Egg-Eater Report

Setup

I choose Diamondback as my starting point.

Egg-Eater

Concrete Syntax

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (<name> <expr>*)
  | (tuple <expr>+)                         (new!)
  | (index! <expr> <expr>)                  (new!)
  | (setindex! <name> <expr> <expr>)        (new!)
```

```
<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | = | .=
(new!)

<binding> := (<identifier> <expr>)
```

Abstract Syntax

```
pub enum Expr {
    ...
    Index(Box<Expr>, Box<Expr>),
    Tuple(Vec<Expr>),
    SetIndex(String, Box<Expr>, Box<Expr>),
}
```

Semantics

The new expressions have the following semantics:

- `(tuple <expr>+)` expressions evaluate every `<expr>` to a value and create a object on the heap. It contains any number of positions.
- `(index! <expr> <expr>)` expressions evaluate the first `<expr>` to a tuple `t` and the second `<expr>` to a index `i`. The `(index <expr> <expr>)` itself will be evaluated to the value at `i` position of the tuple as `t[i]`.
- `(setindex! <name> <expr> <expr>)` expressions evaluate the first `<expr>` to a index `i`, the second `<expr>` to a value `v`. And then it will change the value

store in the `i` position of tuple `<name>` to the new value `v` as `<name>[i] = v`. The `(setindex! <name> <expr> <expr>)` itself will be evaluated as the updated `<name>` .
- `.=` means **structural equality**, distinguishing it from `=`, which denotes **referential equality**. The outputs are same on `int` and `bool` values, but they are different on `tuple` value.

There are several examples further down to make this concrete.

The compiler should stop and report an static error if:

- There is no any `<expr>` in `(tuple <expr>+)` . The error should contains `empty tuple`.
- In `(setindex! <name> <expr> <expr>)`, identifier `<name>` is unbound (there is no surrounding let binding for it) The error should contain the string `"Unbound variable identifier <name>"`

The compiler should stop and report an runtime error if:

- In `(index! <expr> <expr>)`, the first `<expr>` is not a `tuple` type or the second `<expr` is not a `int` type. The error should contains `invalid argument`.
- In `(index! <expr> <expr>)`, the first `<expr>` is not a `tuple` type or the second `<expr` is not a `int` type. The error should contains `invalid argument`.
- In `(setindex! <name> <expr> <expr>)`, the first `<expr>` is not a `int` type. The error should contains

`invalid argument`.

Example 1

```
(let ((x (tuple 1 2 (tuple 3 4))))
    (block
        (print x)
        (index! x (+ 1 1))
    )
)
```

output

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

Example 2

```
(let ((x (tuple 1 2 3)) (y (tuple 1 2 4)))
    (block
        (print x)
        (print y)
        (print (= x y))
        (print (.= x y))
        (setindex! x 2 x)
        (setindex! y 2 x)
        (print x)
        (print y)
        (print (= x y))
        (.= x y)
    )
)
```

output

```
(tuple 1 2 3)
(tuple 1 2 4)
false
false
(tuple 1 2 ...)
(tuple 1 2 (tuple 1 2 ...))
false
true
```

Type Representation

We have 3 types in Egg-Eater: `int`, `bool` and `tuple`. For easier type checking, I change the `int` from signed 63 bits integer to signed 62 bits integer.

The general data representation form is

data[63:2]	type[1:0]
62bits	2bits

For specific,

`int`

i62[63:2]	00
-2^{61} ~ $2^{61}-1$	type bits

`bool`

0[63:3]	1	01
	True	

0[63:3]	0	01
	False	

`tuple`

address[63:2]	10
starting address of the tuple on the heap	

Heap Allocation

The position 0 is always store the length of tuples. So a tuple with `k` elements will be allocated `k+1` positions on the heap.

Take `(tuple 4 5 6)` as an example:

assembly

```
our_code_starts_here:
    sub rsp,16
    jo overflow_label
;===== tuple begin =====
    mov qword [rsp + 0],16      ;the first
position  16(4)
    mov qword [rsp + 8],20      ;the second
position 20(5)
```

