Progvo

Documentation (draft)

Version 0.9α (draft), November 11, 2019

Thomas Nguyen (Pttn)

Progvo.dev

Contents

1. Introduction

1.1 This document

It is the official documentation of Progvo, downloadable in its website.

`https://Progvo.dev/Dok/Dokumentado_En.pdf`

Tutorials and examples are outside the scope of the documentation, see instead other documents, like the official website of the project `https://Progvo.dev/Lerni/` (in Esperanto).

Thank you for your interest in Progvo!

1.2 About Progvo

Progvo is a general purpose programming language created by Thomas Nguyen (Pttn) in 2019[2]. It is a new project, and the long term goal is to have a simple, versatile and productive language.

It does not have a strict paradigm, but has various properties of object-oriented, functional and procedural languages.

Programs written in Progvo should depend the less possible on the system, making it a high-level programming language. However, low-level features can also be available in some Progvo implementations.

1.3 Audience

Progvo is meant to be usable by any programmer and should be simple enough for beginners, it could be a fun first programming language for new programmers. Give it a try!

1.4 Implementations

Implementations of Progvo must follow the documentation. Else, there are no more conditions and there could for example be Progvo interpretors as well as compilers.

They may also implement more *external* classes or functions than the ones in the documentation, but must not allow new or different syntaxes or semantics.

[1] `https://creativecommons.org/licenses/by-sa/4.0/legalcode`

[2] A first version of the documentation was published the November 11th 2019.

2. Base

2.1 Language

2.1.1 Alphabet and words

Progvo uses the *Ekio*[1] character encoding. The *alphabet* Σ of the programming language is:

- Space, tabulation and line break (Σ_{spa});
- Digits (Σ_{cif}): `0123456789`;
- Letters (Σ_{lit}): `abcĉdefgĝhĥijĵklmnoprsŝtuŭvzqwxyABCĈDEFGĜHĤIJĴKLMNOPRSŜTUŬVZQWXY`;
- Delimiters (Σ_{lim}): `↓↑↔|/\,;.:[]{}`;
- Other special characters (Σ_{spe}): $+ - \times \div = \neg \vee \wedge \Leftrightarrow \infty <> \emptyset \in$`*°′∫^~±Δ□○%‰’"!?§#@★-_`.

If the code has a chacacter $s \notin \Sigma$, it is invalid and no program can be created (support of UTF-8 and other common encodings can come in future versions of Progvo).

2.1.2 Comments

One *comments* the code using the symbol `§`. If `§` is followed by `[`, `§` and characters after it until `]` or the end of file are ignored.

Else, `§` and characters after it until the next line break or the end of file are ignored.

2.1.3 Tokens

Tokens are keywords, delimiters, identifiers and constants. They are separed with (a) space(s), but delimiters can also be glued with other tokens.

Tabulations and line breaks are considered as spaces. Multiple spaces are the same as only one space.

Keywords

Progvo has the following *keywords* (set $\hat{S}$): `ERARO`, `INKLUZIVI`, `FINO`, `NOMSPACO`, `EKSTERA`, `KLASO`, `KONSTANTO`, `KONSTRUILO`, `SE`, `ALIE`, `RIPETI`, `DUM`, `HALTI`, `ELIRI`.

They are case sensitive.

Delimiters

Delimiters (set L) are single symbols from Σ_{lim}. For example, `.`, `;`, `{`, `)`, `↦`.

[1] See `https://progvo.dev/Ekio.html` for the official table.

Identifiers

Identifiers are $I = ((\Sigma \setminus (\Sigma_{lim} \cup \Sigma_{spa}))^+ \setminus (\hat{S} \cup K)$. For example, `abc`, `+`, `a1b2c3`, `7a8b9c`, but not ϵ (empty word), `.`, `[]`, `SE`, `→`, `7a8 b9c`, or `a→b`.

Constants

Constants (set K) are special words, representing a constant object of a class defined in the documentation.

2.2 Basic concepts and syntax

Starting from now, the code must have been tokenized for parsing.

For simplicity, the syntax will be informally defined in this section, but should be precise enough in order to avoid ambiguity. Phrases between `<` and `>` *must* (including `<` and `>`) be replaced by (a) valid token(s) explained by the phrase, those between `[` and `]` *can* be replaced by (an) appropriate token(s).

