Egg-Eater Specification

Akhilan Ganesh, A15975349

Concrete Syntax

The concrete syntax of a Snek program is shown below. This includes but is not limited to integers, booleans, variables, let bindings, unary operators, binary operators, conditional and control expressions, functions, and function calls.

```
 <prog> := <defn>* <expr>
<defn> := (fun (<fname> <identifier>*) <expr>)
<expr> :=
  | <integer>
  | <boolean>
  | <tuple>                     <-- new!
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (if <expr> <expr> <expr>)
  | (loop <expr>)
  | (break <expr>)
  | (set! <identifier> <expr>)
  | (tinit <expr:integer> <expr>)               <-- new!
  | (tset <expr:tuple> <expr:integer> <expr>)   <-- new!
  | (tget <expr:tuple> <expr:integer>)          <-- new!
  | (block <expr>+)
  | (<fname> <expr>*)

<integer>    := (-)?[0-9]*
<boolean>    := true | false
<tuple>      := (tuple <expr>*)                     <-- new!
<expr:[value]>  := <expr> that holds type [value]   <-- new!

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

<fname>      := [a-zA-z][a-zA-Z0-9]*
<identifier> := [a-zA-z][a-zA-Z0-9]*
<binding>    := (<identifier> <expr>)
```

Note that integers must be within the bounds -2^{62} to $2^{62} - 1$. `input` refers to an optional argument provided at runtime of the Snek binary.

New Syntax

1. The `tuple` construct. Declared above as `<tuple>`, the default instantiation of a tuple is of the form `(tuple <expr>*)`. This operation returns a tuple value that refers to contiguous heap-allocated data maintaining the order of the elements from left-to-right. Note that any expression type is possible, as well as a mix of different types (integers, booleans, tuples). These values are exactly stored as an element on the heap-allocated construct. Additionally, with this syntax the zero-length tuple can also be constructed, which has no elements.
2. The `tinit` operation. This is the non-standard tuple instantiation operation, taking the form `(tinit <expr:integer> <expr>)`. Here the first expression represents the intended length of the tuple, and must be of type integer (which is runtime checked). The second expression is the intended default value to store as all elements of the tuple, and can be of any type. This operation returns a tuple that refers to contiguous heap-allocated data of the specified length and all elements of the specified default value.
3. The `tset` operation. This is the tuple update operation, and is of the form `(tset <expr:tuple> <expr:integer> <expr>)`. The first expression must be of type tuple (runtime checked) and stores the tuple value to update. The second expression is the index of the element to be updated, and must be of type integer (runtime checked). The last expression is the new value for the element, and can be of any type. This operation returns the same tuple (same heap-allocated data) but with an element changed.
4. The `tget` operation. This is the tuple lookup operation, and is of the form `(tget <expr:tuple> <expr:integer>)`. The first expression must be of type tuple (runtime checked) and stores the tuple value to lookup the element from. The second expression is the lookup index, and must be of type integer (runtime checked). This operation returns the value at the specified index for the specified tuple.
5. An update to the `print` operation. This operation handles tuples by printing out each element of the tuple separated by commas and enclosed by parentheses. For example: `(1, 2, false, 4, (1, 2, 3))`.
6. Updates to all runtime tag-checking so that tuples are treated as a separate value from integers and booleans. Tuples are considered invalid operands for some operations (like `+`).
7. An update to the = operation. This operation checks equality between tuples solely by reference (no structural equality). Tuples cannot be compared to non-tuple values.

Abstract Syntax

The abstract syntax of Snek, parsed out of a .snek file and then compiled into instructions, is shown below.

```
 enum Op1 { Add1, Sub1, IsNum, IsBool, Print, }

enum Op2 { Plus, Minus, Times, Equal, Gt, Gte, Lt, Lte, }

enum Expr {
    Number(i64),
    Boolean(bool),
    Tuple(Vec<Expr>),
    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>),
    TInit(Box<Expr>, Box<Expr>),
    TSet(Box<Expr>, Box<Expr>, Box<Expr>),
    TGet(Box<Expr>, Box<Expr>),
    Block(Vec<Expr>),
    Call(String, Vec<Expr>),
}

struct Function {
    name : String,
    args : Vec<String>,
    body : Expr,
}

struct Program {
    defns : Vec<Function>,
    main : Expr,
}
```

Value Representations

Values, such as integers, booleans, etc., are represented in the Snek runtime environment with two parts: a code and a tag. The tag is on the less significant part of the byte (includes LSB). The value representations are as follows. Note that the code reflects a decimal representation of the actual binary code part.

