joRdan User Specifications

Document created 22 April 2021.

1. Overview

This is an implementation of the programming langauge R in TypeScript by Hoe Chan Wei Daniel and Chin Synn Khee, Joash. Note that this implementation attempts to be as faithful as possible to the original implementation: we have taken some liberties as to our implementation due to time constraints on the project.

R is an expression based, vector-oriented, functional language, well suited for scientific programming. It's primitive types are all 1-indexed vectors, making it easy to work with large amounts of data without having to write loops.

R is also a homoiconic language where function calls and entire programs are R objects which can be manipulated. This gives rise to powerful metaprogramming capabilities which allow library authors to write powerful Domain Specific Languages making working with data very convenient. Examples of such DSLs include the entire Tidyverse suite of libraries authored by Hadley Wickham, including the dplyr library for working with data frames and ggplot2 for creating plots.

---

2. Grammar / Syntax

A program in our R interpretation, defined Backus-Naur Form, is as follows:

$$
\begin{array}{lll}
\textit{program} & ::= & (\textit{expression}\ \texttt{;})\ldots \\
\textit{expression} & ::= & \textit{assignment} \\
 & | & \texttt{function}\ (\ \textit{formals}\ )\ \ \textit{block} \\
 & | & \texttt{if}\ (\ \ \textit{expression}\ \ )\ \ \textit{expression}\ (\ \texttt{else}\ \ \textit{express} \\
 & | & \texttt{while}\ (\ \ \textit{expression}\ )\ \textit{expression} \\
 & | & \texttt{for}\ (\ \ \textit{symbol}\ \texttt{in}\ \textit{expression}\ )\ \textit{expression} \\
 & | & \texttt{repeat}\ \ \textit{expression} \\
 & | & \texttt{break} \\
 & | & \texttt{next} \\
 & | & \texttt{\{}\ \ (\textit{expression}\ \texttt{;})\ldots\ \ \texttt{\}} \\
 & | & \textit{literal} \\
 & | & \textit{symbol} \\
 & | & (\ \ \textit{expression}\ \ ) \\
 & | & \textit{expression}\ \ (\ \ \textit{args}\ \ ) \\
 & | & \textit{unary}-\textit{operator}\ \ \ \textit{expression} \\
 & | & \textit{expression}\ \ \textit{binary}-\textit{operator}\ \ \ \textit{expression} \\
 & | & \textit{expression}\texttt{[}\textit{expression}\texttt{]} \\
 & | & \textit{expression}\texttt{[[}\textit{expression}\texttt{]]} \\
 & | & \textit{expression}\texttt{\$}\textit{name} \\
 & | & \textit{name}\texttt{::}\textit{name} \\
\textit{name} & ::= & \textit{symbol}\ |\ \textit{string} \\
\textit{formals} & ::= & \ \in\ |\ \textit{formal}\ (\ ,\ \textit{formal}\ )\ \ldots \\
\textit{formal} & ::= & \textit{symbol} \\
 & | & \textit{symbol}\ \ \texttt{=}\ \ \textit{expression} \\
\textit{args} & ::= & \ \in\ |\ \textit{arg}\ (,\ \textit{arg}\ )\ \ldots \\
\textit{arg} & ::= & \textit{expression} \\
 & | & (\textit{name}|\texttt{NULL})\ \ \texttt{=}\ \ \textit{expression}? \\
\textit{assignment} & ::= & \textit{name}\ \ \texttt{<-}\ \ \textit{expression} \\
 & | & \textit{name}\ \ \texttt{<<-}\ \ \textit{expression} \\
 & | & \textit{expression}\ \ \texttt{->}\ \ \textit{name} \\
 & | & \textit{expression}\ \ \texttt{->>}\ \ \textit{name} \\
\textit{unary}-\textit{operator} & ::= & \texttt{+}\ |\ \texttt{-}\ |\ \texttt{!} \\
\textit{binary}-\textit{operator} & ::= & \texttt{+}\ |\ \texttt{-}\ |\ \texttt{*}\ |\ \texttt{/}\ |\ \hat{}\ |\ \texttt{\%\%}\ |\ \ \texttt{\%/\%} \\
 & | & \texttt{<}\ |\ \texttt{>}\ |\ \texttt{<=}\ |\ \texttt{>=}\ |\ \texttt{==}\ |\ \texttt{!=} \\
 & | & \texttt{\&}\ |\ \texttt{\&\&}\ |\ \texttt{|}\ |\ \texttt{||}\ |\ \texttt{:}\ |\textit{user}-\textit{op} \\
\textit{user}-\textit{op} & ::= & \texttt{\%}\ \ \textit{character}\ldots\ \ \texttt{\%} \\
\textit{literal} & ::= & \textit{number} \\
 & | & \textit{number}\ \texttt{L} \\
 & | & \textit{string} \\
 & | & \texttt{NA} \\
 & | & \texttt{Inf}\ |\ \texttt{NaN} \\
 & | & \texttt{NULL} \\
 & | & \texttt{T}\ |\ \texttt{TRUE} \\
 & | & \texttt{F}\ |\ \texttt{FALSE}
\end{array}
$$

Syntax things to take note

- Return functions are only allowed in the bodies of functions

- Break and next functions are only allowed in loop bodies
- Our parser technically requires semicolons to terminate expressions, though single expressions entered in the REPL will be able to be parsed correctly with only an error logged in the console.
- Strings accept unicode escape sequences, however, since we use Javascript's underlying JSON parsing facilities joRdan strings must be surrounded by double quotes "".
- Various number formats are supported, including hexadecimal (0xAF) and scientific notation (1E-10)

3. Objects

3.1. Basic types

3.1.1. Vectors

- Vectors can be thought of as variable length arrays that hold data.
  - Vectors are 1-indexed.
  - Each atomic value in the array can be accessed using operations such as `x[5]` (more on this in 4.4. Indexing).
- joRdan has four basic ('atomic') vector types: *logical, integer, double, character.*
  - Variation from actual R — we have excluded *complex* and *raw* data types from our implementation as they are rarely used.
  - The atomic types follow a type hierarchy of *character* > *double*> *integer* > *logical*.
    - i.e. the logical `TRUE` widens to 1L (integer), which can widen to 1 (double) which lastly widens to "1" (character).
- Single numbers (e.g. `4`) and simple strings (e.g. `Hello world!`) are still considered vectors of length 1.
  - Note: Vectors of length 0 are possible.
- Vectors are most commonly constructed via the `c` primitive function:
  - Combines vectors of the same type (widening if needed so that all arguments have the same type)
  - E.g. `c(1,2,3)` produces a numeric vector of length 3, containing 1, 2 and 3.
- String vectors are character vectors:
  - A single element of a character vector is referred to as a *character string*.
- All atomic vector types can contain missing data, denoted as `NA`.
  - E.g. `c(1, NA, 3)` is a double vector of length 3, with a missing 2nd value.

