- Heap Data
- Garbage Collection
- Closures

ucsd-progsys / **131-web**

Branch: master ▾ 131-web / lectures / **07-egg-eater.md** Find file Copy path

ranjitjhala update page cec2b93 17 minutes ago

1 contributor

826 lines (563 sloc) 18 KB

title	date	headerImg
Data on the Heap	2013-03-02	egg-eater.jpg

Next, lets add support for

- **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
```

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

- `Char`
- `Double` or `Float`
- `Long` or `Short`

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 (4-bytes)

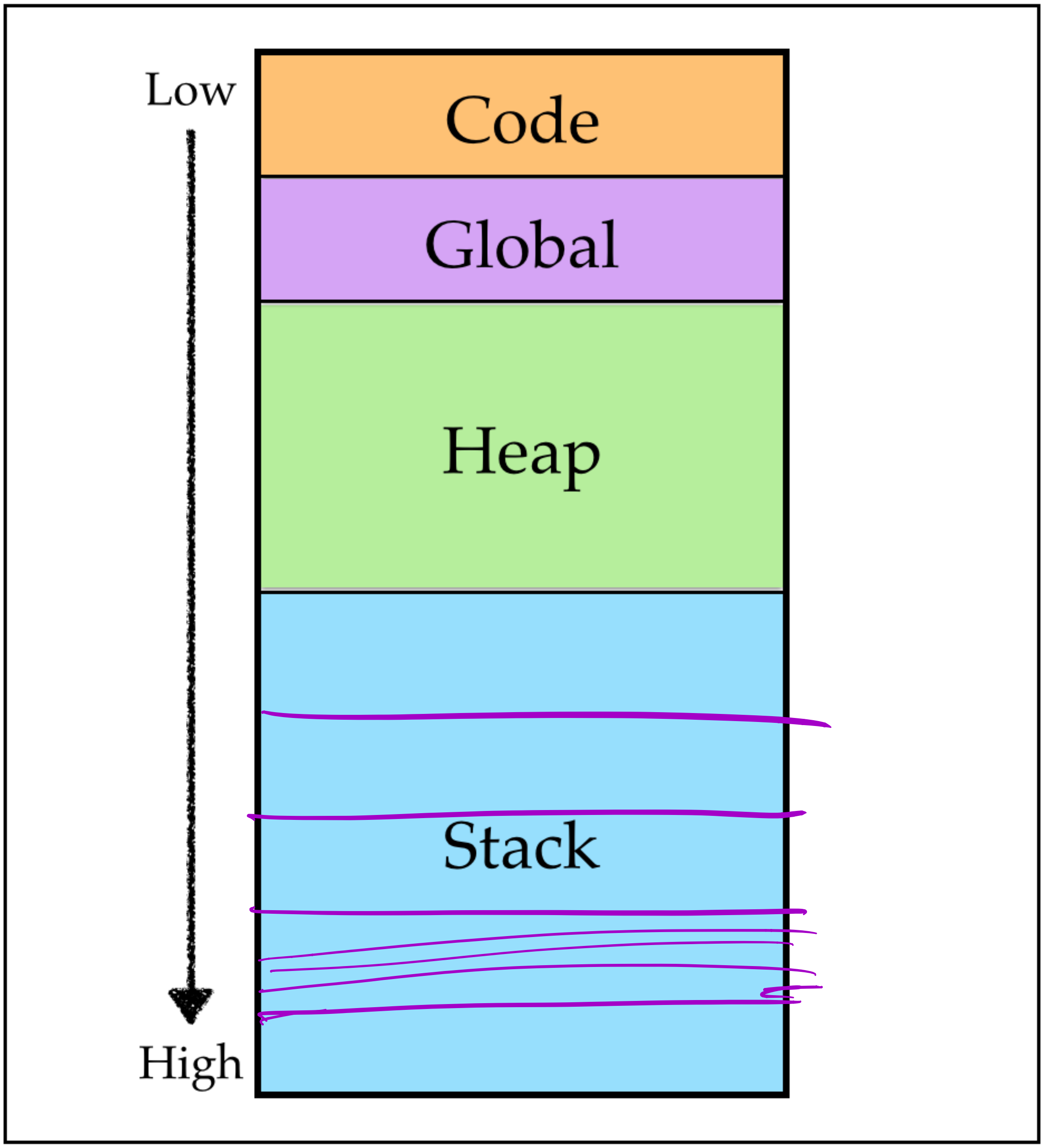
Instead, we're going to look at how to make **unbounded data structures**

- Lists
- Trees

which require us to put data on the **heap** (not just the *stack*) that we've used so far.

- (e_1, e_2)

- $e_1[e_2]$



Pairs

While our *goal* is to get to lists and trees, the journey of a thousand miles, etc., and so, we will *begin* with the humble **pair**.

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.

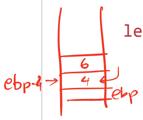
For example,

let t = (1, (2, (3, 4)))

in t[0] + t[1] + t[2] + t[4]

(e_1, e_2)

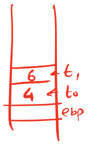
let t = (2,3) in
foo(t)



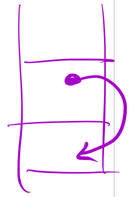
```
let t = (2, 3) in
  t[0] + t[1]
```

foo = t

let t0' = 2
, t1^2 = 3
in t0 + t1

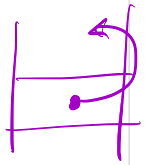


should evaluate to `5` .



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 memore,
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`

as a **single word** either

- 4 bytes on the stack, or
- a single register `eax` .

EXERCISE

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

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

Pairs vs. Primitive Values

The main difference between pairs and primitive values like `number` and `boolean` is that there is no *fixed* or *bounded* amount of space we can give to a pair. For example:

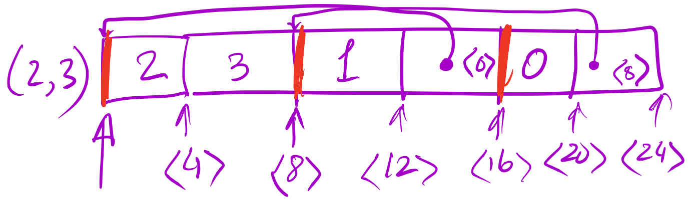
- `(4, 5)` takes at least 2 words,
- `(3, (4, 5))` takes at least 3 words,
- `(2, (3, (4, 5)))` takes at least 4 words and so on.

Thus, once you start *nesting* pairs we can't neatly tuck all the data into a fixed number of 1- or 2- word slots.