2.2.1 Instructions and sequences of instructions

In Progvo, the idea of *instruction* is very general and an instruction can be different things like declarations, definitions or evaluations. Ordered groups of instructions (which can be empty) are *sequences of instructions*, which is itself an instruction.

Every instruction must end with a semicolon `;`, except sequences of instructions, which must be between `{` and `}` (except the main sequence, which does not belong to another one).

Declarations and definitions in sequences of instructions can only be used in the same sequence (and its subsequences).

2.2.2 Expressions

An *expression* is a group of tokens, which can be evaluated to an object or list of objects. See below of the definition of objects and how the evaluation works.

Expressions are not instructions, but the evaluation of a full expression, which is not part of another instruction, is an instruction.

2.2.3 Lists

Lists are only convenience syntax, which allow in particular to use functions with multiple output values.

They are nested, and the general syntax is:

- Empty list: `()`, cannot be part of a list of deepness ≥ 1 (for example, `((a, b), (), c)` is not a valid list);
- List of deepness 1: `(<item>[, <item>[, ...[, <item>]...]])`;
- List of deepness > 1: `(<list>[, <list>[, ...[, <list>]...]])`.

Lists are not instructions.

Object lists

`item` is an identifier of an object.

Input or output lists

`item` is a declaration without assignment of an object (see below).

Expression lists

`item` is an expression.

2.2.4 Objects

An *object* is an instance of a class (see below), which has data (a state) and methods, according to its class.

Declarations and definitions of objects are instructions.

Declarations

One can create named objects using *declarations*, with the syntax

```
<identifier (of the class)> <identifier (of the object)>
```

This creates a new object of the provided class.

Assignments

An *assignment* is replacing the state of an object by the one of another object. One assigns object(s) with

```
<identifier (of the object)> ← <compatible expression>
```

aŭ

```
<list of objects> ← <compatible expression>
```

The expression must evaluate to an object or list of objects of the same class(es) of the assigned object(s).

One can also both declare and assign an object in one instruction with

```
<declaration of object> ← <compatible expression>
```

The object can be constant:

```
KONSTANTO <declaration of object>[ ← <compatible expression>]
```

2.2.5 References

A *reference* is a special object, of class `K'` ("K Ref"), which allows the full access to an object of class `K` with an alias. References can be created like normal objects, with the same syntax.

The classes for references are automatically declared with their normal classes, don't need any definition, and are unique.

2.2.6 Classes

A *class* is a template representing a set of objects with common data types (attributes) and functions (methods).

Classes proposed by the Progvo implementation that are not fully defined in the source code are *external classes*. Objects of external classes have a state, which may not only depend on their attributes, but also on the Progvo implementation; the dependence must otherwise be complete for other classes.

Declarations and definitions

External classes must be declared before their use, with the syntax

```
EKSTERA KLASO <identifier>
```

As soon as an external class is declared, it can be used as any other class.

The definition of new classes (user classes) starts with `KLASO classname :` and ends with `FINO KLASO`. In the definition, the attributes are defined as declarations without assignments of objects, and methods with the same syntax as normal functions.

Declarations and definitions of classes are instructions.

Constructors

A constructor is a special method, which creates a new object of the class. The syntax is

```
KONSTRUILO <input list> <sequence of instructions>
```

Use

One accesses the attributes (encapsulation will come in future versions of Progvo) with

```
<identifier (of the object)>.<identifier (of the attribute)>
```

Methods are used with

```
<identifier (of the object)>.<call of function>
```

Constructors are used like normal functions, which name is the class name. The output is an object of the class.

2.2.7 Functions

A *function* is a structure with an input, an output and a sequence of instructions, which can be invoked. The output is the result of the evaluation of the call of the function, and depends on its input and instructions. Functions must be defined before their call, with the syntax

`<identifier> : <input list> → <output list> <sequence of instructions>`

The definition of a function is an instruction. In the instruction sequence, one can exit from the function with the instructon `ELIRI`.

A function is called with `<identifier (of the function)> <list of expressions>`.