3.1.2. Lists

- Lists are another way to store data in joRdan.
- Lists have elements, each of which can contain any type of joRdan object.
  - Lists can be thought of as a generic vector holding arbitrary joRdan objects instead of elements of the same type.
- List elements can be accessed using three different indexing operations (`[`, `[[`, `$`, further elaborated in 4.4. Indexing).

3.1.3. Language & symbol objects

- Symbols are bindings to data in environments.
  - E.g. if we do `x <- 4` then x is a symbol in the global environment bound to the double vector 4.
  - Symbols evaluate to the value they are bound to and thus are not commonly seen when programming in joRdan, but can be exposed and manipulated via metaprogramming (see `quote`)
- joRdan's rules for valid symbols are:

    - Symbol names consist of alphanumeric characters, the period `.` and underscore `_`, and begins with either a letter or a period followed by not a number. E.g. `hello123` `.dot_dot` are valid, but `.12hello` and `hello world` is not.
    - Invalid names can also be used as names by wrapping them in backticks. Thus `` `.12hello` `` and `` `hello world` `` can be used as variables
- Language objects are function calls, that evaluate to the result of applying the given function to the supplied arguments.
    - These are also not commonly seen but can be expressed and manipulated via `quote` as well.

3.1.4. Expression objects

- Expression objects represent joRdan programs. They are the result of `parse` and are simply lists of joRdan values that can be evaluated in sequence via `eval`.

3.1.5. Function objects and Promises

- All language constructs in joRdan are in fact implemented as functions - including assignment (`<-`), conditions (`if` statements), loops and even blocks (`{`)!
- joRdan has 3 function types - specials, builtins and closures.

3.1.5.1. Closures

- Closures are functions defined in joRdan itself - base library and user-defined functions.
- Closures comprise of formals, the function body and its enclosing environment.
- Closures are first class values in joRdan , meaning they can be passed as arguments, returned from closures and even mutated dynamically.
- Closures are *lazy*, meaning that they do not evaluate their arguments unless necessary.
    - This is achieved by wrapping each argument in a promise object (i.e. a thunk in joRdan terminology) which is evaluated as necessary.

3.1.5.2. Builtins

- Builtins are javascript functions that are not lazy - they evaluate all arguments before applying.

3.1.5.3. Specials

- Specials are JavaScript functions that also do not evaluate their arguments before application.
- They are used to implement most language constructs that have evaluation semantics that require not evaluating all arguments beforehand, such as conditions and loops.
- Builtins and Specials can be Primitive or Internal.
    - Primitive functions can be called directly, making them more efficient.
    - Internal functions can only be called via the `.Internal()` primitive function.
    - Internal function calls are usually wrapped up in regular closures, which allows default arguments to be supplied or type coercions to be performed before passing on to the internal function.

3.1.6. NULL

- joRdan consists of a special type `NULL`.
- It can used to represent the absence of an object; however, do not confuse it with a vector of length 0.
- The `NULL` object has no type and no modifiable properties.
- joRdan only has one `NULL` object, to which all instances refer.
- Attributes cannot be set on `NULL`.

3.1.7. Dotdotdot

- joRdan has a special object, '...' that appears in parameter lists of function definitions and within the function body.
- If the function's formals includes the dotdotdot, when the function is called, all supplied arguments that do not match the function formals precisely (positionally or by exact naming) will be collected into the dotdotdot object,

which is like a list, and can be passed to other functions called within the function body.

- This allows, for example, a function `f` to call another function `g` that has a lot of configuration parameters, while allowing the user to specify these parameters and simply pass them along without cluttering the interface of `f`.

- If a dotdotdot object exists in scope (e.g. in the body of a function with it as a formal), then `..1`, `..2`, etc. are special symbols that refer to the corresponding values within the dotdotdot object positionally.

3.1.8. Environments

- Environments contain bindings of symbols to joRdan objects.
  - Unlike other joRdan objects, environments do not have copy-on-modify semantics (more in 4. Evaluation semantics).
  - This means environments are mutated in place when assignments occur.
- All environments have a parent environment, thus forming a tree structure called the search path.
- In the default joRdan shell or in script execution, the execution environment is the global environment.
  - Any attached library's namespace will become the parent of the global environment and the child of whatever preceding library was attached, finally ending at the base environment where core language objects are defined (such as control flow and operators), and the parent of the base environment is the empty environment, whose parent is `NULL`.

The search path in joRdan is a simplified version of that in R. In R, each attached package has 2 environments - a package environment (the one which is attached to the search path) and a namespace environment, and functions in the package can be found in both environments. This setup is to allow packages to control what their own dependencies are without being subject to the current search path. (Imagine if a package redefined some base function - all subsequent packages on the path would be broken!), and allows packages to hide their internal private functions and only expose certain functions on the search path. To support these additional environments, the environment hierarchy is greatly complicated.

In joRdan there is only a single search path, and function lookup only takes place on this single search path.

3.1.9. Pairlist objects

- Pairlists are similar to linked lists: they have a `next` pointer that points to the next item in the list.
- Used extensively in the internals of joRdan (e.g. passing arguments) but rarely seen at language level as they are much less efficient than lists.
- Elements can be accessed using the same `[[]]` or `$` syntax as lists.

3.2. Attributes

- All objects except `NULL` can have arbitrary metadata attached to them.
- Attributes are stored as a pairlist, where all elements are named and should be thought of as key : value pairs.
- A listing of the attributes of an object can be obtained using `attributes(x)`, and can be set using `attributes(x)<-`.
- Individual attributes can be accessed using `attr` and `attr<-`.
- Some attributes have special accessor functions that should be used whenever available (e.g. the `names` and `names<-` functions to get and set names on vectors and lists)
- Some attributes are specially considered (e.g. `dim`), and R will intercept calls that are relevant to these special attributes to enforce certain checks.
- Attributes are used to implement joRdan's generic function dispatch system, via the `class` attribute which can be used to add classes to R objects, which then have different behaviors when generic functions are called on them.

3.2.1. Names

- The `names` attribute, when present, labels the individual elements of a vector or a list.

