Concrete Syntax

Please note: the braces, `[]`, are currently not used for anything; they just serve to show what the `*` and `+` operators are being applied to, without being confused for parentheses

```
<prog> := <defn>* <expr>

<defn> :=
  | fun <name>()::<type> <block>; // zero params
  | fun <name>(<name>::<type>[, <name>::<type>]*)::<type> <block>; // 1+ params
  | struct <name> (<name>::<type>[, <name>::<type>]*); // structs have 1+ fields

<expr> :=
  | <number>  // naturals (incl 0)
  | ~<number> // this is how we make negative numbers
  | true
  | false
  | input
  | <identifier>
  | let (<binding>[, <binding>]*) <block>
  | <op1> <expr>
  | <wrapped_expr> <op2> <wrapped_expr>
  | <name> := <expr>
  | if (<expr>) <block> else <block>
  | {<expr>[; <expr>]*} // block
  | do <block> until (<expr>)
  | <name> (<expr>*) // function call
  | null <name>
  | new <name> // alloc
  | <expr>.<name> // lookup
  | <expr>.<name> := <wrapped_expr> // update
  | <wrapped_expr>

<wrapped_expr> :=
  // don't need to be wrapped as they're a single thing
  | <number>
  | ~<number> // since the ~ is a "sticky" operator
  | true
  | false
  | input
  | <identifier>

  // already wrapped in curly braces
  | block

  // need to wrap, specifically when used in binops
  | (let (<binding> [, <binding>]*) <block>)
  | (<op1> <expr>)
  | (<wrapped_expr> <op2> <wrapped_expr>)
  | (<name> := <expr>)
  | (if (<expr>) <block> else <block>)
  | (do <block> until (<expr>))
  | (<name> (<expr>*))
  | (null <name>)
  | (new <name>)
  | (<expr>.<name>)
  | (<expr>.<name> := <wrapped_expr>)
  | (<wrapped_expr>) // when parsing, this is simply unwrapped; extra parens have no meaning

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

<type> := int | bool | <name>

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

Tests

We wrote a transpiler to exhaustively test our new syntax. The transpile.py and transpile.rs files can rewrite any .snek programs into their .bet counterparts. Updated versions of eggeater programs can be found in bet-samples. We have new versions of the bst, factorial, fibonnaci points and simple_examples success files. We also have a few failure examples, labeled as parse_*_fail.bet, to illustrate invalid syntax.

We will now walk through the new bst syntax to illustrate our improvements.

```
struct bst (val::int, left::bst, right::bst); # C-like structure definitions

fun newBST ()::bst { # C-like function syntax with type annotations
  new bst
};

fun setVal (tree::bst, newVal::int)::bst {
  tree.val := newVal; # Block syntax - everything has a semicolon except the last entry in a block
  tree
};

fun setLeft (tree::bst, newLeft::bst)::bst {
  tree.left := newLeft; # Walrus operator for assignation
  tree
};

fun setRight (tree::bst, newRight::bst)::bst {
  tree.right := newRight; # C-like field access
  tree
};

fun search (tree::bst, searchVal::int)::bst {
  if (tree == (null bst)) { # C-like conditional syntax
    null bst
  } else {
    let (
      cur:=tree.val
    ) {
      if (cur == searchVal) {
        tree
      } else {
        if (cur < searchVal) {
          search(tree.right, searchVal) # Clean function calls
        } else {
          search(tree.left, searchVal)
        }
      }
    }
  }
};

fun addTree (tree::bst, newVal::int)::bst {
  if (tree == (null bst)) {
    let (
      newTree:=newBST
    ) {
      newTree.val := newVal;
      newTree
    }
  } else {
    let (
      cur:=tree.val
    ) {
      if (cur == newVal) {
        tree
      } else {
        if (cur < newVal) {
          if (tree.right == (null bst)) { # Pointer comparison
            tree.right := newBST;
            tree.right.val := newVal; # Access composite field
            tree
```

```
          } else {
            tree.right := (addTree(tree.right, newVal));
            tree
          }
        } else {
          if (tree.left == (null bst)) {
            tree.left := newBST;
            tree.left.val := newVal;
            tree
          } else {
            tree.left := (addTree(tree.left, newVal));
            tree
          }
        }
      }
    }
  }
};

let (
  x:=(null bst)
) {
  x := (addTree(x, 4)); # Use () wrappers to force a value and rmeove parsing ambiguity
  print x;
  print x.left;
  print x.right;
  x := (addTree(x, 3));
  print x;
  print x.left;
  print x.right;
  x := (addTree(x, 2));
  print x;
  print x.left;
  print x.right;
  x := (addTree(x, 5));
  print x;
  print x.left;
  x.right
}
```