```
    mov rax,24                      ;the third
position  24(6)
                                    ;the length of
the tuple is 3.
    mov qword [r15 + 0],12          ;Store length=3
at index 0
    mov qword [r15 + 24],rax        ;Store 12(3) at
index 3
    mov rbx,[rsp + 0]
    mov [r15 + 8],rbx               ;Store 4(1) at
index 1
    mov rbx,[rsp + 8]
    mov [r15 + 16],rbx              ;Store 8(2) at
index 2
    mov rax,r15
    or  rax,2                       ;set the type
bits of the tuple to 0b10
    add r15,32                      ;allocate 4
positions (3+1) on the heap
    jo overflow_label
;===== tuple end =====
    add rsp,16
    jo overflow_label
    ret
```

diagram of the heap

rax	0x102
r15	0x128

	heap
0x100	12(3=len)
0x108	16(4)
0x110	20(5)
0x120	24(6)

Testing

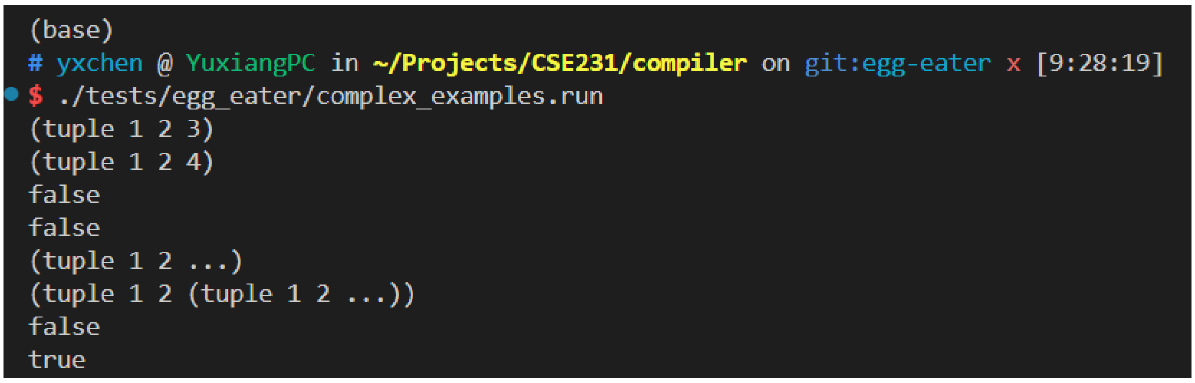
complex_example.boa

```
(let ((x (tuple 1 2 3)) (y (tuple 1 2 4)))
    (block
        (print x)
        (print y)
        (print (= x y))
        (print (.= x y))
        (setindex! x 2 x)
        (setindex! y 2 x)
        (print x)
        (print y)
        (print (= x y))
        (.= x y)
    )
)
```

expect

```
(tuple 1 2 3)
(tuple 1 2 4)
false
false
(tuple 1 2 ...)
(tuple 1 2 (tuple 1 2 ...))
false
true
```

output



```
(base)
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:28:19]
$ ./tests/egg_eater/complex_examples.run
(tuple 1 2 3)
(tuple 1 2 4)
false
false
(tuple 1 2 ...)
(tuple 1 2 (tuple 1 2 ...))
false
true
```

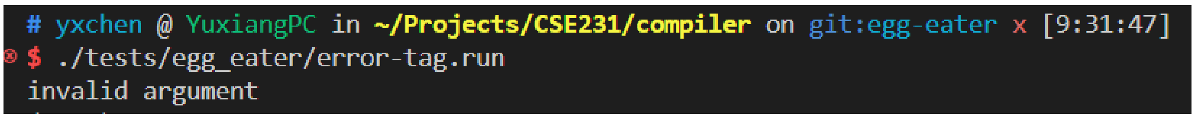
error-tag.boa

```
(let ((x 1))
    (index! x 0)
)
```

expect error

```
invalid argument
```

output



```
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:31:47]
$ ./tests/egg_eater/error-tag.run
invalid argument
```

tracing

```
our_code_starts_here:
    sub rsp,16
    jo overflow_label
;===== binding begin =====
; x
    mov qword [rsp + 0],4
;===== binding end =====
;===== index begin =====
;load x
    mov rbx,[rsp + 0]
    mov [rsp + 8],rbx
    mov rbx,3
    and rbx,[rsp + 8]    ;get the type bits
    cmp rbx,2            ;check tuple
    jne invalid_label    ;<= catch here
    ...
```

```
Expr::Index(tuple_expr, index) => {
    let mut result = vec![Instr::Info("=====
index begin =====".to_string())];
    // put tuple ptr in stack
    ...

