long result = our_code_starts_here(HEAP);
  print(result);
  return 0;
}
```

The above code,

1. **Allocates** a big block of contiguous memory (starting at `HEAP`), and
2. **Passes** this address in to `our_code`.

Now, `our_code` needs to, at the beginning start with instructions that

- copy the parameter (in `rdi`) into global pointer (`r15`)
- and then bump it up at each allocation.

Printing

To print pairs, we must **recursively traverse** pointers

- until we hit `number` or `boolean`.

We can check if a value is a pair by looking at its last 3 bits:

```c
int isPair(int p) {
  return (p & 0x00000007) == 0x00000001;
}
```

We can use the above test to recursively print (word)-values:

```c
void print(long val) {
  if(val & 0x1 == 0) { // val is a number
    printf("%ld", val >> 1);
  }
  else if(val == CONST_TRUE) {        // val is true
    printf("true");
  }
  else if(val == CONST_FALSE) {       // val is false
    printf("false");
  }
  else if(val & 7 == 1) {
    long* valp = (long *) (val - 1);  // extract address
    printf("(");
    print(*valp);                     // print first element
    printf(", ");
    print(*(valp + 1));               // print second element
    printf(")");
  }
  else {
    printf("Unknown value: %#010x", val);
  }
}
```

Types

Next, lets move into our compiler, and see how the **core types** need to be extended.

Source

We need to extend the *source* `Expr` with support for tuples

```haskell
data Expr a
  = ...
  | Pair    (Expr a) (Expr a) a   -- ^ construct a pair
  | GetItem (Expr a) Field    a   -- ^ access a pair's element
```

In the above, `Field` is

```haskell
data Field
  = First     -- ^ access first element of pair
  | Second    -- ^ access second element of pair
```

NOTE: Your assignment will generalize pairs to **n-ary tuples** using

- `Tuple [Expr a]` representing `(e1,...,en)`
- `GetItem (Expr a) (Expr a)` representing `e1[e2]`

Dynamic Types

Let us extend our **dynamic types** `Ty` see to include pairs:

```haskell
data Ty = TNumber | TBoolean | TPair
```

Assembly

The assembly `Instruction` are changed minimally; we just need access to `r15` which will hold the value of the *next* available memory block:

```haskell
data Register
  = ...
  | R15
```

Transforms

Our code must take care of three things:

init alloc

alloc

access

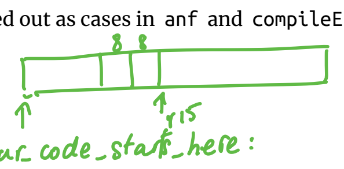
1. **Initialize** `r15` to allow heap allocation,
2. **Construct** pairs,
3. **Access** pairs.

(e1, e2) → Pair e1 e2

e[0] → GetItem e f

The latter two will be pointed out as cases in `anf` and `compileEnv`

- `Tuple`
- `GetItem`



our_code_starts_here:

① align-alloc

②

how to ~~set~~ r15?

'rdi' is HEAP

mov r15, rdi

rax ← rdi

rax ← rax+8

rax ← mask out lo

Initialize

We need to **initialize** `r15` with the **start position** of the heap

- passed in as `rdi` by the run-time.

How shall we get a hold of this position?

To do so, `our_code` starts off with a `prelude`

```haskell
prelude :: [Instruction]
prelude =
  [ IMov (Reg R15) (Reg RDI)     -- copy param (HEAP) off rdi
  ]
```

Is that it?

QUIZ

Is `r15` 8-byte aligned?

A. Yes
B. No

Ensuring alignment

```haskell
prelude :: [Instruction]
prelude =
  [ IMov (Reg RAX) (HexConst 0xFFFFFFFF)      -- setup regMask
  , IShl (Reg RAX) (Const 32)
  , IOr  (Reg RAX) (HexConst 0xFFFFFFF8)