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.

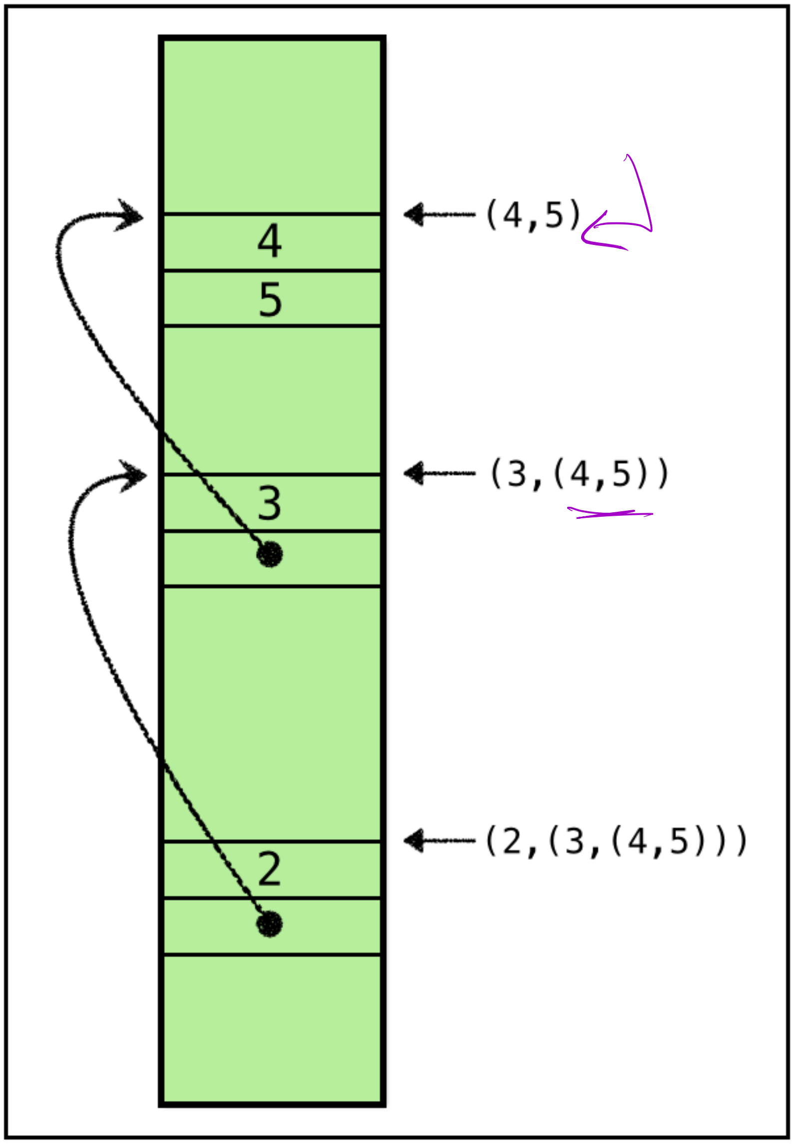


b ↦ <0> }

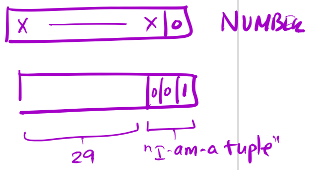
let b = (2,3)
a = (1, b)

a ↦ ⟨8⟩

c = (0, a)



let t = (0,10)
in t+2



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

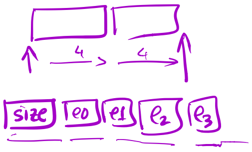
1. `(4, 5)` or
2. `(4, (...))`.

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

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



000

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.)

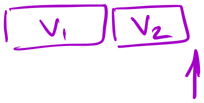
Address Alignment

As we have a **3 bit tag**, leaving **32 - 3 = 29 bits** for the actual address. This means, our actual available addresses, written in binary are of the form

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

let bs_1, v_1
bs_1, v_2
(v_1, v_2)

(V_1, V_2)



mov eax, esi
mov [esi], <v1>
mov [esi+4], <v2>
add esi, 8
add eax, 1

$e \longrightarrow [\text{Instr}]$

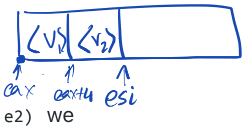
eax= val of 'e'

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

2. Construction Egg

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

`(e1, e2)` (V_1, V_2)



mov eax, esi
mov [eax], <v1>
mov [eax+4], <v2>
add esi, 8
add eax, 1

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

1. **Allocate** a new 2-word block, and getting the starting address at `eax`,
2. **Copy** the value of `e1` (resp. `e2`) into `[eax]` (resp. `[eax + 4]`).

$(v_1, v_2, v_3 \dots v_n)$

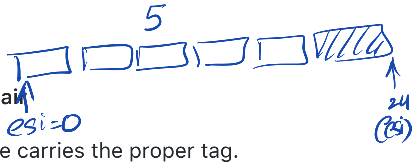
A 8
B 4*n

mov eax, esi
mov [eax], n
[mov [eax+4*i] vi | i<-1..n]
add esi, $\boxed{8}$

C 4*(n+1)
D 4*(n+2)
E 4*(n+3)

n = 4

3. **Tag** the last bit of `eax` 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 both immediate expressions which will make the second step above straightforward.

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

mov

Allocating Addresses

We will use a **global** register `esi` 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 `esi` into `eax`
   - set the last bit to `1` to ensure proper tagging.
   - `eax` will be used to fill in the values
2. **Increment** the value of `esi` by `8`
   - thereby "allocating" 8 bytes (= 2 words) at the address in `eax`

$V_1[V_2]$