2.2.8 Control structures

If statements

If statements allows to execute instructions only if a condition is verified. The syntax is

```
SE <expression 1> <instruction sequence 1>
[ALIE SE <expression 2> <instruction sequence 2>
[ALIE SE <expression 3> <instruction sequence 3>
[...
[ALIE SE <expression n> <instruction sequence n>]...]]]
[ALIE <instruction sequence n + 1>]
```

All expressions must evaluate to an object of the external class `Buleo` (true or false).

The ith instruction sequence is executed if, and only if, the ith expression evaluates to true, and $\forall j < i$: the jth expression evaluates to false. The expression for `ALIE` is always true.

If statements are not insructions.

Loops

A *loop* allows to repeat instructions while a condition is verified. The syntax is

```
RIPETI [DUM <expression>] <instruction sequence>
```

The expression must evaluate to a `Buleo` object, and the sequence of instruction is repeated while it is true. With only `RIPETI`, the expression is always true.

One can break the loop with the instruction `HALTI`.

2.2.9 Namespaces

Classes and functions can be grouped with *namespaces*.

A namespace starts with `NOMSPACO <identifier>`, ends with `FINO NOMSPACO`, and is not an instruction. Namespaces can be nested.

All classes and functions defined in a namespace `N` must be used with the prefix `N/` when outside of the namespace. If `N` is itself in the namespace `M`, the prefix is `M/N/`.

In a namespace, the prefix `/` can be used in declarations and definitions; they would be treated as if they were outside the namespace. In other cases, this also allows to use classes or functions outside of the current namespace.

2.2.10 Arrays

Will come in a future version of Progvo.

2.2.11 Templates

Will come in a future version of Progvo.

2.3 Semantics

2.3.1 Environments

An *environment* is a set of classes, functions and objects. Each instruction sequence has its own environment, which also includes the items of environments of parent sequences.

In an instruction sequence, each time that a class, function or object is defined or declared, it is added to the environment. Two items of the same environment cannot have the same name (there is no variable shadowing in Progvo), and it is not possible to use classes or objects, or call functions, that are not in the environment. One exception, two functions can have the same name if their inputs have different classes.

2.3.2 Forward declarations

User classes can be declared without definition, for example in case of circular dependences. The syntax is the same as the declaration of an external class, without the keyword `EKSTERA`.

In this case, the class must be defined later, and the control of the use of this class is done at the end of the processing of the source code.

2.3.3 Evaluations of expressions

The *evaluations* of expressions work like this:

- The objects and list of objects are the base elements of expressions;
- The constants, if the corresponding class is proposed by the Progvo implementation, is evaluated to the appropriate object;
- The calls of functions are evaluated like mathematical functions (the functions take the evaluation of their input and return objects or list of objects);
- Nothing else can be evaluated, and be an expression or part of expression.

2.3.4 Reference rules

One can create references that do not refer to any object, they are *uninitialized references*.

A reference cannot reference another reference, and it is not possible to change the referenced object (but an uninitialized reference can reference an object later).

If the reference and the referenced object are not constants, it is possible to assign the referenced object with the same syntax as the assigment of normal objects (for the expression compatibility, the referenced object is considered rather than the reference). An uninitialized reference will reference the new object.

2.4 Processing of the source code and execution

2.4.1 Environment variables

The Progvo implementation must have the following environment variables, which allow the programmer to adapt the behavior of the processing of the source code:

- `INKL`: the default folder for the include files.

2.4.2 Inclusions

Modular programming is possible with inclusions of code, using

The bit sequence must represent the relative path (to the current folder) in Ekio.

If the path starts with `/`, it is relative to the path given by the environment variable `INKL`.

Includes are instructions, but only useful for the code organization, and is replaced by the code of the file to include before the parsing.

2.4.3 Source code processing

In order to begin the processing of the source code, a source file must be provided. Then, the code is processed like this:

- The includes are extended (if an include instruction is invalid, the processing stops);
- The comments are removed, the constants recognized, and the code tokenized;
- The token parsing starts. If somewhere the syntax is invalid, the processing stops. Definitions and declarations are recognized and the validity of the environments and instruction sequences is controlled;
- The processing ends once all tokens were parsed, and only in this case a program can be created.