  , IMov (Reg R15) (Reg RDI)                  -- copy param (HEAP) off rdi
  , IAdd (Reg R15) (Const 8)                  -- add 8 and mask 3 bits to
ensure
  , IAnd (Reg R15) (Reg RAX)                  -- 8-byte aligned
  ]
```

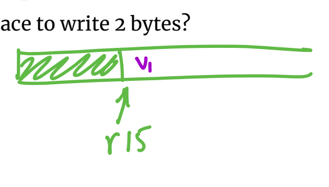
1. **Copy** the value off the (parameter) stack, and
2. **Adjust** the value to ensure the value is *8-byte aligned*.

QUIZ

Why add `8` to `r15`? What would happen if we *removed* that operation?

1. `r15` would not be 8-byte aligned?
2. `r15` would point into the stack?
3. `r15` would not point into the heap?
4. `r15` would not have enough space to write 2 bytes?

(V1, V2)



Construct

To *construct* a pair `(v1, v2)` we directly implement the above strategy:

```haskell
compileEnv env (Tuple v1 v2)
  = pairAlloc                         -- 1. allocate pair, resulting addr i
n `rax`
 ++ pairCopy First  (immArg env v1)  -- 2. copy first value into slots
 ++ pairCopy Second (immArg env v2)  -- 3. copy second value into slot
 ++ setTag   RAX     TPair           -- 3. set the tag-bits of `rax`
```

Lets look at each step in turn.

Allocate

To allocate, we just copy the current pointer `r15` and increment by `16` bytes,

- accounting for two 8-byte blocks for each element.

```haskell
pairAlloc :: Asm
pairAlloc
  = [ IMov (Reg RAX) (Reg R15)    -- copy current "free address" `esi` int
o `eax`
    , IAdd (Reg RAX) (Const 16)   -- increment `esi` by 8
    ]
```

Exercise How would you make this work for `n`-tuples?

Copy

We copy an `Arg` into a `Field` by

- saving the `Arg` into a helper register `rcx`,
- copying `rcx` into the field's slot on the heap.

```haskell
pairCopy :: Field -> Arg -> Asm
pairCopy fld arg
  = [ IMov (Reg RCX)      arg
    , IMov (pairAddr fld) (Reg RCX)
    ]
```

Recall, the field's slot is either `[rax]` or `[rax + 8]` depending on whether the field is `First` or `Second`.

QUIZ

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

```haskell
pairAddr :: Field -> Arg
pairAddr First  = RegOffset ?1 RAX
pairAddr Second = RegOffset ?2 RAX
```

0

(-1)

A. `0` and `1`

B. `0` and `-1`

C. `1` and `2`

D. `-1` and `-2`

E. huh?

Tag

Finally, we set the tag bits of `rax` by using `typeTag TPair` which is defined

```haskell
setTag :: Register -> Asm
setTag r = [ IAdd (Reg r) (HexConst 0x1) ]
```

e[0]

⟨e⟩

assertTy ~~x~~ e TTuple

rax has value of 'e'

strip out tag on RAX

rcx ← [rax]

#

rax ← rcx

Access

To access tuples, lets update `compileEnv` with the strategy above:

```haskell
compileExpr env (GetItem e fld)
  = assertType env e TPair               -- 1. check that e is a (pair) poi
nter
 ++ [ IMov (Reg RAX) (immArg env e) ]   -- 2. load pointer into eax
 ++ unsetTag RAX                        -- 3. remove tag bit to get addres
s
 ++ [ IMov (Reg RAX) (pairAddr fld) ]   -- 4. copy value from resp. slot t
o eax
```

we remove the tag bits by doing the opposite of `setTag` namely:

```
unsetTag :: Register -> Asm
unsetTag r = ISub (Reg RAX) (HexConst 0x1)
```

N-ary Tuples

Thats it! Lets take our compiler out for a spin, by using it to write some interesting programs!

First, lets see how to generalize pairs to allow for

- triples `(e1,e2,e3)`
- quadruples `(e1,e2,e3,e4)`
- pentuples `(e1,e2,e3,e4,e5)`

and so on.

We just need a library of functions in our new `egg` language to

- **Construct** such tuples, and
- **Access** their fields.

Constructing Tuples

We can write a small set of functions to **construct** tuples (up to some given size):