```
mov  ebx, <V1>
sub  ebx, 1
mov  ecx, V2
ishr ecx, 1
mov  eax, [ebx]
cmp  eax, ecx
mov  eax, [V1+4*ecx]
```

actual size > index ≥ 0

Note that *if*

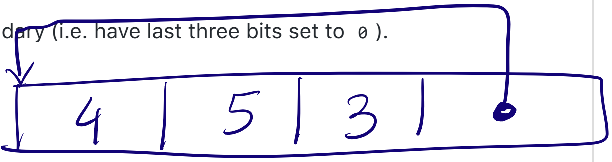
- we *start* our blocks at an 8-byte boundary, and
- we *allocate* 8 bytes at a time,

then

- each address used to store a pair will fall on an 8-byte boundary (i.e. have last three bits set to `0`).

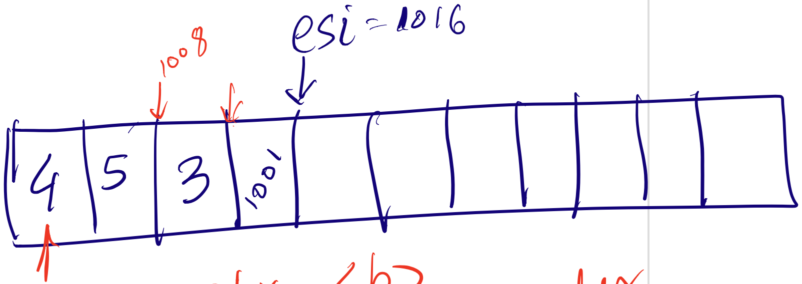
So we can safely turn the address in `eax` into a `pointer`

- by setting the last bit to `1`.



NOTE: In your assignment, we will have blocks of varying sizes so you will have to take care to *maintain* the 8-byte alignment, by "padding".

Example: Allocation



In the figure below, we have

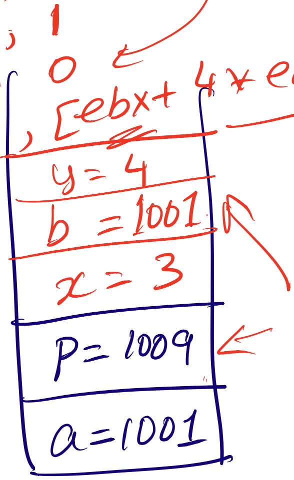
- a source program on the left,
- the ANF equivalent next to it.

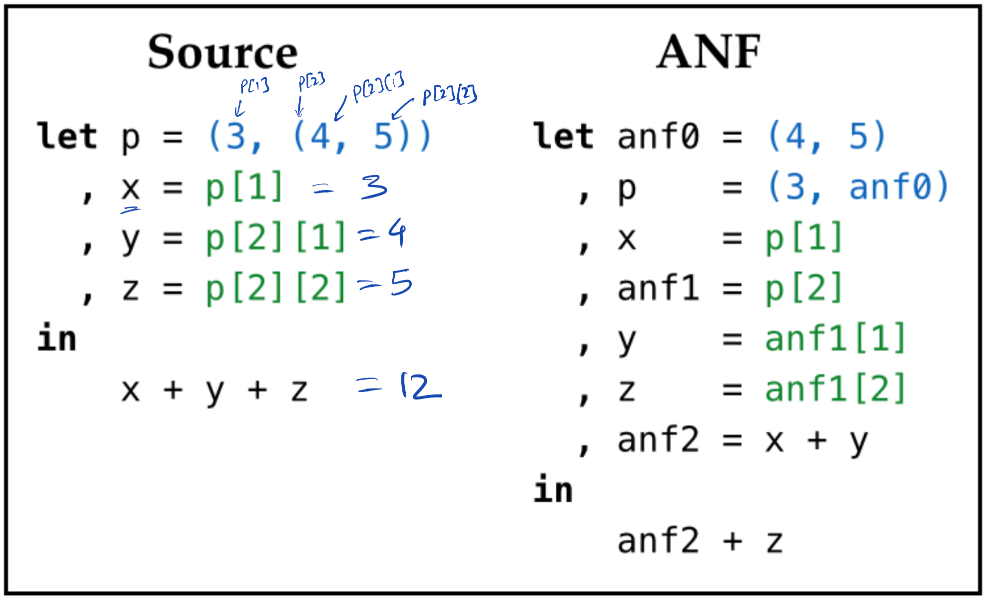
let p = (3,(4,5))
x = p[0]
y = p[1][0]
z = p[1][1]
in (x+y)+z

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

