The BigDuck Programming Language Proposal

Jair Antonio Bautista Loranca

September 26, 2021

Contents

1 Objective

This document is to describe the general characteristics for the BigDuck programming language. BigDuck is language aimed for the developement of mathematical models commonly used in Machine-Learning and Data Science.

Therefore this language will include integer and floating point arithmetic, vector and matrix operations, and some basic utilities for reading and writing `.csv` files. All this with the purpose to make it easier for the user to work within the Machine-Learning and Data Science fields.

2 Language Requirements

2.1 Basic Elements

2.1.1 Identifiers

In this language identifiers can be build starting with any alphabetic character followed by the any sequence of alphanumeric characters or underscore characters, and terminated by any whitespace character. This being identical to the common convention used in many popular programming languages like C, Java, or Python.

2.1.2 Whitespace and comments

Whitespace charaters include spaces, tabs, newlines. The BigDuck compiler ignores whitespace characters, however the usage of this is recommend and may be use to align text to improve readability.

The token `#|` indicates the start of a comment and must be closed by the token `|#` to end the comment. All the text inside this tokens will be ignored by the compiler. Comments are just help clarify or document programs.

2.1.3 Reserved Keywords

`proc`	`return`	`if`	`else`
`loop`	`break`	`skip`	`and`
`or`	`not`	`var`	`int`
`float`	`bool`	`true`	`false`

Most of these keywords are used in many programming languages so they do not require explanation, however some some of them defer.

`proc` is use to specify a procedure declaration.

`loop` is use instead of the usual `while`, `do while`, and `for` keywords. The rational is that all of these can be rewritten into a for-loop, however the traditional for-loop reading gets messed up when doing this, therefore the `loop` keyword was chosen instead.

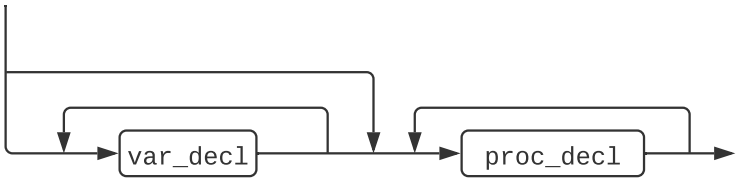
`skip` is use instead of the usual `continue`. The rational for this replacement is that it is more descriptive of the action performed.

2.1.4 Other notations

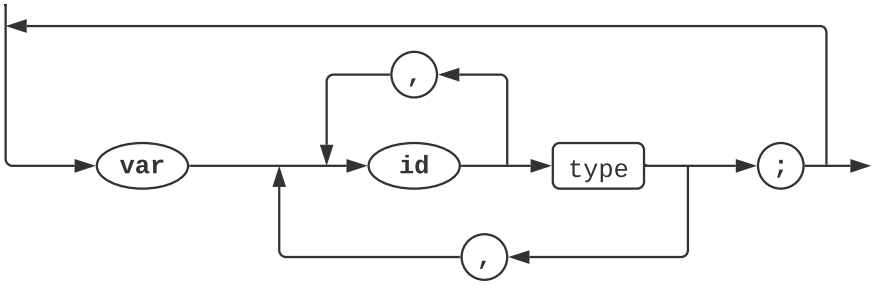
There are several tokens left out from this section, however these will be mention within their syntactical or semantic context. This is to avoid redundancy on this document.