- When an object is printed, if the `names` attribute is available, it will be used to label the elements of the object.
- Names are often directly specified when creating vectors or lists.
  - For example, `list(count = 1, isBig = TRUE)` creates a list of length 2, with 1st element having name `count` and 2nd element having name `isBig`.
  - Also, `c(a=1, b=2, c=3, 4)` creates a double vector of length 4 with 1st 3 values having names "a", "b", "c". The name of the last value is set to "", which is the default name given to unnamed values.
- The `names` attribute can be used for indexing purposes.
  - For example in the earlier list `x <- list(count = 1, isBig = TRUE)` we can retrieve the first element not just positionally via `x[[1]]` but also by name, either by `x[["a"]]` or by `x$a`.
  - This makes lists function just like objects in other languages like JavaScript.
- The `names` and `names<-` constructions can be used to get and set names; setting names will perform the necessary consistency checks.
- Pairlists are treated specially —
  - For pairlist objects, a virtual `names` attribute is used; it is actually constructed from each key of the pairlist's components.

3.2.2. Classes

- The `class` attribute must be a character vector, and it denotes the classes an object has, which determines function dispatch for generic functions (See section 4.6.3 - Object Oriented Programming)
- If an object has a defined `class` attribute, those classes are taken to be all the classes the object has (implicit classes are ignored).
- If an object has no explicit classes set, each object has at least one implicit class which does not depend on the `class` attribute, depending on its type:
  - Builtins, Specials, Closures have class "function"
  - Language objects have class "call"
  - Double and Integer vectors have an additional common class "numeric" appended to the class vector, in addition to their own classes "double" and "integer"
  - Other types of objects have their class names equal to the name of the type
  - For atomic vectors, if the `dim` attribute is set, an additional class "matrix" is prepended to the class list if `dim` is of length 2, else the class is "array" (general number of dimensions)

joRdan's behavior around classes (explained later in 4.6.3. Object Oriented Programming) differs from R's S3 class system. Specifically, R's functions that work together to provide OOP functionality - `inherits`, `class`, etc. give conflicting and confusing output regarding what classes an object has, mostly due to the way implicit classes are generated.

Thus in joRdan we have decided to provide a cleaner and more consistent interface rather than be fully compatible with R. `class` lists all classes of an object as per the rules above. `inherits` likewise returns `TRUE` for all classes in the output of `class` and false otherwise. Generic function dispatch also follows the exact same rules.

3.2.3. Copying of attributes

- Scalar functions should preserve attributes
  - Scalar functions are ones that operate element-by-element on a vector
  - Their output is similar to the input
- Binary operations normally copy most attributes from the longer argument
  - If both arguments have the same length, the preference is to take attributes from the longer argument
  - Note that `names`, `dims` and `dimnames` are set depending on the operator

- Subsetting will drop all attributes except `names`, `dim` and `dimnames` which are reset to appropriate values
  - Subassignment will preserve attributes
  - Coercion drops all attributes

---

4. Evaluation semantics

4.1. Simple evaluation

4.1.1. Constants

- All constant values evaluate to themselves
- Logical, integer, numeric, character vectors are constants
- Lists, pairlists, NULL, environments and expression objects are also constants

4.1.2. Name lookup

- Symbols are evaluated to the value they are bound to in the environment.
- If the symbol is not found in the current environment, the parent environments are searched recursively until the base environment is reached, after which an error is thrown if the symbol is not found.

4.2. Control structures

- if-else
  - Evaluates the condition, and if the condition is `TRUE`, evaluates the consequent expression and returns it. Else it does not evaluate the consequent and returns `NULL`
  - If the else clause is present, it will evaluate the else clause instead if condition is `FALSE` and return it
  - The condition should be a logical vector of length 1, else a warning will be issued that only the first element is used, if the first element is `NA` an error results.
  - The condition need not be a logical vector; if it is an atomic vector of another type (integer/numeric/character) it will be attempted to be coerced into a logical.
    - Characters 'T', 'True', 'TRUE', 'true' evaluate to `TRUE`, while 'F', 'False', 'FALSE', 'false' evaluate to `FALSE`.
    - For numbers, zero is `FALSE`, other numbers are true except NaN which is `NA`.
- repeat
  - Infinitely evaluates body, unless break or return (in a function) is called. Returns `NULL` if it terminates.
- while
  - Evaluates the condition in the same fashion as if-else. If the condition is true it evaluates the body, then repeats. Also returns `NULL`.
  - Can terminate early via return or break in the body.
  - `next` skips to the next iteration if called within the body.
- for
  - Structure of a for loop: `for (symbol in expression) body`
  - `expression` can only be one of the vector or list types: expression, list, pairlist, logical, integer, numeric or character
  - Repeatedly evaluates body for *x* times, where *x* is the length of the vector or list expression.
  - At the *i*-th iteration, symbol is bound to the *i*-th value of the expression in the current environment.
  - After the loop terminates the binding of symbol still persists to the last bound value.
  - Loop can terminate early via `return` or `break`, and iterations can be skipped via `next`.

- braces { }
  - Braces consist of any number of expressions evaluated in order, and the return value of the entire block is the value of the last expression.
  - An empty brace (`{}`) returns `NULL`
  - If the body of a function contains a return expression it will return the argument of the return call without evaluating the rest of the block.
  - If the body of a loop contains a break or next call it breaks from the loop or continues to the next iteration respectively without evaluating the rest of the block.

4.3. Elementary arithmetic operations

This section will go into detail in the computation of basic operations, such as the addition or multiplication of two vectors.

4.3.1. Recycling and its rules

- Recycling happens when a binary arithmetic operation is provided with two vectors of unequal lengths.
  - The shorter vector will be recycling to the length of the longer one.
    - If the shorter vector has a length that is not a factor of the length of the longer vector, a warning is issued.
  - For example, when you add `c(1, 2, 3)`, a *real* vector of length 3, to `c(1:5)`, a *real* vector with numbers 1 to 10, the shorter vector is recycled to the same length: `c(1, 2, 3, 1, 2)`

4.3.2. Coercion

- When two arguments of different types are provided, the arguments will be coerced following a hierarchy of types
  - The type hierarchy is *real* > *integer* > *logical*.
  - For example, if vector x is type logical and vector y is type real, vector *x* will be coerced to type logical.
- Both arguments will be coerced to the same type before the arithmetic operation is carried out on them.
  - logical types — `TRUE` is coerced to 1, `FALSE` is coerced to 0.
- Coercion does not preserve all attributes — only `names`, `dims` and `dimnames` from the longer vector are copied

4.3.2. Propagation of names

- Names are taken from the longer argument (as are attributes) if it exists.
- Recycling will cause the shorter vector to lose its names.

