I started from the Egg-Eater lecture code:

https://github.com/ucsd-compilers-s23/lecture1/tree/egg-eater

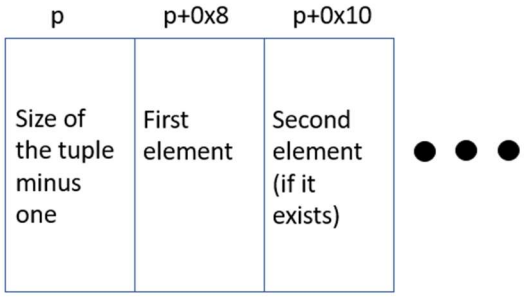
The concrete grammar of the new language is as follows:

```
<prog> := <defn>* <expr>
<defn> :=
      | (fun (<name> <name>) <expr>)
      | (fun (<name> <name> <name>) <expr>)
<expr> :=
      | <number>
      | true
      | false
      | nil                                                    (new!)
      | input
      | <identifier>
      | (let (<binding>) <expr>)
      | (<op1> <expr>)
      | (<op2> <expr>)
      | (set! <identifier> <expr>)
      | (if <expr> <expr> <expr>)
      | (block <expr>+)
      | (loop <expr>)
      | (break <expr>)
      | (<name> <expr>)
      | (<name> <expr> <expr>)
      | (pair <expr> <expr>)                                   (new!)
      | (tuple <expr>+)                                        (new!)
      | (lookup <expr> <expr>)                                 (new!)
<op1> := add1 | sub1 | print
<op2> := + | - | < | =
```

The new features, `nil`, `pair`, `tuple`, and `lookup`, allow for heap-allocated tuples (equivalent to vectors or lists in other languages) to be created and accessed in Egg-Eater, a language in the Snek family based off a subset of Diamondback. `nil` represents an empty tuple and is only in the language to enable compatibility with the tests provided with the lecture code. `pair` creates a tuple of length 2 and is also only in the language for compatibility purposes. `tuple` creates a heap-allocated tuple from the elements it is provided, and it can be instantiated from an arbitrary number of variables. `lookup` takes a tuple *t* and a number *n* as an argument, and it uses 0-based indexing to return the value at index *n* in *t*. It will dynamically throw an error if a tuple is not passed as the first argument, a number is not passed as the second argument, or if the number is out of bounds for the tuple.

The tuples are arranged on the heap as follows:

Let this tuple start at address *p*.
It's allocated in the heap as follows:



The value stored in the tuple's identifier is *p* + 1. We add 1 to conform with the tag rules.

When storing the tuple, we subtract 1 from the size to easily check if an lookup index is out of bounds for this tuple and my implementation of Egg-Eater uses 0-indexing. This size is a *size*, not an Egg-Eater number, so it is stored as its actual value instead of being shifted left arithmetically by 1.

The tag rules allow the program to identify during runtime whether a value at an address is a number, a Boolean value, or a pointer to a tuple. Numbers are 63-bit values stored in a 64-bit word with the least significant bit being 0, Booleans are stored as 0b111 for true, and 0b011 for false, and tuple pointers are stored as the pointer value + 1, so the pointer may be recovered while still being identifiable as a tuple pointer. This requires the pointer to be 8-byte aligned, which can be done fairly easily since a word is 64 bits, or 8 bytes.

Testing

The first test I wrote was `simple_examples.boa`:

```
(let (lst0 (tuple 2 2 2))
    (let (lst1 (tuple 0 1 lst0 3 4))
        (block
            (print (lookup lst1 0))
            (print (lookup lst1 1))
            (print (lookup lst1 2))
            (print (lookup lst1 3))
            (print (lookup lst1 4))
            (print lst1)
        )
    )
)
```

This test outputs the following:

```
0
1
(tuple 2 2 2)
3
4
(tuple 0 1 (tuple 2 2 2) 3 4)
(tuple 0 1 (tuple 2 2 2) 3 4)
```

which is the expected output. The reason I use two `let` statements instead of just putting `lst0`'s value inside of lst1 is that I wasn't able to implement tuples containing tuples that are allocated inside of that tuple. So in my implementation, a tuple can store tuples, it's just that the inner tuple's values need to be allocated outside of the outer tuple. The outer tuple just contains a pointer to the inner tuple.