```
mov  ebx, <b>
sub  ebx, 1
mov  ecx, 0
mov  eax, [ebx+4*ecx]
```

index

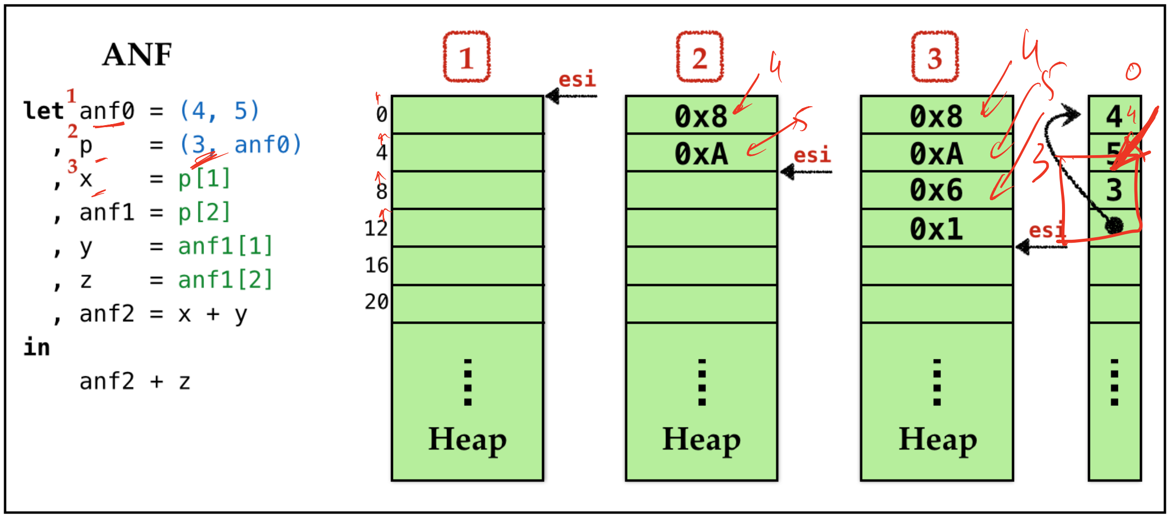