2.2 Syntax Diagrams

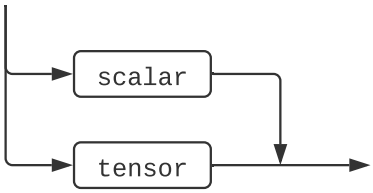
< program >



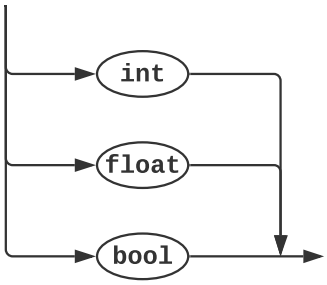
< var_decl >



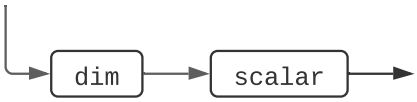
< type >



< scalar >

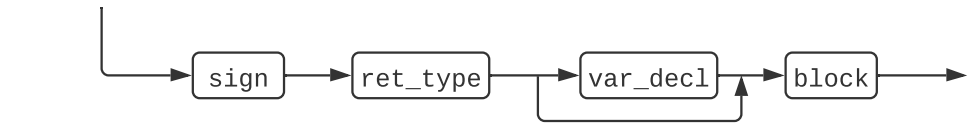


< tensor >

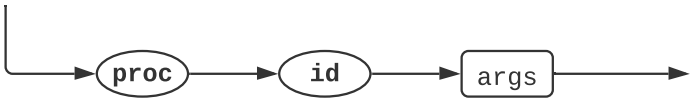


< dim >

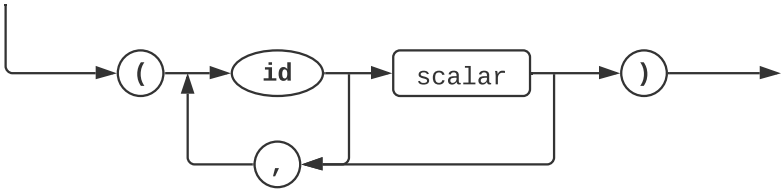
< proc_decl >



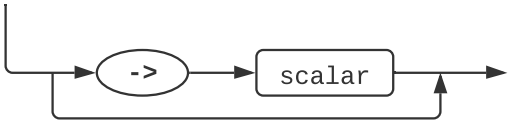
< sign >



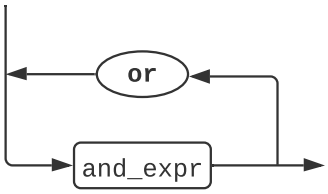
< args >



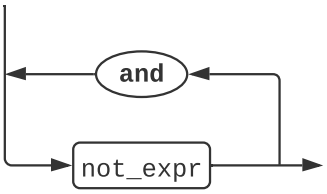
< ret_type >



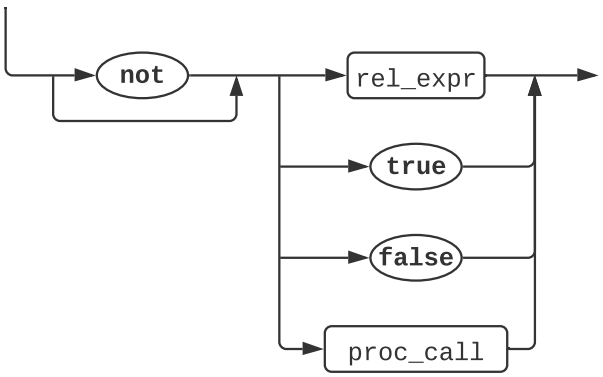
< bool_expr >



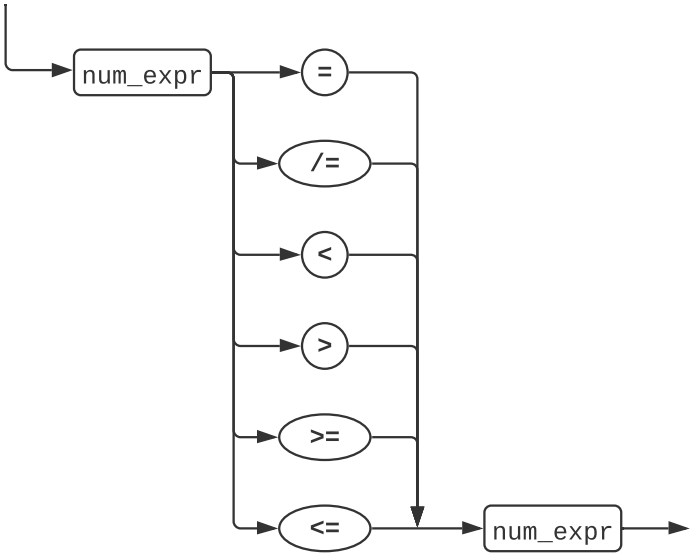