    // validate ptr

 result.push(Instr::TupleGuard(Val::RegOffset(R
eg::RSP, state.si))); //<= cache here
    ...
}
```

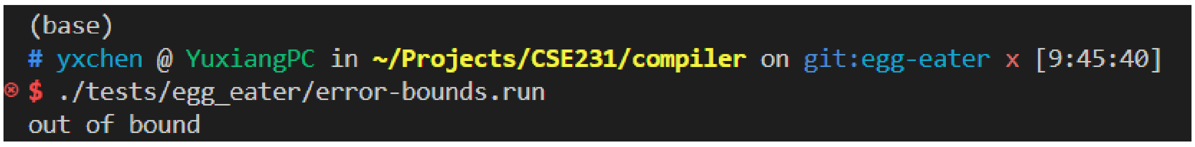
error-bound.boa

```
(let ((x (tuple 1)))
    (index! x 1)
)
```

expect error

```
out of bound
```

output



```
(base)
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:45:40]
$ ./tests/egg_eater/error-bounds.run
out of bound
```

tracing

```
...
our_code_starts_here:
    mov rbx,[rsp + 0]
    mov [rsp + 8],rbx
    mov rbx,3
    and rbx,[rsp + 8]
    cmp rbx,2
    jne invalid_label
    ...
    mov rbx,[rsp + 8]        ; load addr in rbx
    and rbx,-4               ; remove type bits
    mov rbx,[rbx]            ; [addr] = length
    cmp rbx,rax              ; compare length
and index (in rax)
    jle out_of_bound_label   ; <= catch here
    ...
```

```
Expr::Index(tuple_expr, index) => {
    let mut result = vec![Instr::Info("=====
index begin =====".to_string())];
    // put tuple ptr in stack
    ...

    // validate ptr

 result.push(Instr::TupleGuard(Val::RegOffset(R
eg::RSP, state.si)));

    // put index in rax
    ...

    result.extend([
        // validate index
        Instr::IntGuard(Val::Reg(Reg::RAX)), //
<= cache here
        ...
    ]);
    Ok(result)
}
```

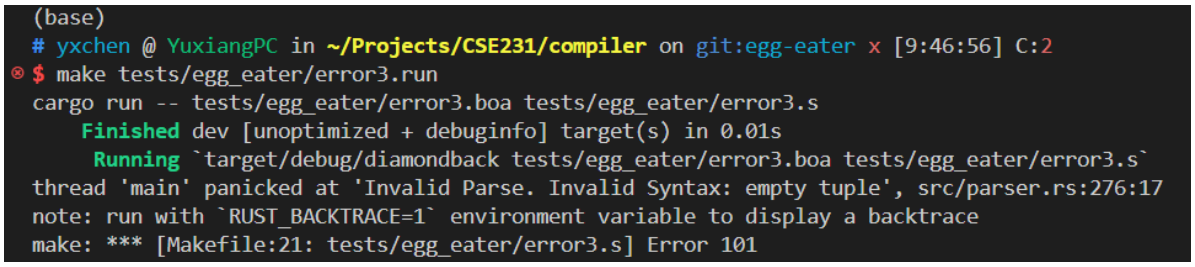
error3.boa

```
(let ((x (tuple)))
    x
)
```

expect

```
empty tuple
```

output



```
(base)
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:46:56] C:2
⊗ $ make tests/egg_eater/error3.run
cargo run -- tests/egg_eater/error3.boa tests/egg_eater/error3.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/diamondback tests/egg_eater/error3.boa tests/egg_eater/error3.s`
thread 'main' panicked at 'Invalid Parse. Invalid Syntax: empty tuple', src/parser.rs:276:17
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
make: *** [Makefile:21: tests/egg_eater/error3.s] Error 101
```

tracing

```
// op <expr>+ => Block/Tuple
[Sexp::Atom(S(op)), rest @ ..] if matches!
(op.as_str(), "block" | "tuple") => {
    let result = rest
        .iter()
        .map(parse_expr)
        .collect::<Result<Vec<Expr>, String>>
()?;

    if !result.is_empty() {
        Ok(if op == "block" {
            Expr::Block(result)
        } else {
            Expr::Tuple(result)
        })
    } else {
        Err(format!("Invalid Syntax: empty {}",
op)) // <= cache here
    }
}
```

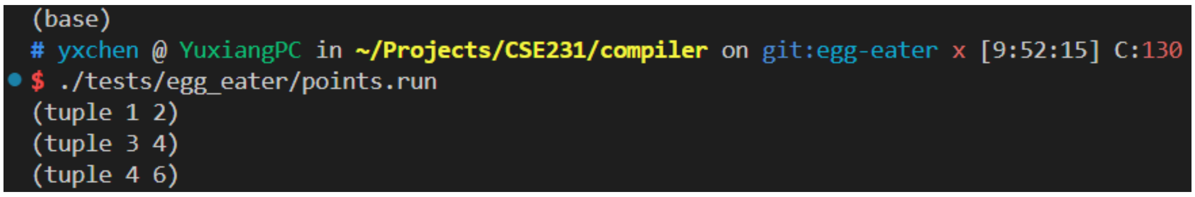
points.boa