4.3.3. NA handling

- Missing values (values whose value is not known) have the value `NA`
- All atomic vector types can take be `NA`, though in R `NA` for different types uses different representations, in joRdan we uniformly represent them with TypeScript's null
- Numerical and logical calculations that involve `NA` will generally return `NA`
  - There are exceptions to the case: `FALSE & NA` returns `FALSE`, `TRUE | NA` returns `TRUE`

4.4. Indexing

- joRdan, like R, allows access to individual elements or subsets through indexing operations.
- joRdan provides indexing of vectors and lists.
- In joRdan, indices begin at 1.
- Indexing can be used to extract parts of an object, or to replace parts of an object.
- Indexing operators include:
  - `x[i]` —
    - Multiple elements of a vector / list can be accessed through vectors.

        - A subset of the selected items of the vector or list is returned.
    - `x[[i]]` —
        - Single element access of vectors / lists using integer or character indexes.
    - `x$a`, `x$"a"` —
        - Access to lists and pairlists via a literal character string or symbol.
- Special considerations:
    - For `[[` and `$` subsetting (note that `$` only applies to lists),
        - Partial matching is only used during extraction and not replacement.
        - `x$aa` will match `x$aabb` if *x* does not contain some component named `aa` and `aabb` is the only name which has prefix `aa`.
        - This behaviour can be controlled using the `exact` argument, which defaults to `FALSE` indicating that partial matching will be used
            - Setting `exact` to `TRUE` means that exact matching is used

4.4.1. Indexing by vectors

- joRdan, similar to R, allows the access of multiple elements by using vectors as indices.
- For atomic vectors, we are only able to use `[` and `[[` for indexing.
- Assuming we have an expression `x[i]`, indexing behaviour depends on the type of *i*,
    - *integer* type (and other numeric types) —
        - If all elements of *i* are positive, then the elements of *x* with those index numbers are selected (1-based indexing).
        - If there are elements that are negative, then all elements except those indicated are selected.
        - If *i* is positive and exceeds `length(x)`, then the corresponding selections are `NA`.
        - If the *i* is a numerical type but not an *integer*, it is coerced to an *integer* object.
        - joRdan differs from R when an index of zero is used — in R, a numeric object with no values is returned while in joRdan, it simply returns the entire object.
    - *logical* type —
        - The selected values from *x* are those for which *i* is `TRUE`.
        - If *i* is shorter than *x*, *i* is recycled (as discussed in 4.3. Elementary arithmetic operations).
        - If *i* is longer than *x*, *i* is extended with `NA`s.
    - *character* type —
        - The strings in *i* are matched against the `names` attribute of *x*, producing the indexes which are used
    - If *i* is not supplied, joRdan simply returns the entire object.
- Indexing with a missing value `NA` will give `NA` result.
    - This rule also applies to logical indexing — elements of *x* that have an `NA` selector in *i* get included in the result, but their value is `NA`.

4.4.2. Indexing of lists

- Indexing of lists follow the same rules as subsetting of atomic vectors.
    - Using `[` will always return a list as the result.
    - Using `[[` and `$` allows you to pull individual elements out of the list.

4.4.3. Subset assignment

- Subassignment — using the aforementioned indexing rules, we can modify the selected values of vectors / lists.
    - Provided values will be assigned to the indexes of the selected vectors.

    - Names from the original object provided will be preserved.
    - Recycling — if the length of the selected indexes are not equal to the length of the input value, joRdan will do recycling on the values if necessary.
- Difference from R — In R, subset assignment of `NULL` to a selected index using `[[` operator will remove the component; however, in joRdan it will assign a literal `NULL` to the value.

4.5. Assignment

4.5.1. Local assignment

- Assignment in local environment is performed with the left or right assignment operators, e.g. `x <- 4` or `4 -> x` respectively.
- joRdan has immutable data, and assignment merely changes the bindings of the symbols in the current environment.
- If the symbol being assigned has an existing binding in the current environment (default being the Global Environment at the REPL) assignment changes the binding, else a new binding is created.
- A character literal can be used in place of a symbol in the assignment position, in which case the string is coerced to a symbol.
    - E.g. `"x" <- 4` works the same as `x <- 4`.
    - This lets users create bindings to invalid names, e.g. `"hello world" <- c(1,2,3,4,5)` as an alternative to using backticks ``` `hello world` <- c(1,2,3,4,5) ```
- Since assignment creates new bindings if they do not exist, assignment cannot rebind symbols in parent environments
    - e.g. function bodies cannot change bindings in the global environment; instead the non-local "superassignment" operators `<<-` and `->>` are used instead.
    - These functions start searching from the parent environment upwards for the symbol to rebind, and if the symbol is not found even in the base environment will create a binding in the global environment.
- All assignment operators return the value of the expression assigned invisibly.
- Besides regular assignment to symbols/characters, joRdan supports "replacement" and "subset" assignments.
    - These are assignments of the form `names(x) <- "hello"` or `x[[4]] <- 2` where the target of assignment is a language object (function call).
    - joRdan transforms such assignments into the form ``x <- `names<-`(x, "hello")`` and ``x <- `[[<-`(x, 4, 2)`` where "names←" and "[[←" are actual functions that usually return their first argument, but with some modifications.
    - For example "names←" returns a value equivalent to its first argument, with it's names attribute set to the character vector supplied as the 2nd argument, and ``` `[[<-`(x, i, value) ``` returns a value equivalent to its first argument x, with the element at position i changed to value.
- Since such replacement assignments return a copy of the object with modified parts it maintains the immutable data semantics of joRdan, meaning multiple values being bound to the same object will not have their value modified even if one of them has a subset assignment performed. For example:

```
x <- c(1,2,3);  # x bound to a vector
y <- x;         # y == x == c(1,2,3)
x[1] <- 10;     # x is now c(10,2,3)
y;              # y is still c(1,2,3)
```

- joRdan can perform this transformation recursively, allowing the user to perform reassignment of a small section of an object. For example `names(x)[2] <- "c"` sets the 2nd name of the vector x to "c", instead of having to set the entire names vector.
- Users can define their own replacement functions easily as well. For example:

```
`increment.by<-` <- function(x, value) {
    return(x + value);
```

```
};
a <- c(1,10,100);
increment.by(a) <- 5;
a;                       # a is now c(6, 15, 105)
```

4.5.2. Non-local assignment

- Assigning outside the current environment is done via the `<<-` operator.
- `<<-` starts searching from the parent of the current environment for the symbol being assigned to, until the base environment or empty environment
- If it finds the symbol, the reassignment is performed in that environment
- If the symbol is not found anywhere on the search path, a new binding to this symbol will be created in the global environment

> Why does joRdan (and R) require 2 separate assignments while other lexically scoped languages like JavaScript handle non-local assignment with just one? Since joRdan does not have variable declarations (e.g. `const` or `let` in JavaScript), there is no distinguishing between defining a binding and rebinding a symbol. Python similarly does not use declarations and thus has to support a `nonlocal` keyword to indicate assignment outside the current scope as well.