```
let p = (3, (4, 5))
  , x = p[1]
  , y = p[2][1]
  , z = p[2][2]
in
    x + y + z
```

```
let anf0 = (4, 5)
  , p    = (3, anf0)
  , x    = p[1]
  , anf1 = p[2]
  , y    = anf1[1]
  , z    = anf1[2]
  , anf2 = x + y
in
    anf2 + z
```

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



```
let anf0 = (4, 5)
  , p    = (3, anf0)
  , x    = p[1]
  , anf1 = p[2]
  , y    = anf1[1]
  , z    = anf1[2]
  , anf2 = x + y
in
    anf2 + z
```

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?

i.e. for p

1. `0x3`
2. `(3, (4, 5))`
3. `0x6`
4. `0x9`
5. `0x10`

3. Accessing

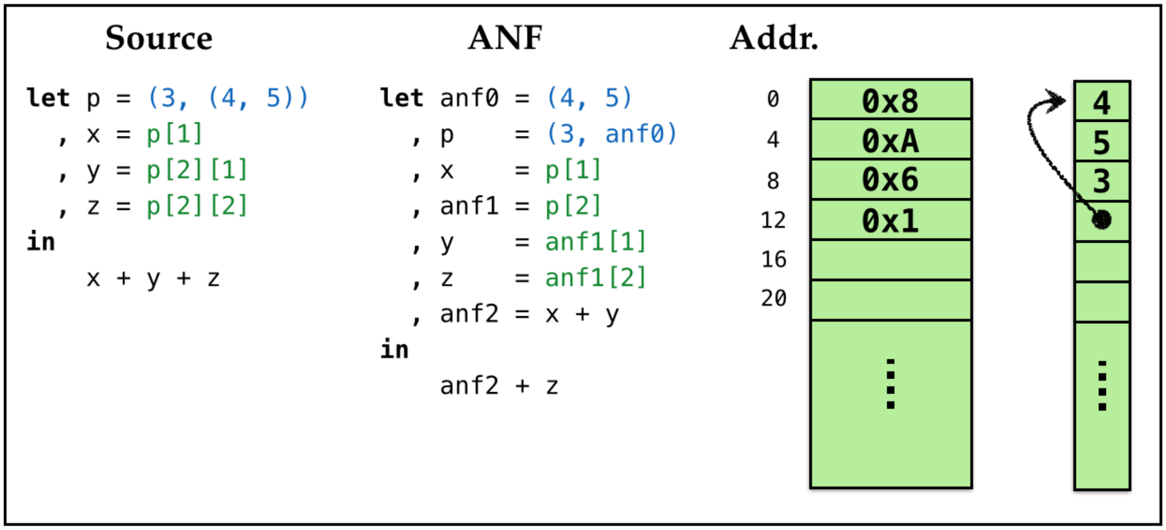
Finally, to **access** the elements of a pair, i.e. compiling expressions like `e[0]` (resp. `e[1]`)

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

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

Example: Access

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



Source

```
let p = (3, (4, 5))
  , x = p[1]
  , y = p[2][1]
  , z = p[2][2]
in
    x + y + z
```

ANF