```
def tup3(x1, x2, x3):
  (x1, (x2, x3))

def tup4(x1, x2, x3, x4):
  (x1, (x2, (x3, x4)))

def tup5(x1, x2, x3, x4, x5):
  (x1, (x2, (x3, (x4, x5))))
```

Accessing Tuples

We can write a single function to access tuples of any size.

So the below code

```
let yuple = (10, (20, (30, (40, (50, false))))) in

get(yuple, 0) = 10
get(yuple, 1) = 20
get(yuple, 2) = 30
get(yuple, 3) = 40
get(yuple, 4) = 50



def tup3(x1, x2, x3):
  (x1, (x2, x3))

def tup5(x1, x2, x3, x4, x5):
  (x1, (x2, (x3, (x4, x5))))

let t  = tup5(1, 2, 3, 4, 5) in
  , x0 = print(get(t, 0))
  , x1 = print(get(t, 1))
  , x2 = print(get(t, 2))
  , x3 = print(get(t, 3))
  , x4 = print(get(t, 4))
in
  99
```

should print out:

```
0
1
2
3
4
99
```

How shall we write it?

```
def get(t, i):
  TODO-IN-CLASS
```

QUIZ

Using the above "library" we can write code like:

```
let quad = tup4(1, 2, 3, 4) in
  get(quad, 0) + get(quad, 1) + get(quad, 2) + get(quad, 3)
```

What will be the result of compiling the above?

1. Compile error
2. Segmentation fault
3. Other run-time error
4. `4`
5. `10`

QUIZ

Using the above "library" we can write code like:

```
def get(t, i):
  if i == 0:
    t[0]
  else:
    get(t[1],i-1)

def tup3(x1, x2, x3):
  (x1, (x2, (x3, false)))

let quad = tup3(1, 2, 3) in
  get(quad, 0) + get(quad, 1) + get(quad, 2) + get(quad, 3)
```

What will be the result of compiling the above?

1. Compile error
2. Segmentation fault
3. Other run-time error
4. `4`
5. `10`

Lists

Once we have pairs, we can start encoding **unbounded lists**.

To build a list, we need two constructor functions:

~~~
def empty():
  false

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


We can now encode lists as:

```python
cons(1, cons(2, cons(3, cons(4, empty()))))
~~~

Access

To **access** a list, we need to know

1. Whether the list `isEmpty` , and
2. A way to access the `head` and the `tail` of a non-empty list.

```
def isEmpty(l):
  l == empty()

def head(l):
  l[0]

def tail(l):
  l[1]
```

Examples

We can now write various functions that build and operate on lists, for example, a function to generate the list of numbers between `i` and `j`

```
def range(i, j):
  if (i < j):
    cons(i, range(i+1, j))
  else:
    empty()
```

```
range(1, 5)
```

which should produce the result

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

and a function to sum up the elements of a list:

```
def sum(xs):
  if (isEmpty(xs)):
    0
  else:
    head(xs) + sum(tail(xs))
```

```
sum(range(1, 5))
```

which should produce the result `10` .

Recap

We have a pretty serious language now, with:

- **Data Structures**

which are implemented using

- **Heap Allocation**
- **Run-time Tags**

map
filter

which required a bunch of small but subtle changes in the

fold

- runtime and compiler

In your assignment, you will add *native* support for n-ary tuples, letting the programmer write code like:

```
(e1, e2, e3, ..., en)   # constructing tuples of arbitrary arity

e1[e2]                  # allowing expressions to be used as fields
```

Next, we'll see how to

- use the "tuple" mechanism to implement **higher-order functions** and
- reclaim unused memory via **garbage collection**.