4.6. Function Application

All functions can be applied via the format: `func( args ... )` where `func` evaluates to the function - which is generally either a valid name that is bound to the function, or an expression that evaluates to the function. This commonly occurs in 2 ways:

- Higher order functions returning functions:

```
make_adder <- function(amount) {
  function(x) {
     x + amount;
  };
};

# make_adder(amount=2) returns a function taking x and returning the sum
make_adder(amount=2)(x=4); # evaluates to 6
```

- Creating a function and calling it immediately to prevent polluting the global environment with local variables

```
# This anonymous function is created and then called immediately
(function() {
    a <- 1;
    b <- 2;
    a + b;
})();
# Evaluates to 3
```

- Since all functions can be called in this manner, this includes infix functions and functions with reserved names, such as `if`, `for` and `{`. However note - that all these functions do not have valid names and thus must be surrounded with backticks '``' as per this interactive demo:

```
# Calling `+` prefix style, equivalent to c(1,2,3) + c(2,3,4)
> `+`( c(1,2,3),   c(2,3,4)  );
[1] 3   5   7

# Calling user defined operator %infix% these 2 ways is equivalent:
> "a"  %infix%  "b";
> `%infix%`("a", "b");

# Running a for loop, function style and regular style
> `for`(i, 1:10, `{`(i));
> for (i in 1:10) { i; };
```

4.6.1. Builtin, Special, Primitive, Internal

Primitives and Internals are functions defined in Typescript, the interpreting language for joRdan. These functions are applied directly on their argument lists, hence positional argument matching is used. For example, in the `if` primitive, the first argument must be the condition and the second argument is the consequent expression, with the optional 3rd argument being the alternative. The difference between them is primitives are called directly from joRdan, whereas internal functions can only be accessed by being wrapped in a call to the `.Internal` primitive function.

For example, `c` the vector constructing function, is a primitive, thus evaluating `c(1,2,3)` directly calls the underlying Typescript function `c` is bound to. However the string repeating function `strrep` is an internal, and thus the Typescript function can be invoked by calling `.Internal(strrep("hi", 3))` whereas simply calling `strrep("hi", 3)` will search for a regular closure or primitive with that name.

Most internal functions are wrapped in closures defined in the base library with the same name, so the user rarely needs to call `.Internal` themselves. Internals are made this way so that they are exposed to users as regular closures, allowing the usual named argument matching, default arguments or conditional checks and type coercions to be performed before calling the internal function. For example, this is the definition of `strrep` in the base library:

```
strrep <-
function(x, times)
{
    if(!is.character(x)) x <- as.character(x);
    .Internal(strrep(x, as.integer(times)));
};
```

These primitive and internal functions can be either builtin or special. Builtins evaluate their arguments before applying the function. Common arithmetic, math, type checking or constructor functions are all builtins. Special functions do not evaluate their arguments and thus are used to implement primitive functions with non-standard evaluation strategies, such as conditions (in `if`, the consequent is only evaluated if the condition evaluates to TRUE, otherwise the alternative is evaluated if it exists), or short-circuiting boolean operators `&&` and `||`.

4.6.2. Closures

Closures are joRdan-defined functions created via the `function` primitive.

4.6.2.1. Function Definition

- Functions are defined with this syntax: `function ( formals... ) body` where formals can be zero or more formal arguments
- Formals are either of the form `<name>` or `<name>=<default value>` in which case if the argument is not supplied when calling the function, the default value is used
- The body is a single expression to which the function evaluates to. To have multiple expressions, use the `{` function to group multiple expressions together
- When a function is defined it captures the environment in which it was defined, even the environment of a function call. Thus higher order functions returning functions are possible
- Closures are regular objects - they must be bound to a name in some environment, e.g. `fun <- function(x) {...}`
- Each function definition can contain one `...` formal argument - which collects unmatched arguments and can be passed as arguments to other function calls within the function bodies.

  ```
  fun <- function(...) {
      list(...);
  };

  fun(a=1, b=T);
  # evaluates to list(a=1, b=T)
  ```

- In addition to passing the entire `...` argument to another function, the individual elements can be referred to as individual variables via `..1`, `..2`, etc.

4.6.2.2. Delayed Evaluation of Arguments

- When a function is called, its arguments are not evaluated immediately, rather they are wrapped in promise objects which are only evaluated if the argument is used
- Promises save their results after being evaluated once, thus they are not evaluated multiple times

4.6.2.3. Argument Matching

- Arguments can be specified positionally, or by name. `fun(1)` or `fun(x=1)`
- When a closure is called, after arguments are wrapped into promises, the supplied arguments are matched with the function formals in a 3-phase process:
  1. Named arguments are matched to formals with exactly the same name
  2. Named arguments are matched partially to formals. An argument matches a formal partially if the name is a prefix of the formal argument name. However if there is a `...` formal argument, all arguments after the `...` formal argument must match exactly (to prevent ambiguous cases where the argument intended to be passed to another function partially matches another argument.
     - For example, a function defined as `fun <- function(x, output) { body }` can be called as `fun(1, out=2)` and `out` will match partially to `output`

     Since such partial matches may be unexpected at times, it is possible to issue warnings when this happens (see chapter 6 for evaluation options)
  3. Remaining arguments are matched positionally, and if a `...` formal is present, all subsequent supplied arguments are collected into the `...` object.
  4. If any unused arguments remain, stop evaluation and report an error

4.6.2.4. Function Evaluation

- If a formal remains unmatched to any supplied argument after the above process, it takes the default value if present, else it has the value `R_MissingArg` which will throw an error if that an attempt is made to evaluate that argument.
- A new function evaluation environment is created containing these function arguments and the body is evaluated in this environment