< and_expr >

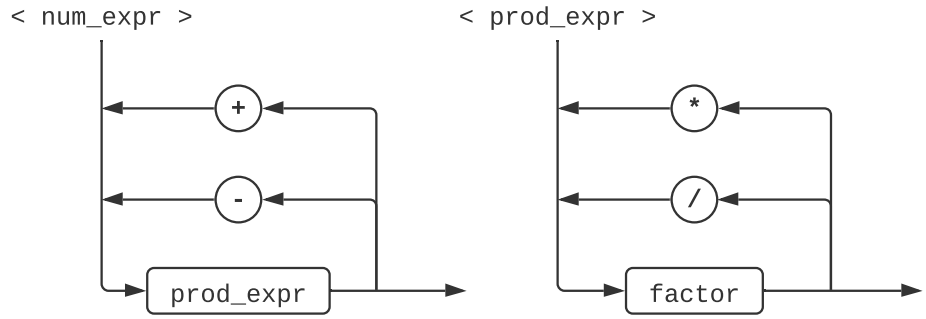


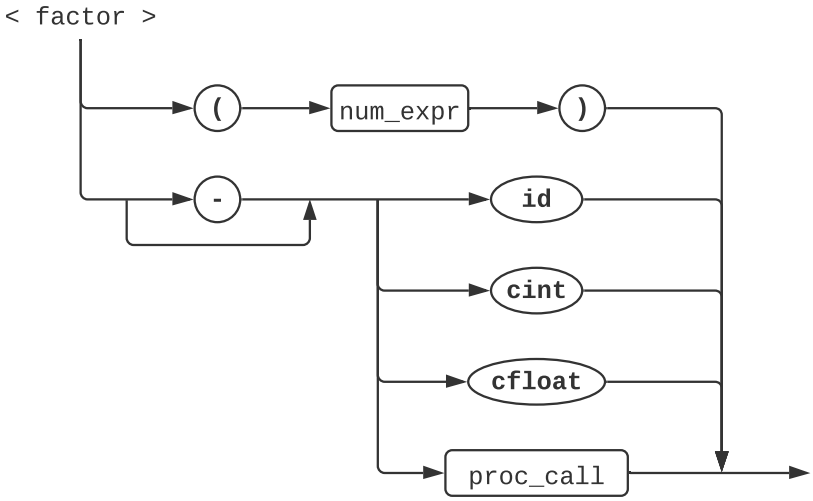
< not_expr >

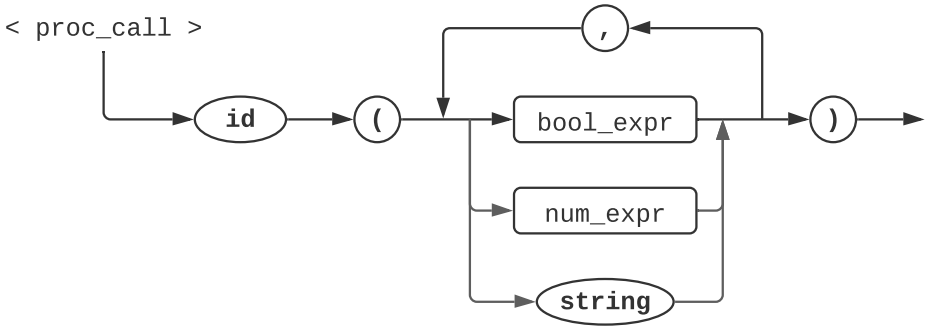


< rel_expr >

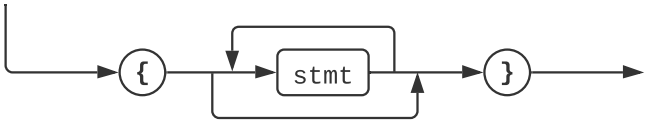



< num_expr >
+
-
prod_expr
< prod_expr >
*
/
factor

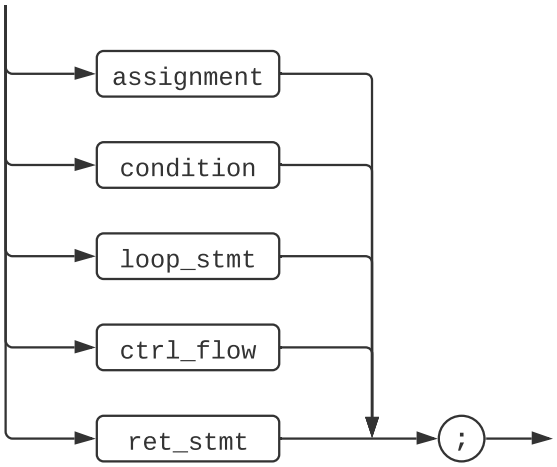

< factor >
(
num_expr
)
-
id
cint
cfloat
proc_call


< proc_call >
,
id
(
bool_expr
)
num_expr
string

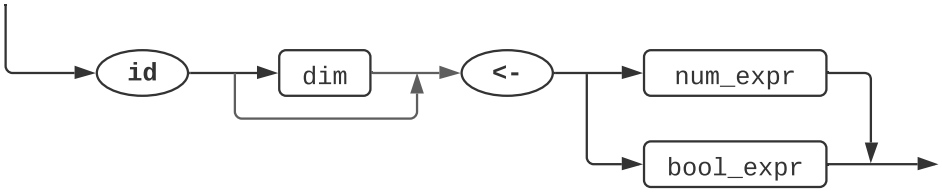
< block >



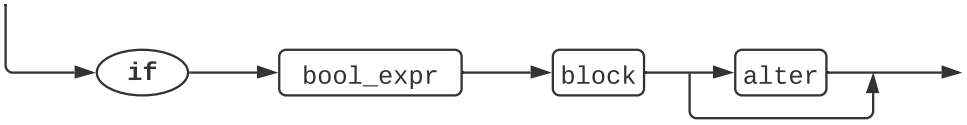
< stmt >



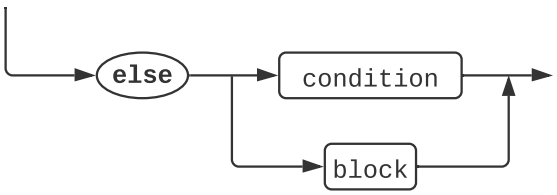
< assignment >



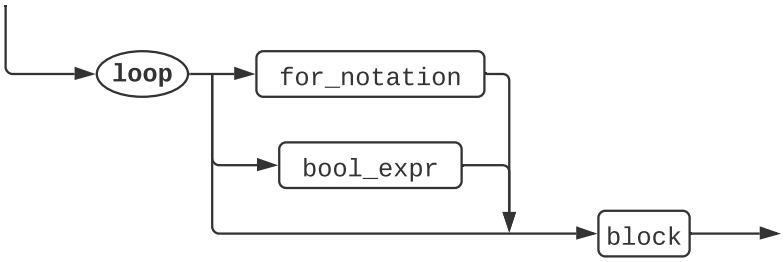
