leftwrite

A lightweight, explorable, tree-based language that usually makes sense!

Syntax

Expressions

In many languages, expressions are delimited by characters such as semicolons or commas. In leftwrite, **no characters are used at the end of a normal expression**. This is the primary hinderance in becoming familiar with leftwrite, as its parser will not throw errors for unintended, uncontained, or "vague" statements as long as they follow grammar.

Unaries

A unary is the simplest statement in leftwrite, and are operands in any compound expression. They are:

- integer
  `0 1 2 3 ... 2147483647`
- string
  `"Hello, world!"`
- list, or array definition
  `{1 1 2 3 5 8 13}`
  `[1 1 2 3 5 8 13]`
  note: (1) no delimiters are used and (2) data types may be heterogeneous
- variable or object
  `foo`
  `bank.balance`
- function or lambda call
  `fibonacci{5}`
  `bank.withdraw{70 "USD"}`
  `lambda{x}{x*x}{3}`
- The **exists** operator `?` may be appended to any variable or object, which evaluates to 1 if the object exists and is not null, or 0 if it is a soft-null or nonexistent.
- Any unary may be prepended with the **logical not** operator `!` or the **unary minus** `-`.

- A compound expression enclosed within **parentheses** `( )` is evaluated as a unary.

- A *list* supports the built-in pop, queue, insert, delete, and lookup operations in linear time.
- An *array* supports only lookup in constant time. An array may be converted to a list or vice-versa with the built-in functions `toArray` and `toList`. Use the **index operator** `#` to access a value.
  `>> myArray#5`

Arithmetic Expressions and Boolean Operators

A binary expression applies an operator to two unaries, and binary expressions may be combined to follow standard mathematical precedence. When precedence of adjacent operators are equal, leftwrite evaluates expressions from left to ~~write~~ right. The level 1 operators are boolean operators and return an integer with a value of either zero or one. If at any time a binary within a compound boolean operation is false, the expression evaluates to false and the remaining operations are not completed.

The precedence levels and corresponding operators are:

level 1

- `=` logical equals
- `&` logical and
- `|` logical or
- `<` `<=` less than (or equal to)
- `>` `>=` greater than (or equal to)

level 2

- `+` plus - may also be used for string concatenation and to combine strings and integers
- `-` minus

level 3

- `*` multiply
- `/` divide - will return an integer unless an operand is a real

level 4

- `^` exponent
- `%` modulo

Variable Definition and Manipulation

Variables are defined using either the `def` or `decl` keyword. `decl` does not take a value and initializes a variable to a soft null. Values may be changed with `set` ,which may search enclosing environments to update the given variable. Variables already defined or declared may not again be defined or declared, and variables not declared may not be set.

```
>> decl foo
>> def fum " eight nine!"
>> set foo 7
>> ptln{foo + fum}
7 eight nine!
```

Logic Flow and Iteration

If no body follows the control statement, the next statement is evaluated if the condition is true. Be wary of ambiguous control statements.

`if` , `else if` , and `else` come with their standard functionality and may be nested. `else if`s may be chained. All conditions must be unary.

```
if !var?{ ~ needs a body so that the next else if is not ambiguous
   if otherVar?
       ptln{"var doesn't exist!}
}
else if (var = 3 & var + otherVar = 4){ ~must be in parenthesis to be a unary
   ptln{"our var is 3"}
   ptln{"and the other var is 4"}
}
else if (var = 4)
   swap var otherVar
else HALT ~stop the program
```

`for` with standard functionality: The four parameters respectively: declare the counter variable, initialize the counter variable to some value, evaluate a condition, and set the counter variable before the next iteration. All parameters must be unary.

```
for(i 1 (i<=1000) (i+1)){ ~doubles every loop!
   ptln{"i is " i}
   if(i = 20) done
}
```

`while` with standard functionality: evaluates a statement or body until its unary condition is false.

```
while(p.parent){
   println{p}
   p = p.parent
}
```

Keywords

A overview of keywords and their functions is listed below. All keywords are case-insensitive.