4.6.3. Object Oriented Programming

- joRdan has a very loose form of object-oriented programming which is enabled by a single primitive function `UseMethod`
- `UseMethod` creates generic functions that can dispatch to multiple possible functions depending on the class of the first argument (the dispatching object)
- Each object has a list of classes (See 3.2.2. Classes) which determines what methods the generic function will dispatch to
- Defining a method on a generic function is as simple as defining a function with the same name as the generic function + `".<classname>"`
  - Generic function `print <- function(x) UseMethod("print");` Define a method `print.Date <- function(x) {...}` which will get called on objects with "Date" in their class list
- The class list is order-sensitive - methods are looked up in order of the classes declared. Since the implicit classes are always behind any explicitly added classes via `class<-` implicit class methods are always the fallback methods (if they exist)
- If no method has been defined on any of the classes of an object, an error is thrown
- In addition to defining methods on specific classes, each generic function can have a `.default` method, e.g. `print.default`. This default method is called if no method has been found for any of the classes of the object

A simple example program demonstrating the use of this OO method dispatch:

```
> shout <- function(x, ...) UseMethod("shout");
> shout.default <- function(x) { "AAH"; };
> shout.integer <- function(x, str="A") { strrep(str, x); };
> shout.character <- function(x) { toupper(x); };
> shout("hello"); # dispatches to shout.character - capitalizes the string
[1] "HELLO"
> shout(4);       # 4 is a double - dispatches to shout.default
[1] "AAH"
> shout(4L);      # 4L is an integer - dispatches to shout.integer
[1] "AAAA"
> shout(4L, str="HA"); # additional arguments can be passed to the dispatched method
[1] "HAHAHAHA"
```

4.6.4. Printing Output

- joRdan is an expression-based language, everything evaluates to a value as even assignment and loops are function calls
- However not every expression produces useful output for users (e.g. `for` loops always evaluate to `NULL` but a user is rarely if ever interested in this)
- joRdan functions can choose whether they want to display their output after being evaluated by setting the global flag `R_Visible`
- All primitive and internal functions have a defined visibility - either turning `R_Visible` on, turning it off or turning it on, but letting subsequent sub-evaluations turn it on or off
  - `+` the addition primitive, sets `R_Visible` to on, thus displaying the result of addition at the REPL
  - `<-` the assignment function sets `R_Visible` off, thus assignments do not display their result (equal to the assigned value)
  - `if` the conditional expression sets `R_Visible` to on-but-mutable, thus whether the result is displayed depends.
    - If the condition evaluated to a branch that evaluates to a value, leave `R_Visible` on
    - If the condition evaluates to false and there is no alternate branch, set `R_Visible` off (so `NULL` is returned invisibly)

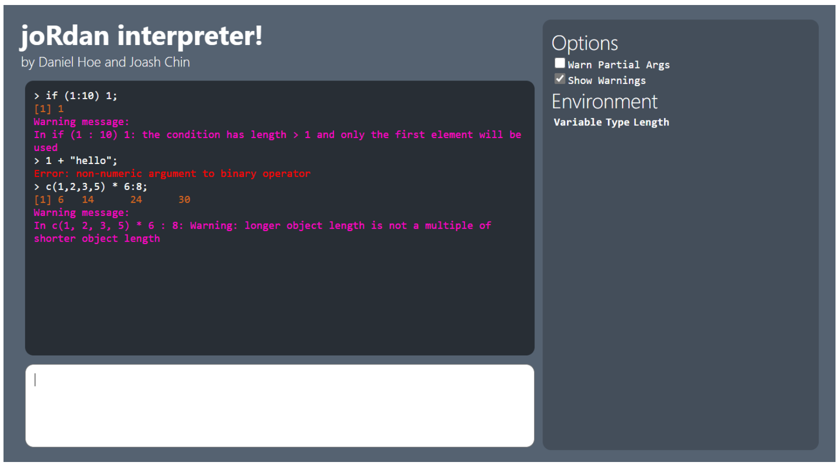
    Example:

    ```
    > if (T) 1; # expression evaluates to a useful value, print it!
    [1] 1
    > if (F) 1; # expression evaluates to the missing branch, thus do not print anything (even though NULL is actually returned i
    > if (F) 1 else 2; # expression also evaluates to a meaningful value here
    [1] 2
    ```
- joRdan-defined closures can also return output invisibly by returning their value passed to the `invisible` primitive. This primitive is simply the identity function but also sets `R_Visible` off.
- In addition to evaluation output, joRdan also produces error output which is always printed, and warnings which can be toggled. See 5. Interacting with joRdan.

5. Interacting with joRdan

To run your own instance of the joRdan interpreter, clone the repository, then install dependencies via `npm install`, then run the server via `npm run serve`. Parcel (the TypeScript/JavaScript bundler) then bundles the app and deploys it on `localhost:1234`.

This is what the joRdan interpreter on the browser looks like:



```
> if (1:10) 1;
[1] 1
Warning message:
In if (1 : 10) 1: the condition has length > 1 and only the first element will be
used
> 1 + "hello";
Error: non-numeric argument to binary operator
> c(1,2,3,5) * 6:8;
[1] 6   14      24      30
Warning message:
In c(1, 2, 3, 5) * 6 : 8: Warning: longer object length is not a multiple of
shorter object length
```

- [ ] Warn Partial Args
- [✓] Show Warnings

Variable Type Length

5.1. Interface controls

The white box at bottom-left is the command box, where the user inputs their joRdan expression to be evaluated.

- *Shift-Enter* creates a new line
- *Enter* sends the text to joRdan for evaluation
- Pressing *Up*/*Down* cycles through one's session input history

5.2. Output Formatting

The section directly above is where joRdan output is displayed. It is formatted as follows:

- User input is logged, prefixed with a ">", and in white
- Evaluation results (if R_Visible is set to true at the end of evaluation) are orange and are printed as per the generic print function
- Errors are denoted in red with the error messages copying R's error messages as closely as possible. Some error messages also report the function call from which they were triggered
- Any warnings will be logged after either the error or evaluation result is printed in purple. There can be any number of warnings triggered within one evaluation and they are all printed.