< condition >



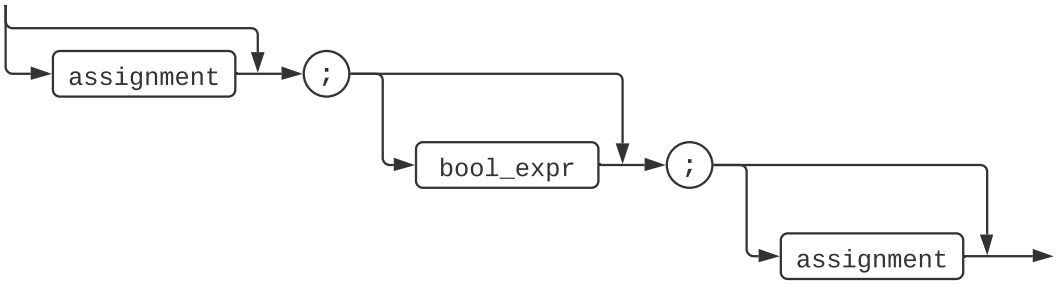
< alter >



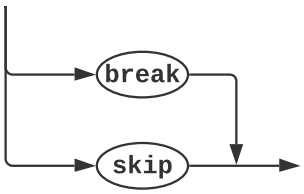
< loop_stmt >



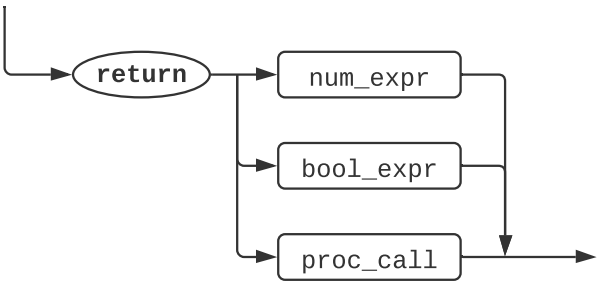
< for_notation >



< ctrl_flow >



< ret_stmt >



2.3 Sematic Characteristics

Following popular programming languages like C, Java, Go, etc, BigDuck is a statically scoped language. Meaning that each variable exists within their scope of definition. However there are only two scopes: global scope and procedure scope. BigDuck is a statically typed language, meaning that variables can only contain values of the same type.

