

1.The concrete grammar of your language, pointing out and describing the new concrete syntax beyond Diamondback/your starting point. Graded on clarity and completeness (it' s clear what' s new, everything new is there) and if it' s accurately reflected by your parse implementation.

```

<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!)
    | (settuple! <expr> <expr> <expr>)
<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | = | == (new! Represent structural
equal)
<binding> := (<identifier> <expr>)

```

Binary representation:

Boolean: 0x0000 0000 0000 0003(false) 0x0000 0000 0000 0007(true)

Number: 0x???? ???? ???? ???0 + 0b????0

Tuple: 0x???? ???? ???? ???0 + 0b???01

2.A diagram of how heap-allocated values are arranged on the heap, including any extra words like the size of an allocated value or other metadata. Graded on clarity and completeness, and if it matches the implementation of heap allocation in the compiler.

To save a tuple with n elements, I allocate (n+1) * 8 bytes to save

it. The length of the elements(count by byte) is saved in the first 8 byte, and the n elements is saved in the following $n * 8$ bytes.

For example, (tuple 1 2 3 4) is saved as following:

0x20	0x2	0x4	0x6	0x8
------	-----	-----	-----	-----

Each cell in the table above represent 64bit.

3.The required tests above. In addition to appearing in the code you submit, they should be in the PDF). These will be partially graded on your explanation and provided code, and partially on if your compiler implements them according to your expectations.

simple_tests

```
(let ((i 0) (a (tuple 0 1 2 3 4)))
  (block
    (print (index a 0))
    (print (index a 4))
    (print (if (> i (index a 2)) true false))
    (loop
      (if (> i (index a 4))
        (break i)
        (set! i (add1 i)))
      )
    )
  )
)
```

This test case will test if the tuple can be correctly printed and accessed in control-flow.

Output:

0

4

false

5

error-tag

```
(index false 0)
```

This test case simply tests if an error is thrown while using index to access a non-tuple value.

Output:

Runtime error: access the index of an non-tuple val

error-bounds

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

This test case tests if an error is thrown while index is out-of-bound.

Output:

Runtime error: index is out-of-bound

error3

```
Test1:(index (tuple 1 2 3 4) -1)
```

```
Test2:(index (tuple 1 2 3 4) true)
```

This test case tests if an error is thrown while index is not non-negative number.

Output:

Runtime error: index is not non-negative number

Points

```
(fun (generate_tuple x y) (tuple x y))
(fun (add_tuple tuple1 tuple2)
  (tuple
    (+ (index tuple1 0) (index tuple2 0))
    (+ (index tuple1 1) (index tuple2 1))
  )
)

(block
  (print (add_tuple (generate_tuple 1 2) (generate_tuple 3 4)))
  (print (add_tuple (tuple 10 12) (generate_tuple 3 4)))
  (add_tuple (generate_tuple 1 2) (tuple 7 10))
)
```

It tests that function generate takes an x and a y coordinate and produces a structure with those values, and function add_tuple takes two points and returns a new point with their x and y coordinates added

together, along with three tests that print example output from calling these functions.