The second test I wrote was `error-tag.boa`:

```
(let (lst (tuple 0 1 2)) (lookup lst true))
```

This test outputs the following when compiled and run:

```
akhil@Not-A-Laptop:~/cse-131-egg-eater$ make input/error-tag.run
cargo run -- input/error-tag.boa input/error-tag.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/egg-eater input/error-tag.boa input/error-tag.s`
nasm -f elf64 input/error-tag.s -o input/error-tag.o
ar rcs input/liberror-tag.a input/error-tag.o
rustc -L input/ -lour_code:error-tag runtime/start.rs -o input/error-tag.run
rm input/error-tag.s
akhil@Not-A-Laptop:~/cse-131-egg-eater$ ./input/error-tag.run
Called lookup with non-number as index
```

This is the expected output. `true` does not have a numerical value in Egg-Eater and hence cannot be used as an index in `lookup`. When we call `lookup` on `lst`, this error is detected dynamically via tag-checking the index parameter, and execution is halted with a meaningful message about the error is reported.

The third test I wrote was `error-bounds.boa`:

```
(let (lst (tuple 0 1 2))
    (lookup lst 7)
)
```

This test outputs the following when compiled and run:

```
akhil@Not-A-Laptop:~/cse-131-egg-eater$ make input/error-bounds.run
cargo run -- input/error-bounds.boa input/error-bounds.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/egg-eater input/error-bounds.boa input/error-bounds.s`
nasm -f elf64 input/error-bounds.s -o input/error-bounds.o
ar rcs input/liberror-bounds.a input/error-bounds.o
rustc -L input/ -lour_code:error-bounds runtime/start.rs -o input/error-bounds.run
rm input/error-bounds.s
akhil@Not-A-Laptop:~/cse-131-egg-eater$ ./input/error-bounds.run
lookup: index out of bounds
```

This is the expected output. `lst` only has 3 values, so its maximum index is 2 and its minimum index is 0. Therefore, 7 cannot be used as an index on it. When we call lookup on `lst`, this error is detected dynamically by comparing the index passed to `lookup`, shifted right by 1, to the value stored in the tuple's pointer: that is, the size of the tuple. A meaningful message for this error is printed in response.

The fourth test I wrote was `error3.boa`:

```
(let (lst (tuple 3 4 true))
    (+ (lookup lst 2) 1)
)
```

This test outputs the following when compiled and run:

```
akhil@Not-A-Laptop:~/cse-131-egg-eater$ make input/error3.run
cargo run -- input/error3.boa input/error3.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/egg-eater input/error3.boa input/error3.s`
nasm -f elf64 input/error3.s -o input/error3.o
ar rcs input/liberror3.a input/error3.o
rustc -L input/ -lour_code:error3 runtime/start.rs -o input/error3.run
rm input/error3.s
akhil@Not-A-Laptop:~/cse-131-egg-eater$ ./input/error3.run
Bad arguments to native function or operator
```

This is the expected output. This test mainly serves to check that `lookup` can return the correct value for its index. `lookup` successfully retrieves the value at index 2 of `lst`, `true`, but `+` only works on two numbers, which is a runtime error. Thus, this error is caught during runtime, leading to the output above.

The message isn't *as* meaningful this time, but this can be fixed fairly easily by passing a value explaining *which* native function or operator threw this error.