```
let anf0 = (4, 5)
  , p    = (3, anf0)
  , x    = p[1]
  , anf1 = p[2]
  , y    = anf1[1]
  , z    = anf1[2]
  , anf2 = x + y
in
    anf2 + z
```

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`	`0x009`	`ptr 8`
`x`	`0x006`	`num 3`
`anf1`	`0x001`	`ptr 0`
`y`	`0x008`	`num 4`
`z`	`0x00A`	`num 5`
`anf2`	`0x00E`	`num 7`
result	`0x018`	`num 12`

Plan

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

As usual, lets continue with our recipe:

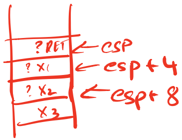
1. Run-time → C ↪ print_tuples ↪ create HEAP
2. Types
3. Transforms

We've already built up intuition of the *strategy* for implementing tuples. Next, lets look at how to implement each of the above.

Run-Time

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

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:

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

mov esi, [esp+4]

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 will copy the parameter into `esi` and then bump it up at each allocation.

Printing

To print pairs, we must recursively traverse the pointers until we hit `number` or `boolean`.

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

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

0b111

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

isNum

is Bool True

is Bool False

```
void printRec(int val) {
  if(val & 0x00000001 ^ 0x00000001) { // val is a number
    printf("%d", val >> 1);
  }
  else if(val == 0xFFFFFFFF) {       // val is true
    printf("true");
  }
  else if(val == 0x7FFFFFFF) {       // val is false
    printf("false");
  }
  else if(isPair(val)) {
    int* valp = (int*) (val - 1);    // extract address
    printf("(");
    printRec(*valp);                 // print first element
    printf(", ");
    printRec(*(valp + 1));           // print second element
    printf(")");
  }
  else {
    printf("Unknown value: %#010x", val);
  }
}
```

"n" ←

for i=0..n-1
valp[i]

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

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

(Pair e_1 e_2 _)

~~Field~~ (Expr a)

e[0] (GetItem e First)

e[1] (GetItem e Second)

In the above, `Field` is

$e_1[e_2]$

```
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:

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

Assembly

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

```
data Register
  = ...
  | ESI
```

Transforms

Our code must take care of three things:

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

The latter two will be pointed out directly by GHC

- They are new cases that must be handled in `anf` and `compileExpr`

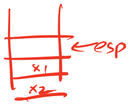
Initialize

We need to **initialize** `esi` with the **start position** of the heap, that is passed in by the run-time.

How shall we get a hold of this position?

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

```
prelude :: [Instruction]
prelude =
```



esp+4

```
[ IMov (Reg ESI) (RegOffset 4 ESP)       -- copy param (HEAP) off stack
, IAdd (Reg ESI) (Const 8)               -- adjust to ensure 8-byte aligned
, IAnd (Reg ESI) (HexConst 0xFFFFFFF8)   -- add 8 and set last 3 bits to 0
]
```

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 `esi` ? What would happen if we *removed* that operation?

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

Construct

(e1; e2)

IMM=multiple sub-exprs which MUST be eval.

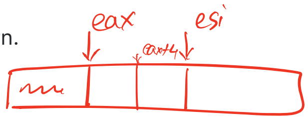
To *construct* a pair `(v1, v2)` we directly implement the above strategy:

add eax 1

```
compileExpr env (Pair v1 v2)
  = pairAlloc                           -- 1. allocate pair, resulting addr in `eax`
 ++ pairCopy First  (immArg env v1)     -- 2. copy values into slots
 ++ pairCopy Second (immArg env v2)
 ++ setTag  EAX    TPair                -- 3. set the tag-bits of `eax`
```

mov [eax], <v1>
mov [eax+4], <v2>
add eax, 1

Lets look at each step in turn.