2.4.4 Entry point

The program starts with the call of the entry point function, which name is `enirejo`, and has empty input and output (entry points with arguments will come in future versions of Progvo).

If the program does not have an entry point, it cannot be directly executed (in future versions of Progvo, it could be used as library).

2.4.5 Undefined behavior

Some situations, pointed out in the documentation, cause *undefined behavior*: the program enters a state, which is not defined by the documentation. The Progvo implementation can choose how to handle the execution in undefined behaviors.

3. Core library

3.1 Presentation

The core library should propose the features, that are enough to code in Progvo in a practical way. However, Progvo being new and still incomplete, many features will come in future versions.

Every full implementation of Progvo must propose the features from this chapter. Programs using the core library must have the same behavior as explained in the documentation.

Simple Progvo implementations may not propose every feature. For example, embedded systems don't need an implementation of arbitrary large numbers and may only propose built-in numbers.

Attributes of external classes are defined in the documentation, but the implementation of the classes don't need to strictly implement them. However, the classes must behave in a way that is compatible with such strict implementation.

All classes and functions in the core library are in the namespace `Baz` and can be external. The Progvo implementation must provide the include file `/Bazo/Bazo.Pvo`, which contains the definitions and declarations of the classes and functions of the core library,

3.2 Core classes

3.2.1 Buleo

The external class `Buleo` allows the use of boolean values.

In the source code, one can create constant booleans with `VERA` and `MALVERA`.

Attributes

The class does not have any attribute, but objects have two possible states (values), true or false.

Metodoj

The class does not have any method.

Related functions

Comparison `/= : (KONSTANTO Buleo' a, KONSTANTO Buleo' b) → (Buleo)`, true if $a = b$, else false.

Not `/¬ : (KONSTANTO Buleo' a) → (Buleo b)`, with $b = \neg a$.

Or `/∨ : (KONSTANTO Buleo' a, KONSTANTO Buleo' b) → (Buleo c)`, with $c = a \vee b$.

And `/∧ : (KONSTANTO Buleo a, KONSTANTO Buleo' b) → (Buleo c)`, with $c = a \wedge b$.

3.2.2 Bitsekvenco

The external class `Bitsekvenco` allows to store and manipulate bit sequences.

A bit sequence is a sequence of $n \in \mathbb{N}$ bits (0 or 1). In the source code, one can create a constant bit sequence with the following syntax:

- Directly: `[`B`]`, with $B \in \{0;1\}^*$. For example, `[01011]` represents a sequence of 5 bits, which first is a 0 and last is an 1;
- With a number: `#`B`[`N`]`, with $B \in \{1; 2; ...; 16\}$ a base and $N \in \mathbb{N}$ a number, written in the right base (digits are `0123456789ABCDEF`). Le processing of the source code converts the number to binary base, as a Little Endian bit sequence, without leading zeroes. For example, `[010111]`, `#2[111010]` and `#10[58]` are the same bit sequence. If $B = 10$, one does not need to precise it: `#[58]` is the same as `#10[58]`;
- With Ekio text: `"[`T`]`, with T an Ekio string, with some special rules: `]`, `§`, `\` and characters $s \notin \Sigma$ cannot be directly used. They must use the escape character `\`, which must be followed by their Ekio number in a *two-digit* hexadecimal number (other characters can also use the escape character). All character corresponds to 8 bits. For example, `"[abc]` is the same `Bitsekvenco` as `[000100000010010001010]`. `"[\5B★/]` is the same as `[010110110111110101011010]` (and represents the string `\★/`).

Attributes

The class does not have any attribute, but objects have states, corresponding to the represented bit sequence.

Methods

Constructors

With a boolean `KONSTRUILO (KONSTANTO Buleo')`

The created object represents a bit sequence of length 1 that matches the value of the boolean.

Concatenation with another `Bitsekvenco`: `kunmeti : (KONSTANTO Bitsekvenco') → ()`

Length `longo : () → (Ind/N)`

Access the $(i+1)$th bit: `get : (KONSTANTO Ind/N' i) → (Buleo)`

If i is greater or equal to the sequence length, the call of this method causes undefined behavior.

Modify the $(i+1)$th bit: `set : (KONSTANTO Buleo' b, KONSTANTO Ind/N' i) → ()`

If i is greater or equal to the sequence length, the call of this method causes undefined behavior.