Output:

```
(tuple 4 6)
(tuple 13 16)
(tuple 8 12)
```

bst

```
(fun (generate_node val left right) (tuple val left right))
(fun (element_in_the_tree head val)
  (let ((node head))
    (loop
      (if (= val (index node 0))
        (break true)
        (if (> val (index node 0))
          (if (index node 2)
            (set! node (index node 2))
            (break false)
          )
          (if (index node 1)
            (set! node (index node 1))
            (break false)
          )
        )
    )
  )
)
```

```
(let
  ((node1 (generate_node 9 false false))
   (node2 (generate_node 5 false node1))
   (node3 (generate_node 17 false false))
   (node4 (generate_node 15 false node3))
   (node5 (generate_node 18 node4 false))
   (node6 (generate_node 12 node2 node5)))
  (block
    (print (element_in_the_tree node6 5))
    (print (element_in_the_tree node6 9))
    (print (element_in_the_tree node6 17))
  )
)
```

```

    (element_in_the_tree node6 999)
  )
)

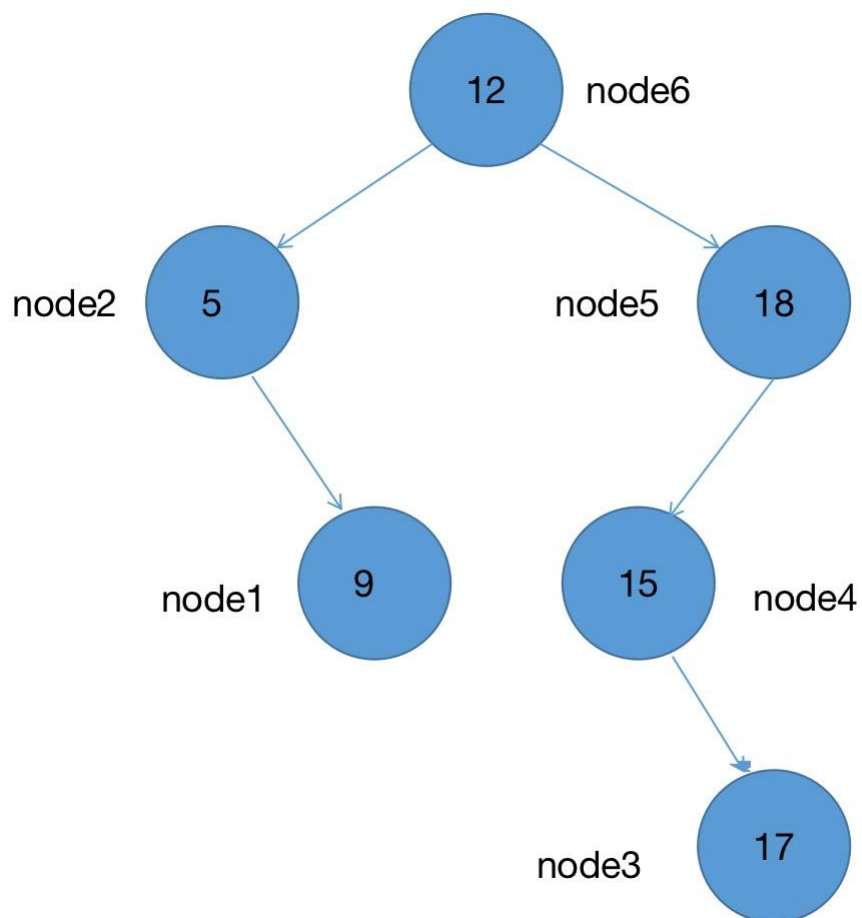
```

In the bst, each node is represented by tuple as following.

Tuple metadata	Node val	Left node	Right node
----------------	----------	-----------	------------

I use false as null to represent that there is no left/right node.

Because updating tuple is not supported, in this test case, I manually build a BST in let bindings instead of writing a function to insert nodes, and the BST looks like this:



Output:

```

true
true
true
false

```

4. Pick two other programming languages you know that support heap-allocated data, and describe why your language's design is more like one than the other.

My language is more like python than c. To allocate heap with c, you have to manage heap manually with malloc and free, while my language and python automatically allocate a space for the given data.

5. A description of your approach to handling structural equality, including relevant snippets of Rust or generated assembly

In my language, operator == is used to judge structural equality, which is parsed by the following code:

```
[Sexp::Atom(S(op)), e1, e2, e3] if op == "settuple!" =>
Expr::SetTuple(e1.to_string(), Box::new(parse_expr(e2)),
Box::new(parse_expr(e3)))
```

To judge structural equality, I use a call to rust code during compilation. The code is as following:

```
Op2::StructuralEqual => {
let mut newsi = si;
if newsi & 1 == 0 {newsi += 1}; // align rsp to 16 bytes
t.push(Instr::IMov(Val::Reg(Reg::RDI), Val::Reg(Reg::RAX)));
t.push(Instr::IMov(Val::Reg(Reg::RSI),
Val::RegOffsetMinus(Reg::RSP, si * 8)));

t.push(Instr::ISub(Val::Reg(Reg::RSP), Val::Imm(8 * newsi as u64)));
t.push(Instr::ICall("snek_structural_equal".to_string()));
t.push(Instr::IAdd(Val::Reg(Reg::RSP), Val::Imm(8 * newsi as u64)));
},
```

If a value is not a tuple, the result will be the same as reference equality. Otherwise, check structural equality of each value in the tuple.

```
#[export_name = "\x01snek_structural_equal"]
pub extern "C" fn snek_structural_equal(val1: i64, val2: i64) -> i64
{
unsafe{snek_structural_equal_helper(val1, val2, &mut
```

```

HashSet::new())}
}

unsafe fn snek_structural_equal_helper(val1: i64, val2: i64, seen:
&mut HashSet<i64>) -> i64 {
// println!("comparing {val1} {val2}");
if val1 == val2 {return 7}
if (val1 & 3) != 1 || (val2 & 3) != 1 {return 3}

if seen.contains(&val1) || seen.contains(&val2) {
    println!("Can't resolve recursive structural equality {val1}
{val2}.");
    return 3;
}

// both val1 and val2 are tuple
let mut address1: *const i64 = (val1 >> 2) as *const i64;
let length1 = *address1 / 8;
let mut address2: *const i64 = (val2 >> 2) as *const i64;
let length2 = *address2 / 8;
if length1 != length2 {return 3}

for _i in 1..=length1 {
    address1 = (address1 as i64 + 8) as *const i64;
    address2 = (address2 as i64 + 8) as *const i64;
    seen.insert(*address1);
    seen.insert(*address2);
    if snek_structural_equal_helper(*address1, *address2, seen) == 3
{return 3}
    seen.remove(&*address1);
    seen.remove(&*address2);
}
return 7
}

```

6.A description of your approach to handling cycles, including relevant snippets of Rust or generated assembly

As shown in the last question, I use a hashset to record seen tuples. If the tuple has been seen, the program will print "Can't resolve recursive structural equality {val1} {val2}." and return false.