The fifth test I wrote was `points.boa`:

```
(fun (pointcr x y) (tuple x y))
(fun (pointsum a b) (tuple
    (+ (lookup a 0) (lookup b 0))
    (+ (lookup a 1) (lookup b 1))
))
(block
    (print (pointsum (pointcr 1 2) (pointcr 1 2)))
    (print (pointsum (tuple 3 4) (tuple 4 3)))
    (print (pointcr true 7))
    (print (let (a (tuple 3 8)) (pointcr 3 a)))
    (pointsum (pointcr 1 1) (pointcr 3 4))
)
```

When run, it gives the following output:

```
(tuple 2 4)
(tuple 7 7)
(tuple true 7)
(tuple 3 (tuple 3 8))
(tuple 4 5)
```

This is the expected output. Interestingly enough, the Egg-Eater lecture code's print function doesn't account for the call instruction incrementing rsp by 8, so the calls to print aren't always 16-byte aligned. I used a bit of a hacky way to get around this: storing `rsp` in `rbx` and calling `and rsp, -16`, as suggested by an Edstem post. It's not ideal, but it's a temporary fix. In PA6, if I am to use my own compiler, my Diamondback compiler does it properly anyway.

The sixth test I wrote was `bst.boa`:

```
(fun (addToBst tree val)
    (if (= tree nil)
        (tuple val nil nil)
        (if (< (lookup tree 0) val)
            (if (= (lookup tree 1) nil)
                (tuple (lookup tree 0) val (lookup tree 2))
                (tuple (lookup tree 0) (addToBst (lookup tree 1) val) (lookup tree 2))
            )
            (if (= (lookup tree 2) nil)
                (tuple (lookup tree 0) (lookup tree 1) val)
                (tuple (lookup tree 0) (lookup tree 1) (addToBst (lookup tree 2) val))
            )
        )
    )
)

(fun (searchBst bst val)
    (if (= bst nil)
        false
        (if (= (lookup bst 0) val)
            true
            (if (< val (lookup bst 0))
                (if (= (lookup bst 1) nil)
                    false
                    (searchBst (lookup bst 1) val)
                )
                (if (= (lookup bst 2) nil)
                    false
                    (searchBst (lookup bst 2) val)
                )
            )
        )
    )
)

(let (a (tuple 1 nil nil))
    (let (b (tuple 3 nil nil))
        (let (c (tuple 2 a b))
            (let (d (tuple 5 nil nil))
                (let (e (tuple 4 c d))
                    (block
                        (print e)
                        (print (searchBst e 5))
                        (print (searchBst e 6))
                        (set! e (addToBst e 6))
                        (print e)
                        (searchBst e 6)
                    )
                )
            )
        )
    )
)
```

It's supposed to implement BSTs in Egg-Eater, and test searching and adding for values. Here's the actual output of running the compiled program:

```
(tuple 4 (tuple 2 (tuple 1 nil nil) (tuple 3 nil nil)) (tuple 5 nil nil))
false
false
(tuple 6 nil nil)
false
```

The expected output is the following:

```
(tuple 4 (tuple 2 (tuple 1 nil nil) (tuple 3 nil nil)) (tuple 5 nil nil))
true
false
(tuple 4 (tuple 2 (tuple 1 nil nil) (tuple 3 nil nil)) (tuple 5 nil (tuple 6 nil nil)))
true
```

This behavior occurs because = always returns that `tree` is equal to `nil`. I'm reasonably sure that my BST implementation on the Snek side is accurate, and after over an hour I couldn't find any problems in `tuple`, `lookup`, or =, and at that point I decided to throw in the towel.

I know that in Python, tuples are heap-allocated, and in C++, `std::vector`s allocate memory on the heap by default. My language's design is closer to Python's tuples, though, since both my implementation and Python use immutable tuples, while C++ `std::vector`s are mutable. In addition, neither my implementation nor Python require that all tuple elements be of the same type, while C++ `std::vector`s require that every element in them has the same type, passed in as a generic type argument.

I used the following resources for my Egg-Eater implementation and this write-up:

https://github.com/ucsd-compilers-s23/lecture1/tree/egg-eater

https://edstem.org/us/courses/38748/discussion/3150044

https://stackoverflow.com/questions/10366474/where-does-a-stdvector-allocate-its-memory