Next, let's 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

```plaintext
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 Of Float
- Long Of 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.
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, let's 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 \((e_0, e_1)\) where \(e_0\) and \(e_1\) 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,

\[
\begin{align*}
\text{let } t = (1, (2, (3, 4))) \\
\end{align*}
\]
let \( t = (2, 3) \) in
\[
\begin{align*}
&\text{let } t_0 = 2, t_1 = 3, \text{ in} \\
&\begin{array}{c}
\text{t[0]} = t_0 \\
\text{t[1]} = t_1
\end{array}
\end{align*}
\]
should evaluate to 5.

**Strategy**

Next, let's 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 \((e_0, e_1)\) 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:

\[
\begin{array}{c}
\text{| } \text{|} \\
\text{| } \text{|} \\
\text{| 3 |} \\
\text{| } \text{|} \\
\text{| 2 |} \\
\text{| } \text{|} \\
\text{| ... |}
\end{array}
\]

**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.
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 it’s 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>xx0</td>
</tr>
<tr>
<td>boolean</td>
<td>111</td>
</tr>
<tr>
<td>pointer</td>
<td>001</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000000</td>
<td>0</td>
</tr>
<tr>
<td>0b00001000</td>
<td>8</td>
</tr>
<tr>
<td>0b00010000</td>
<td>16</td>
</tr>
<tr>
<td>0b00011000</td>
<td>24</td>
</tr>
<tr>
<td>0b00100000</td>
<td>32</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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**

Next, let’s look at how to implement pair construction that is, generate the assembly for expressions like:

\[(e_1, e_2)\]

To construct a pair \((e_1, e_2)\) we

1. **Allocate** a new 2-word block, and getting the starting address at \(\text{eax}\);
2. **Copy** the value of \(e_1\) (resp. \(e_2\)) into \([\text{eax}]\) (resp. \([\text{eax} + 4]\));

\[
\begin{align*}
& (e_1, e_2) & \left( \begin{array}{c} v_1, v_2 \end{array} \right) & \begin{array}{c} \text{mov} \text{ eax, esi} \\
& \text{mov} [\text{eax}], <v_1> & \text{mov} [\text{eax} + 4], <v_2> & \text{add} \text{ esi, 8} & \text{add} \text{ eax, 1} \\
& \end{array} \end{align*}
\]

\[
\begin{align*}
& (v_1, v_2, v_3, \ldots v_n) & \begin{array}{c} \text{mov} \text{ mov} [\text{eax}], v_i & \text{add} \text{ esi, 8} & \text{add} \text{ eax, 1} \\
& \text{mov} [\text{eax} + 4], v_i & \text{add} \text{ esi, 8} & \text{add} \text{ eax, 1} \\
& \end{array} \end{align*}
\]
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 \( e_1 \) and \( e_2 \) are both **immediate expressions** which will make the second step above straightforward.

**EXERCISE** How will we do ANF conversion for \((e_1, e_2)\)?

### 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 \( eax \)

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

### Source

```
let p = (3, (4, 5))
  , x = p[1] = 3
  , y = p[2][1] = 4
  , z = p[2][2] = 5
in
  x + y + z = 12
```

### 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
```

**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. \( 0x6 \)
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 \([\text{eax}]\) (resp. \([\text{eax} + 4]\)) into \(\text{eax}\).

Example: Access

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

<table>
<thead>
<tr>
<th>Source</th>
<th>ANF</th>
<th>Addr.</th>
</tr>
</thead>
</table>
| \textbf{let} \(p = (3, (4, 5))\), \(x = p[1]\), \(y = p[2][1]\), \(z = p[2][2]\) \textbf{in} \(x + y + z\) | \textbf{let} \(\text{anf0} = (4, 5)\), \(p = (3, \text{anf0})\), \(x = p[1]\), \(\text{anf1} = p[2]\), \(y = \text{anf1}[1]\), \(z = \text{anf1}[2]\), \(\text{anf2} = x + y\) \textbf{in} \(\text{anf2} + z\) | \begin{align*}
0 & \quad 0x8 \\
4 & \quad 0xA \\
8 & \quad 0x6 \\
12 & \quad 0x1 \\
16 & \quad 0x3 \\
20 & \quad \ldots
\end{align*} |

Let's 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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Hex Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{anf0}</td>
<td>0x001</td>
<td>ptr 0</td>
</tr>
<tr>
<td>\text{p}</td>
<td>0x009</td>
<td>ptr 8</td>
</tr>
<tr>
<td>\text{x}</td>
<td>0x006</td>
<td>num 3</td>
</tr>
<tr>
<td>\text{anf1}</td>
<td>0x001</td>
<td>ptr 0</td>
</tr>
<tr>
<td>\text{y}</td>
<td>0x008</td>
<td>num 4</td>
</tr>
<tr>
<td>\text{z}</td>
<td>0x00A</td>
<td>num 5</td>
</tr>
<tr>
<td>\text{anf2}</td>
<td>0x00E</td>
<td>num 7</td>
</tr>
<tr>
<td>\text{result}</td>
<td>0x018</td>
<td>num 12</td>
</tr>
</tbody>
</table>

Plan

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

As usual, let's continue with our recipe:

1. Run-time
2. Types
3. Transforms

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

Run-Time

We need to extend the run-time (\texttt{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:

```c
int main(int argc, char** argv) {
    int* HEAP = calloc(HEAP_SIZE, sizeof(int));
    int 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 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:

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

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

```c
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("\n");
        for(int i = 0; i < valp[i];
for i : valp[i]) {
        printf("\n");
        printRec("valp + 1");
        printf("\n");
        printf("Unknown value: %#010x", val);
    } else {
        printf("Unknown value: %#010x", val);
    }
}
```

### Types

Next, let's 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) --^ construct a pair
    | GetItem (Expr a) (Expr 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` to include pairs:

```haskell
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:

```haskell
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`

```haskell
prelude :: [Instruction]
prelude =
```
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**

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

```plaintext
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) -- 3. set the tag-bits of `eax`
```

Let's 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.

```plaintext
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.

```plaintext
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`.

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

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

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

Access

To access tuples, let's update compileExpr with the strategy above:

```plaintext
compileExpr env [GetItem e fld]
```
(x1, (x2, (x3, (x4, x5))))

**Accessing Tuples**

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

So the below code

```python
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))
```

should print out:

0
1
2
3
4
99

How shall we write it?

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

**QUIZ**

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

```python
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)
```

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

given 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[8]
```
QUIZ

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

```python
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:

```python
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.

```python
def is_empty(l):
    l == empty()
def head(l):
```
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 $j$

```python
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:

```python
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:

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

```python
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.