# 1 Syntax Bindings

Since every expression in Anterogradia must either be a string literal or a function call, **syntax bindings** were introduced in order to solve the readability and practicality issues. Syntax bindings are nothing more than fancy syntactical entities that are directly translated into standard library function calls by the parser. Take a look at the following piece of code as an example:

```
progn {
    fun sayHi {
        "Hello, World!"
    }
    eval sayHi
}
```

This code is simple enough to be able to almost immediately notice the two obvious primitive expressions:

- 1. The **variadic function** call of progn
- 2. The string literal "Hello, World!"

However, here, fun and eval don't match the established syntax for any ANTG primitive. Thus, you'd be correct to conclude that they are in fact syntax bindings. With the help of the *astd* (AST Dump) function we can take a peek behind what's going in the program above:

```
astd(expr = progn { ...
Yields:

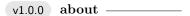
progn {
    _fun(
        expr = progn {
        "Hello, World!"
     },
     id = "sayHi"
    ),
    _eval(id = "sayHi")
}
```

Here you can see that the fun entity has been translated to a \_fun call with a progn call as its return value. The function identifier is now also provided as a discrete parameter. As for the eval entity, it was also changed a bit by turning into an \_eval call with the function id as its parameter. The difference is not particularly striking in this example, but it's enough to establish what this technique is all about. Syntax bindings really start to shine when your code samples slightly grow in complexity, as illustrated by the following example:

```
Here, the original code ...
progn {
  fun sayHi <to> {
    sequence { "Hello, " & to "!" }
  eval sayHi(to = "World")
}
\dots expands to \dots
progn {
  _fun(
    expr = progn {
      __require_prop(
        err = "Required prop not present",
        id = "to"
      ),
      progn {
        sequence {
           "Hello, ",
           get(key = "to"),
      }
    },
    id = "sayHi"
  ),
  progn {
    set(value = "World", key = "to"),
    _eval(id = "sayHi")
}
```

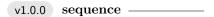
To briefly summarize what the parser has done, the most noticeable change is the transformation of wouldbe discrete function parameters to variable declarations prior to evaluating the stored function. The parser also generates safeguards at the beginning of the function body to ensure that all required variables are in fact present; this is achieved via the \_require\_prop function, which checks for the existence of a variable id and causes the runtime to throw an AntgRuntimeException with the error message err whenever it fails to locate said variable. It is also worth mentioning that since the parser has no notion of functions and variables (after all, this functionality is implemented in the standard library) there is no way for it to check the validity of the parameters passed to the eval entity, and thus it will transform any discrete parameters into variable declarations, regardless of whether they're actually required by the callee.

# 2 Standard Library Functions



It is mandatory for all Anterogradia libraries to implement an *about* function. The standard library is no exception to this rule. The standard implementation provides basic information about the lib at hand.

about ()



The **variadic** sequence function evaluates every parameter expression and returns a string made up of all individual results.

```
sequence {
    "Hello, "
    "World!"
}
=> "Hello, World!"
```

### v1.0.0 progn —

Just like in Common Lisp, the **variadic** progn function evaluates all parameters in sequence and returns the last value.

```
progn {
    "Hello, "
    "World!"
}
=> "World!"
```

### v1.0.0 nothing —

Returns an empty string (a string with length 0).

```
nothing ()
=> ""
```

```
v1.0.0 repeat —
```

Repeats the expression str for count times, each iteration separated by an optional separator, otherwise unseparated.

```
v1.0.0 random —
```

The **variadic** function *repeat* randomly evaluates a single expression.

```
random {
    "Foo"
    "Bar"
    "Baz"
}

(1) => "Bar"
(2) => "Baz"
(2) => "Baz"
(3) => "Bar"
(4) => "Foo"
...
```

```
v1.0.0 _ if _____
```

The \_if function implements conditional control flow. If cond is "true", the then expression will be evaluated and returned as the result of the function. Otherwise, the function evaluates and returns the else expression.

```
v1.0.0 equal ———
```