Value	Tag Size	Code	Tag
integer	1 bit	n	0
tuple	2 bits	addr	01
true	2 bits	1	11
false	2 bits	0	11

Heap-Allocated Data Diagram

This diagram will be included as a `.jpg` on the repository.

Testing

A number of tests were prepared and run with the Egg-Eater compiler. These tests are located in the `input` directory of the repo. They are also duplicated in the `tests` directory. Some extra tests beyond the specified ones are located in the `input/extra` directory, and these are also present in the `tests` directory.

`simple_examples.snek`

```
(block
    (let ((t1 (tuple 1 2 3)) (t2 (tinit 5 false)) (t3 (tuple)))
        (block
            (print t1)
            (print t2)
            (print t3)

            (print (tset t2 0 true))
            (print (tset t2 1 1))
            (print (tget t1 1))
        )
    )

    (tinit 0 0)
)
```

This program includes a number of simple tests that implement correct tuple operations and print the results out. The first is the simple construction of tuples via `tuple` and `tinit`. These tuples are printed out after construction, also showing the result of a `print` operation on tuples.

This program also shows how the `tset` operation performs persistent updates to heap-allocated values by printing the results of the operation. The `tget` operation is also tested to lookup an element of a tuple.

Finally, the construction of zero-length tuples are shown via the two possible methods: `(tuple)` and `(tinit 0 <expr>)`. These tuples are also printed out (as `()`).

`error-tag.snek`

```
(block
    (print (= (tuple true 4 false) 4))
    (print (+ (tuple) 4))
    (print (add1 (tinit 0 0)))
)
```

This program tests 3 different runtime tag-checking errors. To test each individually, each print statement should be isolated (the others temporarily removed).

The first statement incorrectly uses the equals operation with a tuple and an integer. Since these two types have different tags, a runtime error results. The second statement incorrectly uses the `+` operation with a tuple, which is not allowed and results in a runtime error. The third statement incorrectly uses a unary operator (`add1`) on a tuple to confirm that tag-checking is consistent throughout all relevant operations. This statement too causes a runtime error.

Note: More tag-checking errors are tested in `error3.snek`. These tag-checking errors are specific to the tuple operations rather than the non-tuple operations.

`error-bounds.snek`

```
(let ((t (tuple 54 43 32 21 10)))
    (block
        (print (tset t -1 0))
        (print (tset t 5 1))
        (print (tset t 8 false))

        (print (tget t -1))
        (print (tget t 5))
        (print (tget t 8))

        (print (tinit -1 0))
    )
)
```

This program shows seven statements that each cause a runtime out-of-bounds error. The first three test out-of-bounds with respect to the `tset` operation, showing that negative indices and indices greater than or equal to the tuple length cause an out-of-bounds error.

The second three test out-of-bounds with respect to the `tget` operation, again showing that negative indices and indices greater than or equal to the tuple length cause an out-of-bounds error. This is consistent with the errors produced by `tset`.

The last statement tests a specific case where `tinit` causes a runtime out-of-bounds error when specifying a negative length, as this would not produce a valid tuple.

`error3.snek`

```
(block
    (print (tinit false 0))
    (print (tget 0 1))
    (print (tset false 1 2))
    (print (tget (tuple 1 2 3) false 0))
    (print (tset (tuple 1 2 3) (tuple 3 2 1)))
    (print (tinit -1 0))

    (tinit 0)
    (tset)
    (tget (tuple 1))
)
```

This program tests tuple operation tag-checking/type errors and parsing errors. Each statement shows an incorrect argument type to each operation. The first argument for all `tinit` must be an integer or a runtime error will occur. The first argument for `tget` and `tset` must be a tuple reference value or a runtime error will occur. Lastly, the second argument for `tget` and `tset` must be an integer or a runtime error will occur.

