Concrete Syntax

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | nil
  | 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>*)
  | (tuples <expr>+)

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

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

Abstract Syntax

```rust
struct Program {
   defs: Vec<Definition>,
   main: Expr,
}

enum Definition {
   Fun(String, Vec<String>, Box<Expr>),
}

enum Expr {
   Number(i64),
   Boolean(bool),
   Id(String),
   Let(Vec<(String, Expr)>, Box<Expr>),
   UnOp(Op1, Box<Expr>),
   BinOp(Op2, Box<Expr>, Box<Expr>),
   If(Box<Expr>, Box<Expr>, Box<Expr>),
   Loop(Box<Expr>),
   Break(Box<Expr>),
   Set(String, Box<Expr>),
   Block(Vec<Expr>),
   Call(String, Vec<Expr>),
   Tuples(Vec<Expr>),
   Vset(Box<Expr>, Box<Expr>,Box<Expr>),
}
```

Semantics
<number> | true | false are value

They should evaluate to themselves.

true -> true
false -> false
10 -> 10

nil is a value that uses for heap-allocated data

<identifier> should look up the variable in the environment and find its value

let binding should be given a list of binding and correctly match variables with the value given and store them in the environment for future reference

(let ((x 10) (y 11) (z 12)) x) -> 10

UnOp
add1 | sub1 adding 1 to the number
isNum | isBool return true is type match else return false
Print print the content of the expression

BinOp
plus | minus | times arithmetic operation on two value
(+ 1 2) -> 3

Equal | Greater | GreaterEqual | Less | LessEqual | Eq compare two value and return true if logic is satisfied(Eq is for comparing heap-allocated object)

(= 1 2) -> false

index returns the value at the specific index in a list

(index (tuples 1 2) 0) -> 1

if statement similar to if statement in other languages
(if true 1 2) -> 1

loop | block |break | set

loop creates an infinite loop to execute the context inside until a break is called
block evaluate the subexpressions in order
break exist out of the infinite loop
set modify the value in a variable

```
(let ((a 2) (b 3) (c 0) (i 0) (j 0))
  (loop
    (if (< i a)
      (block
        (set! j 0)
        (loop
          (if (< j b)
            (block (set! c (sub1 c)) (set! j (add1 j)))
            (break c)
          )
        )
        (set! i (add1 i))
      )
      (break c)
    )
  )
)

-> -6
```

fun | call
fun will create a function that can be called by the main program and the call will invoke the function with a specific argument

```
(fun (isodd n)
  (if (< n 0)
      (isodd (- 0 n))
      (if (= n 0)
          false
          (iseven (sub1 n))
      )
  )
)

(fun (iseven n)
  (if (= n 0)
      true
      (isodd (sub1 n))
  )
)

(block
  (print input)
  (print (iseven input))
)

-> 5 false
```

tuples create a heap-allocated data

```
(tuples 1 2 3 4 5) -> (1 2 3 4 5)
```