5.3. Functionalities

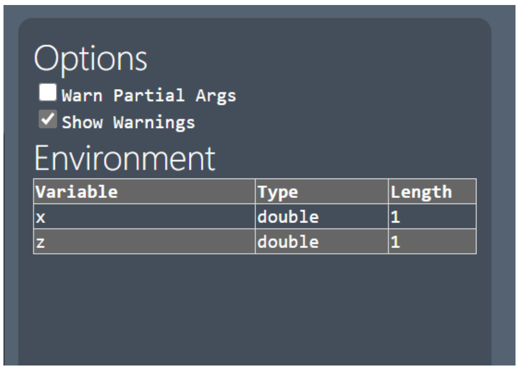
5.3.1. .Last.value

`.Last.value` is a special variable in the base environment set at the end of each evaluation to the evaluation result. This is convenient, for example, if a long computation was just performed and the user wishes to save the result of what was evaluated without reevaluating it.

5.3.2. Environment panel

The Environment panel shows currently defined variable bindings in the Global Environment. It reports some additional useful information such as the type of x as well as its length.

5.3.3. Options



- [ ] Warn Partial Args
- [✓] Show Warnings

Variable	Type	Length
x	double	1
z	double	1

The right panel contains a small list of output options. R provides a large number of options for interactive use, for example formatting the width of the printed output, how many warnings to print, what types of warnings to generate, etc. via the `options` builtin. We have only replicated 2 options:

- Warn Partial Args: Generate warnings if an argument is matched in a function by a partial name matching, e.g. a function `generateSomething(x, output)` is called as `generateSomething(10, out=TRUE)` then out is partially matched to output. This may be a source of unexpected behavior while convenient, thus warnings can optionally be generated for such patterns
- Show Warnings: Whether to print warnings or not

The options can simply be toggled by checking or unchecking the checkbox.

---

6. Function reference

This section will act as a reference to the primitives and internals we have implemented in joRdan. Since our implementation is near identical to R's implementations, links to R documentation will be provided, and the variations from their implementations will be indicated appropriately.

6.1. Primitives

6.1.1. Language construct functions

6.1.1.1. Control flow (R Docs - Control function)

- `if`
- `for`
- `while`
- `repeat`
- `break`
- `next`

6.1.1.2. Function definition (R Docs - Function definition)

- `function`
- `return`

6.1.1.3. Parantheses and braces (R Docs - Parentheses and Braces)

- `{`
- `(`

6.1.1.4. Call an Internal function (R Docs - Call an Internal Function)

- `.Internal`

6.1.1.5. Call a Primitive function (R Docs - Call a Primitive Function)

- `.Primitive`

6.1.2. Arithmetic operators

(R Docs - Arithmetic Operators)

6.1.2.1. Unary arithmetic operators

- `+`
- `-`

6.1.2.2. Binary arithmetic operators

- `+`
- `-`
- `*`
- `/`
- `^`
- `%%`
- `%/%`

🚫 joRdan's operators are NOT internal generics (unlike R) thus we cannot define S3 class methods on primitive operators.
Internal generics are primitive functions (implemented in the interpreting language, which is C in the case of R), but which can still be dispatched to R methods based on the class of the object. These are not supported in joRdan

6.1.3. Logical operators

(R Docs - Logical Operators)

- `!`
- `&`
- `|`
- `||`
- `&&`

🚫 Logical operators are NOT internal generics

6.1.4. Relational operators

(R Docs - Relational Operators)

- `<`
- `>`
- `<=`
- `>=`
- `==`
- `!=`

🚫 Relational operators are NOT internal generics

6.1.5. Subsetting and assignment

6.1.5.1. Assignment (R Docs - Assignment Operators)

- `<-`

- `<<-`

6.1.5.2. Subsetting and subset assignment (R Docs - Extract or Replace Parts of an Object)

- `[`
- `[[`
- `$`
- `[<-`
- `[[<-`
- `$<-`

🚫 joRdan only supports subsetting and subset assignment operations on vectors, lists and pairlists.
For lists, using `x[[i]] <- NULL` will not remove a component (which is what R does); instead, it will set it to `NULL`.

🚫 Subsetting functions are NOT internal generics

6.1.6. Attribute-related functions

6.1.6.1. Object attributes (R Docs - Object Attributes)

- `attr`
- `attr<-`

🚫 joRdan does not support partial matching during getting of an attribute via `attr`. The `exact` argument is also not usable in joRdan.

6.1.6.2. Object attribute lists (R Docs - Object Attribute Lists)

- `attributes`
- `attributes<-`

6.1.6.3. Object names (R Docs - The Names of an Object)

- `names`
- `names<-`

🚫 joRdan only supports assignment of `names` to vector objects, lists and pairlists.

🚫 `names` and `names<-` are NOT internal generics

6.1.6.4. Object classes (R Docs - Object Classes)

- `class`
- `class<-`

6.1.6.5. Object dimensions (R Docs - The Dimensions of an Object)

- `dim`
- `dim<-`

🚫 Currently, joRdan will only do the necessary checks as per R for `dim<-`. Setting the dimensions will not do anything to the object: it will simply set the `dim` attribute of the object and leave it.

6.1.6.6. Object comments (R Docs - Query or Set a "comment" Attribute)

- `comment`
- `comment<-`

6.1.7. Mathematical functions

6.1.7.1. Trigonometric functions (R Docs - Trigonometric Functions)

- `sin`
- `cos`
- `tan`
- `asin`
- `acos`
- `atan`
- `atan2`
- `sinpi`
- `cospi`
- `tanpi`

6.1.7.2. Hyperbolic functions (R Docs - Hyperbolic Functions)

- `sinh`
- `cosh`
- `tanh`
- `asinh`
- `acosh`
- `atanh`

6.1.7.3. Rounding of numbers (R Docs - Rounding of Numbers)

- `ceiling`
- `floor`
- `sign`
- `trunc`
- `round`
- `signif`

6.1.7.4. Logarithms and exponentials (R Docs - Logarithms and Exponentials)

- `log`
- `log2`
- `log10`
- `log1p`
- `exp`
- `expm1`

6.1.7.5. Summary functions

- `sum` (R Docs - Sum of Vector Elements)
- `min` (R Docs - Maxima and Minima)
- `max` (R Docs - Maxima and Minima)
- `prod` (R Docs - Product of Vector Elements)

- `mean` (R Docs - Arithmetic Mean)

> 🚫 joRdan does not support the inputting of the `trim` argument in `mean`.