Arguments to procedures are always passed by value, and all parameter expressions in a procedure call are evaluated first wether the procedure needs the result of the evaluation or not. As opposed to lazy evaluation done in some functional languages like Haskell.

There must be at least one procedure defined, the last procedure on the program will be taken as the main procedure for the program.

2.4 Built-in procedures

IO operations

```
#| Prints any object given |#
proc print(any)

#| Reads value from input|#
proc read(any) -> any

#| Opens csv file |#
proc open_csv(filename, sep string, header bool) -> [][]any

#| Writes object to a csv file |#
proc write_csv(filename string, any)
```

Math operations

```
#| Returns the arccosine for a given angle |#
proc acos(x float) -> float

#| Returns the arcsine for a given angle |#
proc asin(x float) -> float

#| Returns the arctangent for a given angle |#
proc atan(x float) -> float

#| Returns the arctangent for the quotient y / x |#
proc atan2(y, x float) -> float

#| Returns the cosine for a given angle |#
proc cos(x float) -> float
```

```
#| Returns the sine for a given angle |#
proc sin(x float) -> float

#| Returns the tangent for a given angle |#
proc tan(x float) -> float

#| Returns the hyperbolic cosine for a given angle |#
proc cosh(x float) -> float

#| Returns the hyperbolic sine for a given angle |#
proc sinh(x float) -> float

#| Returns the hyperbolic tangent for a given angle |#
proc tanh(x float) -> float

#| Returns e to power of the given exponent |#
proc exp(x float) -> float

#| Returns the natural logarithm for a given number |#
proc ln(x float) -> float

#| Returns the logarithm base b for a given number x |#
proc log(x, b float) -> float

#| Returns the mod base b for a given number x |#
proc mod(x, b float) -> float

#| Returns the remainder of the quotient x / b |#
proc rem(x, b float) -> float

#| Returns x to the power of the given exponent y |#
proc pow(x, y float) -> float

#| Returns the squared root for a given number |#
proc sqrt(x, y float) -> float

#| Returns the absolute value for a given number |#
proc abs(x, y float) -> float

#| Returns the ceiling for a given number |#
proc ceil(x float) -> float

#| Returns the floot for a given number |#
proc floor(x float) -> float
```

Vector operations

```
#| Returns the magnitude for a given vector |#
proc mag(v1 []float) -> float

#| Returns the euclidean distance for two given vectors |#
proc dist(v1, v2 []float) -> float

#| Returns unit vector with same direction for a given vector |#
proc normalize(v1, v2 []float) -> float

#| Returns the dot product for two given vectors |#
proc dot(v1, v2 []float) -> float

#| Returns the cross product for two given vectors |#
proc cross(v1, v2 []float) -> []float

#| Returns the angle between two given vectors |#
proc angle(v1, v2 []float) -> float
```

Matrix operations

```
#| Returns the identity matrix |#
proc id_mat() -> [][]float

#| Returns the transpose matrix for a given matrix |#
proc transpose(m [][]float) -> [][]float

#| Returns the determinant for a given matrix |#
proc det(m [][]float) -> [][]float

#| Returns the adjugate matrix for a given matrix |#
proc adj(m [][]float) -> [][]float

#| Returns the inverse matrix for a given matrix |#
proc inv(m [][]float) -> [][]float
```

2.5 Data types

The BigDuck programming language uses the following data types.

- Scalar types
  - `int` Integer types
  - `float` Real types
  - `bool` Boolean types
- `[n][m]...<scalar type>` Tensorial types

Scalar types consists of individual values, represented within a numeric scale. Real types are not feasable to represent within a computational environment thus they will be represented by floating point numbers like any other programming language.

Tensorial types is the math term equivalent to multidimensional arrays, thus a 1D tensor is an array, a 2D tensor is a matrix, and a 3D tensor is a cube.

Tensorial types have fixed size, so once declared they can not be resized. Vector and matrix operations are determined by the dimensions of the tensor. And lastly all elements of a tensorial types are from the same scalar type.

Despite strings being mentioned on some syntax and function signatures, they are only string literals used for some built-in procedures that require them for IO.

3 Implementation environment

The BigDuck compiler will be developed using the Go programming language. Antlr4 will be used as lexer and parser generator. And it will be developed on MacOS, any other system support is not considered.