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

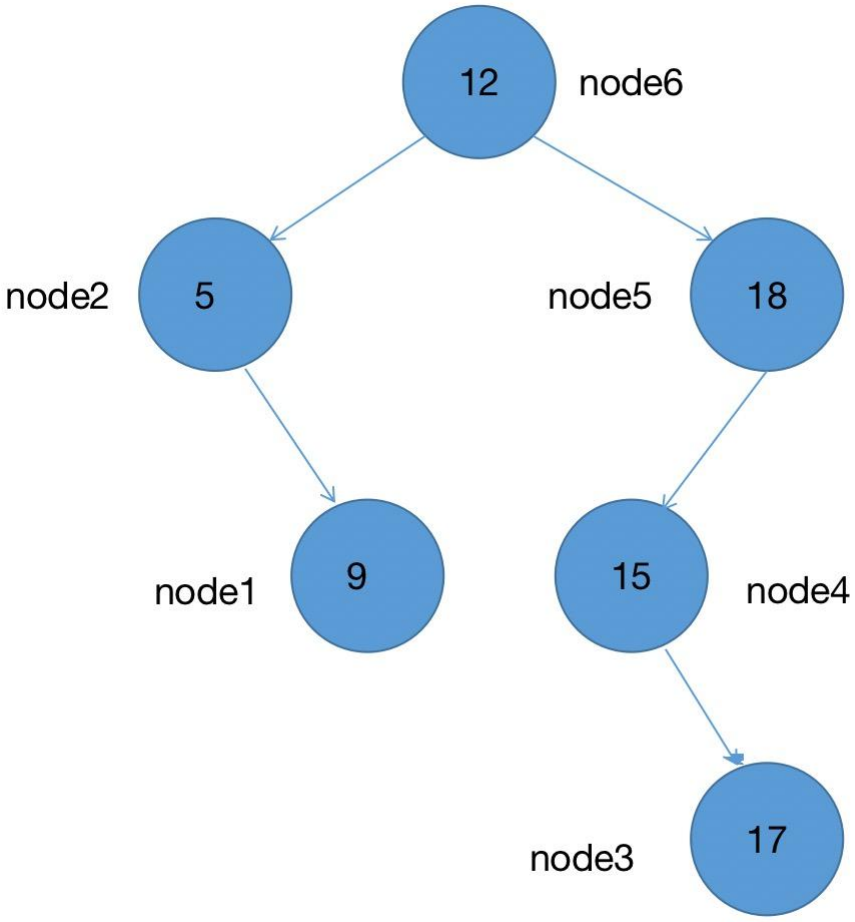
)

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 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.