```
(fun (create_point x y)
  (tuple x y))

(fun (add_points p1 p2)
  (tuple (+ (index! p1 0) (index! p2 0))
         (+ (index! p1 1) (index! p2 1))))

(let ((point1 (create_point 1 2))
      (point2 (create_point 3 4)))
  (block
    (print point1)
    (print point2)
    (add_points point1 point2)))
```

expect

```
(tuple 1 2)
(tuple 3 4)
(tuple 4 6)
```

output



```
(base)
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:52:15] C:130
$ ./tests/egg_eater/points.run
(tuple 1 2)
(tuple 3 4)
(tuple 4 6)
```

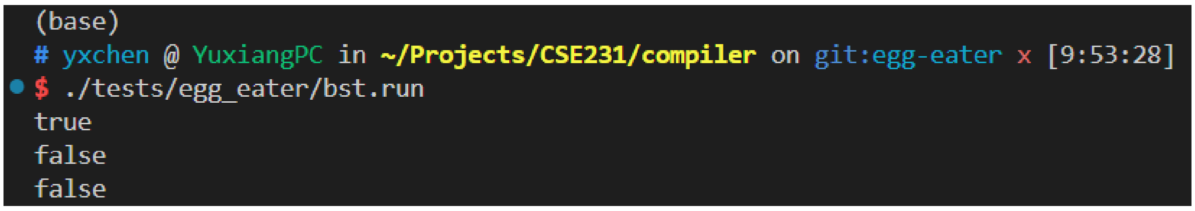
bst.boa

```
(fun (add_element root value)
    (if (.= root (tuple false false false))
        (setindex! root 0 value)
        (let ((node_value (index! root 0))
                (left_node (index! root 1))
                (right_node (index! root 2)))
            (if (< value node_value)
                (if (isbool left_node)
                    (setindex! root 1 (tuple
value false false))
                    (add_element left_node
value)
                )
                (if (isbool right_node)
                    (setindex! root 2 (tuple
value false false))
                    (add_element right_node
value)
                )
            )
        )
    )
)

(fun (search_element root value)
    (if (isbool root)
        false
        (let ((node_value (index! root 0))
                (left_node (index! root 1))
                (right_node (index! root 2)))
            (if (= value node_value)
                true
                (if (< value node_value)
                    (search_element left_node
value)
                    (search_element right_node
value)
                )
            )
        )
    )
)

(let ((bst (tuple false false false)))
    (block
        (add_element bst 5)
        (add_element bst 3)
        (add_element bst 7)
        (add_element bst 1)
        (add_element bst 4)
        (print (search_element bst 4))
        (print (search_element bst 6))
        (search_element bst 8)
    )
)
```

expect

```
true
false
false
```

output



```
(base)
# yxchen @ YuxiangPC in ~/Projects/CSE231/compiler on git:egg-eater x [9:53:28]
$ ./tests/egg_eater/bst.run
true
false
false
```

Language Comparison

Python

In Python, tuples are immutable sequences, and by default, they are allocated on the heap. Since tuples are immutable, their memory allocation remains fixed after creation.

Here's an example of creating and accessing elements of a tuple in Python:

```python
# Create a tuple
my_tuple = (1, 'hello', 2.0)

# Access elements of the tuple
print("First element:", my_tuple[0])
print("Second element:", my_tuple[1])
print("Third element:", my_tuple[2])
```

Java

In Java, all objects are allocated on the heap by default. The `new` keyword is used to dynamically allocate memory for objects on the heap. Java provides automatic memory management through garbage collection, which handles deallocation of objects when they are no longer in use. In Java, tuples are not built-in data structures like in some other programming languages.

Egg-Eater

Egg-Eater is more similar to Python than Java due to the former's affinity for weak typing, in contrast to Java's strong typing. Hence, put objects of different types into one structure is much easier in Python and Egg-Eater.

However, the mutability of tuples in Egg-Eater distinguish them from their counterparts in Python. Due to the uniform 8 bytes size of `int` , `bool` and `tuple`, it is possible for Egg-Eater to update an element of a tuple stored on the heap without any change of the heap allocation. Conversely, objects in Python exhibit distinct sizes, thereby modifying tuples requires a reallocation of the heap. So tuples are set immutable in Python.