Three additional tests are included to show parsing errors related to these tuple operations.

`points.snek`

```
(fun (point x y)
    (tuple x y)
)

(fun (addpoints pt1 pt2)
    (tuple (+ (tget pt1 0) (tget pt2 0)) (+ (tget pt1 1) (tget pt2 1)) )
)

(let ((p1 (point 13 -51)) (p2 (point 79 24)) (p3 (point 14 -32)))
    (block
        (print p1)
        (print p2)
        (print p3)
        (print (addpoints p1 p2))
        (print (addpoints p1 p3))
        (print (addpoints p2 p3))
        (addpoints p1 (addpoints p2 p3))
    )
)
```

This program shows how abstract structures can be created via tuples. In this case, the structure created is a "point." This point is really a 2-tuple. The `point` function returns a point construct. The `addpoints` function adds two of these point constructs by adding their x and y coordinates.

These functions are tested by printing constructed points and the additions of these points together.

`bst.snek`

```
(fun (insert bst len val)
    (let ((i 0))
        (loop
            (if (>= i len)
                (break false)
                (let ((get (tget bst i)))
                    (if (isbool get)
                        (block
                            (tset bst i val)
                            (break true)
                        )
                        (if (= get val)
                            (break true)
                            (if (< val get)
                                (set! i (+ (* i 2) 1))
                                (set! i (+ (* i 2) 2))
                            )
                        )
```

```
                    )
                )
            )
        )
    )
)

(fun (lookup bst len val)
    (let ((i 0))
        (loop
            (if (>= i len)
                (break false)
                (let ((get (tget bst i)))
                    (if (isbool get)
                        (break false)
                        (if (= val get)
                            (break true)
                            (if (< val get)
                                (set! i (+ (* i 2) 1))
                                (set! i (+ (* i 2) 2))
                            )
                        )
                    )
                )
            )
        )
    )
)

(fun (newbst len)
    (tinit len false)
)

(let ((len 20) (tree (newbst len)))
    (block
        (insert tree len 10)
        (insert tree len 13)
        (insert tree len 4)
        (print tree)
        (insert tree len 6)
        (insert tree len 12)
        (insert tree len 4)
        (insert tree len -3)
        (print tree)
        (print (lookup tree len 3))
        (print (lookup tree len 10))
        (print (lookup tree len 12))
        (print (lookup tree len -3))
        tree
    )
)
```

This program tests whether a binary search tree implementation is possible within the Egg-Eater language specification. Indeed it is. Here the BST operations `new`, `insert`, and `lookup` are reproduced as Snek functions.

`newbst` initializes a new tuple with `false` values as a starting BST structure. `false` indicates an empty cell.

`insert` loops through the BST structure, traversing it based on the arithmetic relation of the value to-be-inserted to each node's value in the BST, and moving "left" or "right" accordingly. Upon the first empty cell, the value is inserted. This function returns `true` if the insert operation succeeded and `false` otherwise. Note that if the BST has no space left, `false` is returned. Note also that duplicate values (found via an equality check with each node's value) are dropped, but `true` is returned.

`lookup` loops through the BST structure similarly to `insert`. In this case however, we check each node's value for equality while traversing. If the lookup value and the node's value are equivalent, we return `true`. If the BST is searched completely according to the lookup value and an empty cell is found, then we return `false`. Here the boolean represents whether or not the lookup value exists in the BST.

This program includes several statements that build up a BST via `insert` operations and prints the tree at various points. Additionally, we have statements that test `lookup` of the tree and print the results appropriately.

Similarities to Other Languages

This Egg-Eater language specification implements tuples similarly to Golang tuples, in that Golang can construct tuples according via the same default construction (specifying each value separately). We can compare this also to Java arrays, in which this construction is not the default (even though such a construction is possible). Additionally Golang has get and set operations for tuples just like this specification. Java only provides array access with a square bracket syntax.

Credits

For implementing the BST: https://stackoverflow.com/questions/68281539/implementing-binary-search-tree-using-array-in-c

For assembly instruction reference: https://www.felixcloutier.com/x86/

Additionally, EdStem posts helped me clarify some of the requirements for my Egg-Eater specification.