Allocate

To allocate, we just copy the current pointer `esi` and increment by `8` bytes,

- accounting for two 4-byte (word) blocks for each pair element.

```
pairAlloc :: Asm
pairAlloc
  = [ IMov (Reg EAX) (Reg ESI)   -- copy current "free address" `esi` into `eax`
    , IAdd (Reg ESI) (Const 8)   -- increment `esi` by 8
    ]
```

Copy

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

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

```
pairCopy :: Field -> Arg -> Asm
pairCopy fld a
  = [ IMov (Reg EBX)    a
    , IMov (pairAddr f) (Reg EBX)
    ]
```

The field's slot is either `[eax]` or `[eax + 4]` depending on whether the field is `First` or `Second` .

```
pairAddr :: Field -> Arg
pairAddr fld = Sized DWordPtr (RegOffset (4 * fieldOffset fld) EAX)
```

```
fieldOffset :: Field -> Int
fieldOffset First  = 0
fieldOffset Second = 1
```

Tag

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

```
setTag :: Register -> Ty -> Asm
setTag r ty = [ IAdd (Reg r) (typeTag ty) ]

typeTag :: Ty -> Arg
typeTag TNumber   = HexConst 0x00000000    -- last 1 bit  is  0
typeTag TBoolean  = HexConst 0x00000007    -- last 3 bits are 111
typeTag TPair     = HexConst 0x00000001    -- last 1 bits is  1
```

Access

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

```
compileExpr env (GetItem e fld)
  assertType env e TPair                     -- 1. check that e is a (pair) pointer
  ++ [ IMov (Reg EAX) (immArg env e) ]       -- 2. load pointer into eax
  ++ unsetTag EAX TPair                      -- 3. remove tag bit to get address
  ++ [ IMov (Reg EAX) (pairAddr fld) ]       -- 4. copy value from resp. slot to eax
```

$V_1[V_2]$

assertType env e Tuple

ebx <= evaluate V_1 at offset 0

evaluate V_2

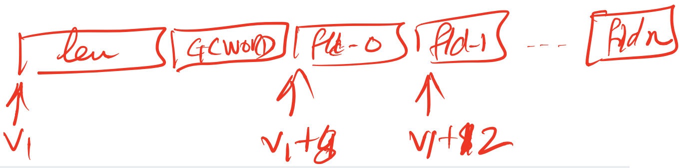
check $V_2 \geq 0$

check $V_2 < V_1$.len

$[V_1 + 4*(V_2+1)]$

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

```
unsetTag :: Register -> Ty -> Asm
unsetTag r ty = ISub (Reg EAX) (typeTag ty)
```



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 (upto 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

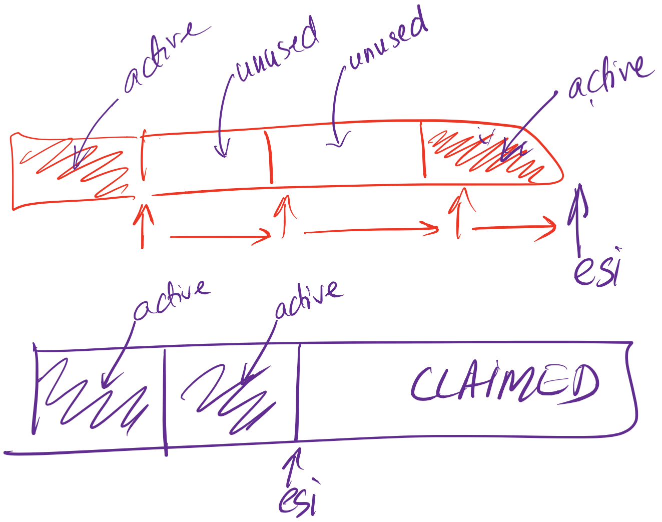
```
let t  = tup5( 1, 2 , 3 , 4 , 5) in
  , x0 = print(get(t, 0))
  , x1 = print(get(t, 1))
  , x2 = print(get(t, 2))
  , x3 = print(get(t, 3))
  , x4 = print(get(t, 4))
in
  99
```

should print out:

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

How shall we write it?

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



① What is MARK(ED)

② How to COMPACT

mark-compact GC

C# Runtime, Haskell, JVM

QUIZ

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

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

def head(e):
  e[0]

def tail(e):
  e[1]

def get(e, i):
  if (i == 0):
      head(e)
  else:
    get(tail(e), i-1)

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

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

get(q, 0) = q[0] = 1
get(q, 1) = get(q[1], 0) = 2
get(q, 2) = get(q[1], 1) = get(q[1][1], 0) = 3
get(q, 3) = get(q[1], 2) = get(q[1][1], 1) = get(q[1][1][1], 0) = get(4, 0)
          = 4[0]
```

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:

```
let quad = tup4(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**.

Construct

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:
    emp()

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**

which required a bunch of small but subtle changes in the

- 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 "pair" mechanism to add support for **higher-order functions** and
- reclaim unused memory via **garbage collection**.