Related functions

Comparison `/= : (KONSTANTO Bitsekvenco' a, KONSTANTO Bitsekvenco' b) → (Buleo)`, true if a and b represent the same bit sequence, else false.

Bitwise Not `/¬ : (KONSTANTO Bitsekvenco' a) → (Bitsekvenco b)`

Bitwise Or `/∨ : (KONSTANTO Bitsekvenco' a, KONSTANTO Bitsekvenco' b) → (Bitsekvenco c)`

If the bit sequences do not have the same length, the call of this function causes undefined behavior.

Bitwise And `/∧ : (KONSTANTO Bitsekvenco a, KONSTANTO Bitsekvenco' b) → (Bitsekvenco c)`

If the bit sequences do not have the same length, the call of this function causes undefined behavior.

Concatenation `kunmeti : (Bitsekvenco, Bitsekvenco) → (Bitsekvenco)`

3.2.3 Ekio

The external class `Ekio` allows to store and manipulate Ekio strings.

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the Ekio string.

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The represented Ekio string of the created object corresponds to the bit sequence (one octet is one character). If the length is not a multiple of 8, the bits after the last octet are ignored.

Concatenation with another `Ekio`: `kunmeti : (KONSTANTO Ekio') → ()`

Length `longo : () → (Ind/N)` (number of characters)

Access the $(i+1)$th character: `get : (KONSTANTO Ind/N' i) → (Bitsekvenco)`

The bit sequence is the octet representing the character.

If i is greater or equal to the string length, the call of this method causes undefined behavior.

Modify the $(i+1)$th character: `set : (KONSTANTO Bitsekvenco' b, KONSTANTO Ind/N' i) → ()`

The bit sequence is the octet representing the character.

If i is greater or equal to the string length, or if the bit sequence is not an octet, the call of this method causes undefined behavior.

Related functions

Comparison `/= : (KONSTANTO Ekio' a, KONSTANTO Ekio' b) → (Buleo)`, true if a and b represent the same string, else false.

Concatenation `kunmeti : (KONSTANTO Ekio', KONSTANTO Ekio') → (Ekio)`

3.2.4 N

This class represents the mathematical set $\mathbb{N}$. The numbers have arbitrary size.

In the source code, one can create a constant object of class $\mathbb{N}$ with a word of the set

$$K_N = (\{0\} \cup \{1;2;...;9\}\{0;1;...;9\}^*)\{N\} \subset K$$

$K_N{}'$ are words without suffix `N`: $\{0\} \cup \{1;2;...;9\}\{0;1;...;9\}^*$

A number can be infinite (∞) or invalid (Not a Number, or NaN, the Ekio character 251 represents it).

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the number. It also has two `Buleo`s, true if the number is respectively invalid or infinite.

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The represented number of the created object corresponds to the bit sequence (the bits are parsed like binary digits). If its length is 0, the number is invalid.

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid natural number, the number is invalid.

Incrementation `alk : () → ()`

For a number `n` of class `N`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1N)`.

Decrementation `dek : () → ()`

For a number `n` of class `N`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1N)`.

Related functions

Comparisons `/= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a = b$, else false.

`/<  : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a < b$, else false.

`/<= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a \leq b$, else false.

`/>  : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a > b$, else false.

`/>= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a \geq b$, else false.

Addition `/+ : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a - b$.

Multiplication `/× : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a \times b$.

Division `/÷ : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = \left\lfloor \frac{a}{b} \right\rfloor$.

Division remainder `/mod : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a - b\left\lfloor \frac{a}{b} \right\rfloor$.

Special cases

- If a or b are NaN, the comparisons are always false, and the arithmetic operations output NaN;
- The comparisons are also false if both numbers are infinite. Else, if only one number is infinite, the result is intuitive (for example, $\infty > a$ is true for any object a of class `N`);
- Addition: $a + \infty = \infty$, $\infty + b = \infty$
- Subtraction: $a - b$ is NaN if $a < b$ or $b = \infty$. Else, $\infty - b = \infty$;
- Multiplication: $0 \times \infty$ and $\infty \times 0$ are NaN. Else, if a or b are infinite, c is infinite;
- Division: $\infty \div \infty$ and $0 \div 0$ are NaN. Else, $\infty \div b = \infty$ and $a \div 0 = \infty$;
- Division remainder: the output is NaN if $a = \infty$ or $b = 0$. $a \mod \infty = a$.