If there is cycles in print, the program will print "[...]" to represent a seen tuple. The code is as following:

```

if t == "tuple".to_string() {
  if seen.contains(&content) {return "...".to_string()}
  seen.insert(content);
  let mut s = "(tuple".to_string();
  unsafe {
    let mut address: *const i64 = (content >> 2) as *const i64;
    let length = *address / 8;
    // println!("length{length}");
    for _i in 1..=length {
      address = (address as i64 + 8) as *const i64;
      let val = *address;
      s += &" ";
      s += &get_real_content(val, seen);
    }
  }
  seen.remove(&content);
  s += &")";
  s.to_string()
}

```

7. Show each of your required tests' code and output when compiled and run

equal.snek

```

(let ((x1 1) (x2 1) (x3 (tuple 1 2)) (x4 (tuple 1 2)) (x5 (tuple 1
2)) (x6 (tuple 1 2 3)) (x7 (tuple 1 2)) (x8 (tuple 1 2)))
  (block
    (settuple! x5 1 x3)
    (settuple! x6 1 x4)
    (settuple! x7 1 x7)
    (settuple! x8 1 x8)
    (print (= x1 x2))
    (print (= x3 x4))
    (print (= x5 x6))
    (print (= x7 x8))
    (print (== x1 x2))
    (print (== x3 x4))
    (print (== x5 x6))
    (print (== x7 x8))
  )
)

```

Result:


```
true
false
false
false
true
true
false
Can't resolve recursive structural equal 561544425505185
561544425505281.
False
```

cycle-print1.snek

```
(let ((x (tuple 1 2 3)))
  (block
    (settuple! x 2 x)
    x
  )
)
```

```
(tuple 1 2 [...])
```

cycle-print2.snek

```
(let ((x (tuple 1 2 3)) (y (tuple 4 5 x)))
  (block
    (settuple! x 2 y)
    x
  )
)
```

```
(tuple 1 2 (tuple 4 5 [...]))
```

cycle-print3.snek

```
(let ((x (tuple 1 2 3)) (y (tuple 4 5 x)))
  (block
    (settuple! x 2 y)
    (settuple! y 1 y)
    x
  )
)
```

```
(tuple 1 2 (tuple 4 [...] [...]))
```

cycle-equal1.snek

```
(let ((a (tuple 1 2 3)) (b (tuple 1 2 3)))  
  (block  
    (settuple! a 2 a)  
    (== a b)  
  )  
)
```

False

cycle-equal2.snek

```
(let ((a (tuple 1 2 3)) (b (tuple 1 2 3)))  
  (block  
    (settuple! a 2 a)  
    (settuple! b 2 b)  
    (== a b)  
  )  
)
```

Can't resolve recursive structural equality 561626029883393
561626029883521.
false

cycle-equal3.snek

```
(let ((a (tuple 1 2 3)) (b (tuple 1 2 3)))  
  (block  
    (settuple! a 2 a)  
    (settuple! b 2 a)  
    (== a b)  
  )  
)
```

true

8. Describe any features other than structural equality and update that you improved since your egg-eater submission

Fixed a bug in a bit operation for error checking while compiling index
// check if index is non-negative number

```

t.push(Instr::IMov(Val::Reg(Reg::RBX),
Val::RegOffsetMinus(Reg::RSP, si * 8)));
t.push(Instr::ITest(Val::Reg(Reg::RBX), Val::Imm(1)));
t.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::Imm(ERROR_INDEX_NOT_NONNEGATIVE_NUMBER)));
t.push(Instr::IJnz(Val::Mark("snek_error".to_string()))); //
detect non-number
t.push(Instr::ICmp(Val::Reg(Reg::RBX), Val::Imm(0)));
t.push(Instr::IJl(Val::Mark("snek_error".to_string()))); //dect
non-negative