- `range` (R Docs - Range of Values)

> 🚫 Summary functions are NOT internal generics

6.1.7.6. Miscellaneous mathematical functions (R Docs - Miscellaneous Mathematical Functions)

- `abs`
- `sqrt`

6.1.8. Type checking and coercion

6.1.8.1. Type checking functions

- `is.null` (R Docs - The Null Object)
- `is.logical` (R Docs - Logical Vectors)
- `is.integer` (R Docs - Integer Vectors)
- `is.double` (R Docs - Double-Precision Vectors)
- `is.character` (R Docs - Character Vectors)
- `is.symbol` (R Docs - Names and Symbols)
- `is.name` (R Docs - Names and Symbols)
- `is.environment` (R Docs - Environment Access)
- `is.list` (R Docs - Lists -- Generic and Dotted Pairs)
- `is.pairlist` (R Docs - Lists -- Generic and Dotted Pairs)
- `is.expression` (R Docs - Unevaluated Expressions)
- `is.na` (R Docs - 'Not Available' / Missing Values)
- `is.nan` (R Docs - Finite, Infinite and NaN Numbers)
- `is.finite` (R Docs - Finite, Infinite and NaN Numbers)
- `is.infinite` (R Docs - Finite, Infinite and NaN Numbers)
- `is.numeric` (R Docs - Numeric Vectors)
- `is.vector` (R Docs - Vectors)

> 🚫 Note that not all the content in these individual document pages apply to joRdan; only the ones relevant to the functions above are valid.

6.1.8.2. Type coercion functions

- `as.logical` (R Docs - Logical Vectors)
- `as.integer` (R Docs - Integer Vectors)
- `as.double` (R Docs - Double-Precision Vectors)
- `as.character` (R Docs - Character Vectors)
- `as.list` (R Docs - Lists -- Generic and Dotted Pairs)
- `as.pairlist` (R Docs - Lists -- Generic and Dotted Pairs)

🚫 Type checking and coercion functions are NOT internal generics

6.1.9. Miscellaneous

- `c` (R Docs - Combine Values into a Vector or List)

🚫 joRdan's version of `c` is only able to handle vector objects (namely logicals, integers, doubles and characters), lists and pairlists.
Due to joRdan's difference in types available compared to R, the type hierarchy is thus: NULL < logical < integer < double < character < pairlist < list

🚫 `c` is NOT an internal generic in joRdan

- `list` (R Docs - Lists -- Generic and Dotted Pairs)
- `:` (R Docs - Colon Operator)

🚫 As joRdan does not support factors internally (as of yet), the colon operator only works on numeric vectors. Attempting to perform interaction between 2 factors, e.g. `f1:f2` will result in an error.

- `length` (R Docs - Length of an Object)

🚫 The replacement function for length, `length<-` does not exist in joRdan. `length` is also NOT an internal generic

6.1.10. Generics

- `UseMethod` (R Docs - UseMethod: Class Methods)
- `NextMethod` (R Docs - UseMethod: Class Methods)

joRdan's S3 system is a slight simplification of R's S3 system, which is itself an implementation of the S Language's object system.

Differences from R:
1. `UseMethod` always dispatches on the class of the first argument of the closure that called `UseMethod`, and passes the closure arguments to the dispatched method
2. R makes a distinction between the *calling environment* where the generic is called, and the *definition environment* where the generic function is defined. In joRdan we have no such distinction since our search path structure is simplified (See the note in 3.1.8. Environments) which means there is a single unified search path in the program.
3. The only S3 variables added in the dispatch are `.Generic`, `.Method` and `.Class` which are character vectors denoting the name of the generic function that was dispatched, the method that was dispatched to and the classes of the dispatching object. `.GenericCallEnv` and `.GenericDefEnv` are not created, because joRdan does not distinguish call environment and definition environment for generic functions (see pt 2)
4. `NextMethod` is a primitive, not internal, simplifying the generic CallStack context lookup. If no arguments are supplied (i.e. simply calling `NextMethod()` the same arguments in the current method are passed to the next method. Else the current dispatched object, together with the supplied arguments, are passed.

For example, let `x` be an object with classes `c("a", "b")` and let `foo` be a generic function with methods `foo.a` and `foo.b`. If the generic method `foo(x, bar=TRUE)` is being executed (thus the method being dispatched to is `foo.a`, and `NextMethod()` is called within `foo.a`, then `foo.b` will be called with the arguments `foo.b(x, bar=TRUE)`. However if it is called with arguments, e.g. `NextMethod(baz=c(1,2,3), FALSE, y)` then `foo.b` will be called with these arguments supplied instead, i.e. `foo.b(x, baz=c(1,2,3), FALSE, y)`.

5. *Internal Generics* are not supported (has been mentioned multiple times in previous notes). Internal generics are primitive functions (not closures that call `UseMethod`) but users can still define methods for them in R which will be dispatched correctly. In joRdan to maintain consistency, the only way to perform method dispatch is the `UseMethod` primitive.

6.1.11. Output Manipulation

- `invisible` (R Docs - Invisible)

6.2. Internals/Standard Library

These functions are defined in the base library loaded when the interpreter starts. Most of them are simply wrappers for internal functions that perform simple input validation or supply defaults.

6.2.1 String manipulation library

6.2.1.1. Pattern matching and replacement (R Docs - Pattern Matching and Replacement)

- `grep`
- `grepl`
- `sub`
- `gsub`

Utilizes javascript regex which may differ in behavior from R's slightly, especially regarding capture groups. Refer to (MDN Javascript RegEx guide) and (MDN string replacement) for more information about the string syntax for regex searching and replacement.

Does not have the `perl` and `useBytes` arguments (since we are using Javascript strings and Javascript regex)

6.2.1.2. Character translation (R Docs - Character Translation and Casefolding)

- `tolower`
- `toupper`

- `casefold`

6.2.1.3. String tests (R Docs - Does String Start or End With Another String?)

- `startsWith`
- `endsWith`

6.2.1.4. Character counting (R Docs - Count the Number of Characters)

- `nchar`

> 🚫 Does not support the `type` and `allowNA` arguments

6.2.1.5. Substring (R Docs - Substring of a Character Vector)

- `substr`

6.2.1.6. Repeating characters (R Docs - Repeat the Elements of a Character Vector)

- `strrep`

6.2.2. Generics

- `inherits` (R Docs - Object Classes)

> 🚫 The output of `inherits` in joRdan differs slightly from R's to more accurately reflect the actual class dispatch order in S3 dispatch

6.2.3. Printing output

`print` is the generic function, `print.default` is the internal default method.

- `print`
- `print.default`

> 🚫 joRdan only supports simply printing out the object string representation, and does not support formatting options as per R's `print.default` (R Docs - print.default: Default Printing)