3.2.5 Z

This class represents the mathematical set $\mathbb{Z}$. The numbers have arbitrary size.

In the source code, one can create a constant object of class $\mathbb{Z}$ with a word of the set

$$K_Z = (\{0\} \cup \{\epsilon, -\}\{1;2;...;9\}\{0;1;...;9\}^*)\{Z\} \subset K$$

$K_Z{}'$ are words without suffix `Z`: $\{0\} \cup \{\epsilon, -\}\{1;2;...;9\}\{0;1;...;9\}^*$.

A number can be infinite (∞) or invalid (Not a Number, or NaN, the Ekio character 251 represents it).

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the number. It also has three `Buleo`s, true if the number is respectively negative, invalid or infinite. The number 0 can only be positive.

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The represented number of the created object corresponds to the bit sequence (the first bit is the sign, the others are parsed like binary digits). If its length is 0 or 1, the number is invalid.

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid integer, the number is invalid.

With a natural number `KONSTRUILO (KONSTANTO N')`

The represented number of the created object corresponds to the number in input.

Incrementation `alk : () → ()`

For a number `n` of class `Z`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1Z)`.

Decrementation `dek : () → ()`

For a number `n` of class `Z`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1Z)`.

Related functions

Comparisons `/= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a = b$, else false.
`/<  : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a < b$, else false.
`/<= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a \leq b$, else false.
`/>  : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a > b$, else false.
`/>= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a \geq b$, else false.

Opposite `/- : (KONSTANTO Z' a) → (Z b)`, with $b = -a$.

Addition `/+ : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a - b$.

Multiplication `/× : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a \times b$.

Division `/÷ : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = \left\lfloor \frac{a}{b} \right\rfloor$.

Division remainder `/mod : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a - b \left\lfloor \frac{a}{b} \right\rfloor$.

Special cases

- If a or b are NaN, the comparisons are always false, and the arithmetic operations output NaN;
- The comparisons are also false if both numbers are infinite with the same sign. Else, if only one number is infinite, the result is intuitive (for example, $+\infty > a$ is true for any object a of class `Z`);
- Addition and subtraction ($a - b$ is $a + (-b)$): $-\infty + \infty$ and $\infty + (-\infty)$ are NaN. Else, $a + (\pm\infty) = \pm\infty$, $\pm\infty + b = \pm\infty$;
- Multiplication: $0 \times (\pm\infty)$ kaj $\pm\infty \times 0$ are NaN. Else, if a or b are infinite, c is infinite with the correct sign;
- Division: $\pm\infty \div (\pm\infty)$ and $a \div 0$ are NaN. Else, if a is infinite, c is infinite with the correct sign, and $a \div \pm\infty = 0$;
- Division remainder: the output is NaN if $a = \pm\infty$ or $b = 0$. $a \mod \pm\infty = a$.

3.2.6 Q

This class represents the mathematical set $\mathbb{Q}$. The numbers have arbitrary size.

In the source code, one can create a constant object of class $\mathbb{Q}$ with a word of the set

$$K_Q = (\{0\} \cup K_Z{}' \cup K_Z{}'\{/\}K_N{}' \cup K_Z{}'\{/\}\{(\}K_Z{}'\{)\})\{Q\} \subset K$$

$K_Q{}'$ are words without suffix `Q`: $\{0\} \cup K_Z{}' \cup K_Z{}'\{/\}K_N{}' \cup K_Z{}'\{/\}\{(\}K_Z{}'\{)\}$.

A number can be infinite (∞) or invalid (Not a Number, or NaN, the Ekio character 251 represents it).

Attributes

The class has two attributes of class `Z`, storing the numerator and denominator. The negative, invalid and infinite properties of the number are determined by these two numbers:

- The number is invalid if the numerator or denominator are invalid, or if both are infinite or the denominator is zero;
- Else, the number is infinite if the numerator is infinite;
- The sign is positive if both number have the same sign, negative else.

Methods

Constructors

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid rational number, the number is invalid.