```

Added settuple! operation, and the concrete syntax of it is:
SetTuple(String, Box<Expr>, Box<Expr>)
The first expr is the index of tuple, and the second is the new value.
The compilation of it is as following:

```

Expr::SetTuple(identifier, place, val) => {
let mut t = vec![];
t.append(&mut compile_to_instrs(place, si, env, ifnum, loopnum,
breaknum, funnames));
t.push(Instr::IShl(Val::Reg(Reg::RAX), Val::Imm(2)));
t.push(Instr::IMov(Val::RegOffsetMinus(Reg::RSP, si * 8),
Val::Reg(Reg::RAX)));
t.append(&mut compile_to_instrs(val, si + 1, env, ifnum, loopnum,
breaknum, funnames));
t.push(Instr::IMov(Val::RegOffsetMinus(Reg::RSP, si * 8 + 8),
Val::Reg(Reg::RAX)));
// check identifier is tuple
let identifier_offset = env.get(identifier);
match identifier_offset {
Some(identifier_offset) =>
t.push(Instr::IMov(Val::Reg(Reg::RAX),
Val::RegOffsetMinus(Reg::RSP, identifier_offset * 8))),
None => panic!("Unbound variable identifier {identifier}"),
}
// check if tuple is really tuple
t.push(Instr::IMov(Val::Reg(Reg::RBX), Val::Reg(Reg::RAX)));
t.push(Instr::IAnd(Val::Reg(Reg::RBX), Val::Imm(3)));
t.push(Instr::ICmp(Val::Reg(Reg::RBX), Val::Imm(1)));
t.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::Imm(ERROR_TAG_CHECKING)));
t.push(Instr::IJne(Val::Mark("snek_error".to_string())));

t.push(Instr::IShr(Val::Reg(Reg::RAX), Val::Imm(2))); // get the

```

address of tuple

```
// check if index is out of bound
t.push(Instr::IMov(Val::Reg(Reg::RBX), Val::RegOffsetPlus(Reg::RAX,
0)));
t.push(Instr::ICmp(Val::Reg(Reg::RBX),
Val::RegOffsetMinus(Reg::RSP, si * 8)));
t.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::Imm(ERROR_OUT_OF_BOUND)));
t.push(Instr::IJle(Val::Mark("snek_error".to_string())));

// check if index is non-negative number
t.push(Instr::IMov(Val::Reg(Reg::RBX),
Val::RegOffsetMinus(Reg::RSP, si * 8)));
t.push(Instr::ITest(Val::Reg(Reg::RBX), Val::Imm(1)));
t.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::Imm(ERROR_INDEX_NOT_NONNEGATIVE_NUMBER)));
t.push(Instr::IJnz(Val::Mark("snek_error".to_string()))); //
detect non-number
t.push(Instr::ICmp(Val::Reg(Reg::RBX), Val::Imm(0)));
t.push(Instr::IJl(Val::Mark("snek_error".to_string()))); // detect
non-negative

t.push(Instr::IAdd(Val::Reg(Reg::RAX), Val::Imm(8)));
// get the result
t.push(Instr::IMov(Val::Reg(Reg::RBX),
Val::RegOffsetMinus(Reg::RSP, si * 8)));
t.push(Instr::IAdd(Val::Reg(Reg::RBX), Val::Reg(Reg::RAX)));
t.push(Instr::IMov(Val::Reg(Reg::RAX),
Val::RegOffsetMinus(Reg::RSP, si * 8 + 8)));
t.push(Instr::IMov(Val::RegOffsetPlus(Reg::RBX, 0),
Val::Reg(Reg::RAX)));
t
}
```

Update bst.snek to support adding node with settuple!

```
(fun (generate_node val left right) (tuple val left right))
(fun (element_in_the_tree head val)
  (let ((node head))
    (loop
      (if (= val (index node 0))
```

```

        (break true)
      (if (> val (index node 0))
        (if (index node 2)
          (set! node (index node 2))
          (break false)
        )
        (if (index node 1)
          (set! node (index node 1))
          (break false)
        )
      )
    )
  )
)

(fun (add node val)
  (if (> val (index node 0))
    (if (index node 2)
      (add (index node 2) val)
      (settuple! node 2 (generate_node val false false))
    )
    (if (index node 1)
      (add (index node 1) val)
      (settuple! node 1 (generate_node val false false))
    )
  )
)

(let ((head (generate_node 12 false false)))
  (block
    (add head 9)
    (add head 5)
    (add head 15)
    (add head 18)
    (add head 17)
    (print (element_in_the_tree head 5))
    (print (element_in_the_tree head 9))
    (print (element_in_the_tree head 17))
    (element_in_the_tree head 999)
  )
)

```

Result:

true
true
true
false

9. A list of the resources you used to complete the assignment, including message board posts, online resources (including resources outside the course readings like Stack Overflow or blog posts with design ideas), and students or course staff discussions you had in-person. Please do collaborate and give credit to your collaborators.

No resource. Everything is done by myself.