`def` binds a variable, procedure, or object to an expression
`>> def x 7`

`decl` binds a variable or object to a soft null value
`>> decl x`

`set` modifies the value of a variable or object
`>> set x 7`

`return` short-circuits the current scope and returns an expression
`>> return x`

`copy` initializes an object or variable to a recursive copy of another. The copied object has the same value(s) as the original object. The values of the original and the copy are distinct and have separate pointers in memory, except for pointers to other environments. This may eliminate the need for many "temp values" when manipulating objects.
`>> copy apples oranges`

`swap` swaps the values of two objects or variables
`>> swap myGrade yourGrade`

`done` short-circuits the scope and returns a soft null value

`HALT` immediately halts execution

`TRUE` and `FALSE` evaluate to integers 1 and 0, respectively
`NULL` evaluates to a "soft null" lexeme (see notes on soft nulls)
`OBJECT` defines structures; equivalent to a new and empty environment.
`THIS` returns the current environment

the `_include` tag will immediately execute a quoted file.
`>> _include "stuff.lx"`

Other Syntax

- Control statements and function definitions contain a *body* - a collection of statements enclosed with braces `{` `}` .

Objects and Environments

leftwrite offers support for objects and structures by manipulating environments as variables. A variable is defined as an empty environment using the keyword `object`. Using the dot operator `.`, object attributes may be assigned and retreived. An object may contain other objects.

```
>> def book OBJECT
>> def book.title "Structure and Interpretation of Computer Programs"
>> def book.altTitle "Getting Wasted and Trying to Code"
>> println{book.title " may also be referred to as " book.altTitle}
Structure and Interpretation of Computer Programs may also be referred to as
Getting Wasted and Trying to Code
```

While an object may certainly be returned from a function, they may by defined *as the function* by using the `this` keyword to return the current environment.

```
>> def bookMaker{t a}{
>>     def title t
>>     def altTitle a
>>     def print{}{
>>         println{title " is also known as " altTitle}
>>     }
>>     return this
>>  }
>>  def myBook bookMaker{"leftwrite manual" "the holy grail"}
>>  myBook.print{}
leftwrite manual is also known as the holy grail
```

Built-in Functions

List and Array Utilities

`toArray{list}` returns a transformation of a list into a static array supporting access in constant time
`toList{array}` returns a transformation of an array into a dynamic list `length{list}` returns the length of a list or array

`lGet{list index}` returns the value at the given index of a list `lPrep{list obj}` prepends an object to the front of a list
`lApp{list obj}` appends an object to the tail of a list
`lIns{list index obj}` inserts an object into a list at the given index
`lDel{list index}` removes and returns an object from a list at the given index

Unless otherwise noted, all utilities above destructively modify their parameters and return soft-null.

Dictionary Functions

leftwrite includes a built-in library for a standard AVL-tree-based dictionary. A dictionary is an **object**, and data is stored as a key/value pair. Keys may be integers, or any other data type provided that an appropriate compare function is passed to the dictionary. In the following example, `x` is set to a standard dictionary with an integer key-type. `y` is set to a dictionary that calls `comparator` to appropriate structure position.

```
_include "dictionary.lr"
def x newDictionary{NULL}
def y newDictionary{comparator}
```

Comparator functions must take exactly two arguments, `a` and `b`. The function must return a value greater than zero, if `a>b` , less than zero if `a<b`, or zero if `a=b`.

A dictionary object includes the following self-explanatory functions/attributes:

```
dictionary.store{key value}
dictionary.remove{key}
dictionary.get{key}
dictionary.printPreorder{}
dictionary.size
```

Display Functions

`ptln` is the main print function in leftwrite. It displays an arbitrary number of arguments followed by a newline.
```
ptln{"x is :" x " and y is : " y}
```

`pt` prints arguments without a newline.

Debugging Functions

`explore{object}` enters an interactive debugging mode used to explore the lexeme tree structure of any object. Commands are followed by the `ENTER` key. Please note that formatting uses console-specific escape characters that may not work on all operating systems.

- `a` takes the left branch
- `d` takes the right branch
- `w` moves up a level (will exit if attempting to zoom out beyond the object `explore` was called with)
- any other key exits the exploration

`penv{object}` prints a table of variables and values for the environment of some object

`pptr{object}` prints the location in memory for some object

`exists{object}` returns true if its argument exists. This is different from the exists *operator* `?` in that soft-null values will return true.

Notes

Soft-Nulls

Most null values encountered in leftwrite will manifest in the form of a *soft-null*. Soft-nulls, for all intents and purposes, are the same as traditional null values, except a `LEXEME_NULL` Lexeme assigned to the value in the underlying C infrastructure (instead of a null pointer). Soft-nulls are created during variable declaration or in any other instance where memory is to be reserved for some data that does not yet have a value.

Credits

leftwrite was authored by Scott Sinischo for CS403 at the University of Alabama under the direction of Dr. John Lusth.