The equal function compares the expressions left and right. If both have the same value, the function returns "true", otherwise it returns "false"

```
equal (left = "123", right = "321")
=> "false"
```

```
v1.0.0 param —
```

Anterogradia may be started with custom startup parameters from within the Kotlin API. This function is used to retrieve said parameters, with *key* being the key of a given startup parameter entry.

```
param (key = "binaryPath")
```

This function together with *get* implement the backbone of Anterogradia's memory features. This function creates or modifies a variable identified by a *key* with a given *value*. This function always returns an empty string.

```
set (key = "message", value = "Hello, World!")
=> ""
```



The get function retrieves a variable key and returns the value.

## v1.0.0 compile —

Thsi function is used to dynamically invoke the Anterogradia interpreter while re-using the current runtime object. Thus, all libraries, functions and variables present in the host script are going to be usable in the code passed to the *source* parameter.

```
progn {
    set (key = "msg", value = "Hi!")
    compile (
        source = "&`msg"
    )
}
    => "Hi!"
```

### v1.0.0 lgt —

The lgt function compares the expressions left and right and returns "true" if the former is greater than the latter; otherwise "false" Depending on the value of both expressions, the comparison will either be numeric or lexicographic.

v1.0.0 len —

Returns the length of the *expr* string.

#### v1.0.0 astd —

Generates valid Anterogradia source code from the parser result of the passed *expr*.

```
v1.0.0 fun —
```

This function stores the *expr* expression as *id*. It is worth mentioning that, unlike variables, what gets stored is not the result of evaluating the given expression, but rather the AST nodes making up said expression. Thus, evaluating such stored expressions **might** yield different values on each iteration.

```
_fun (id = "greet", expr = & abc)
```

```
v1.0.0 _eval ____
```

The \_eval function is closely related with the \_fun function. Its purpose is to retrieve the expression *id* stored via the former function and evaluate it at a given point in time.

```
sequence {
    set(key = "abc", value = "Hi!")
    _eval (id = "greet")
    " "
    set(key = "abc", value = "Hello!")
    _eval (id = "greet")
}
    => "Hi! Hello!"
```

### v1.0.0 \_\_\_require\_prop \_\_\_\_

This function checks for the existence of the variable id and causes the interpreter to throw an AntgRuntimeException with the err message whenever it cannot find the required variable. Note that the existence of a variable is determined by the value of its length being > 0

```
__require_prop(
   id = "abc",
   err = "Variable not found!")
```

(This causes the runtime to throw the aforementioned exception, since the variable is not present in this context. This also means that the execution of the script will be interrupted at this exact point.)

It is worth mentioning, that this function is a utility designed to implement reliable function calls and

was originally meant to be generated exclusively by the function definition syntax binding. It is not recommended to use it manually.

Evaluates to the result of adding the *left* and *right* operands together.

Evaluates to the result of subtracting the right operand from the left operand.

### v1.0.0 mul —

Evaluates to the result of multiplying the left and right operands together.

Evaluates to the result of dividing the left operand by the right operand.

Evaluates to the result of retrieving the division remainder of *left / right*.

## v1.0.0 signflp —

Evaluates to expr with a flipped sign.

#### v1.0.0 vsignflp —

Performs a sign-flip on the variable id and stores the result in the source variable.

```
progn {
    set (key = "a", value = "12")
    vsignflp (key = "a")
    &`a
}
    => "-12"
```

#### v1.0.0 increment —

Increments the value of the variable id and stores the result in the source variable.

```
progn {
    set (key = "a", value = "10")
    increment (id = "a")
    &`a
}
    => "11"
```

### v1.0.0 decrement —

Decrements the value of the variable id and stores the result in the source variable.

```
progn {
    set (key = "a", value = "10")
    decrement (id = "a")
    &`a
}
    => "9"
```