With an integer `KONSTRUILO (KONSTANTO N')`
`KONSTRUILO (KONSTANTO Z')`

The represented number of the created object corresponds to the number in input.

With numerator and denominator `KONSTRUILO (KONSTANTO Z' a, KONSTANTO Z' b)`

The represented number of the created object is $\frac{a}{b}$.

Incrementation `alk : () → ()`

For a number `n` of class `Q`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1Q)`.

Decrementation `dek : () → ()`

For a number `n` of class `Q`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1Q)`.

Related functions

Comparisons `/= : (KONSTANTO Q' a, KONSTANTO Q' b) → (Buleo)`, true if $a = b$, else false.
`/<  : (KONSTANTO Q' a, KONSTANTO Q' b) → (Buleo)`, true if $a < b$, else false.
`/<= : (KONSTANTO Q' a, KONSTANTO Q' b) → (Buleo)`, true if $a \leq b$, else false.
`/>  : (KONSTANTO Q' a, KONSTANTO Q' b) → (Buleo)`, true if $a > b$, else false.
`/>= : (KONSTANTO Q' a, KONSTANTO Q' b) → (Buleo)`, true if $a \geq b$, else false.

Opposite `/- : (KONSTANTO Q' a) → (Z b)`, with $b = -a$.

Addition `/+ : (KONSTANTO Q' a, KONSTANTO Q' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO Q' a, KONSTANTO Q' b) → (N c)`, with $c = a - b$.

Multiplication `/× : (KONSTANTO Q' a, KONSTANTO Q' b) → (N c)`, with $c = a \times b$.

Division `/÷ : (KONSTANTO Q' a, KONSTANTO Q' b) → (N c)`, with $c = \frac{a}{b}$.

Special cases They are the same as `Z`.

3.3 Core functions

3.3.1 Input/Output functions

Implementations of Progvo must propose basic Input/Output features: a terminal for basic display and input of text, or file manipulations. The following functions allows to use these features.

Input in terminal These functions allow to read and interpret the user input of Ekio strings in a terminal.

Of a boolean `enigi : (Buleo') → ()`
The boolean is true if the input is `VERA` or `1`, false otherwise.

Of a bit sequence `enigi : (Bitsekvenco') → ()`
The input must only contain `0` and `1`. Else, the sequence will be empty.

Of an Ekio string `enigi : (Ekio') → ()`

Of a number `enigi : (N') → ()`
`enigi : (Z') → ()`
`enigi : (Q') → ()`
The input must be valid. Else, the number is NaN.

Display

Of a boolean `printi : (KONSTANTO Buleo') → ()`
This prints `VERA` or `MALVERA`.

Of a bit sequence `printi : (KONSTANTO Bitsekvenco') → ()`
This prints the `Bitsekvenco` with the characters `0` and `1`.

Of an Ekio string `printi : (KONSTANTO Ekio') → ()`

Of a number `printi : (KONSTANTO N') → ()`
`printi : (KONSTANTO Z') → ()`
`printi : (KONSTANTO Q') → ()`

Files

Loading `ŝargi : (KONSTANTO Ekio' path) → (Bitsekvenco)`
`path` is the path of the file. The `Bitsekvenco` is its data.

Writting `skribi : (Bitsekvenco' data, KONSTANTO Ekio' path) → ()`

4. Built-in features

4.1 Presentation

Implementations of Progvo should propose the features in this chapter, if possible.

Their purposes are to allow low-level operations or efficient computations using built-in capabilities.

All classes and functions in the built-in library are in the namespace `Ind` and external. The Progvo implementation must provide the include file `/Bazo/Indi.Pvo`, which contains the declarations of the classes and functions of the built-in library.

4.2 Built-in classes

4.2.1 Adreso

The external class `Adreso` allows to store memory addresses. In other words, the objects of this class are pointers. It allows very low-level operations.

In the code, one can create a constant object with a word of $K_N{}'$.

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bitsequence representing a number, which is the memory address. Its length is constant and must match the memory addresses length.

Methods

If the program is executed in an operating system, read and write to unallocated memory area result on undefined behavior.

Allocation `asigni : (KONSTANTO N' n) → (Adreso)`

Allocates a contiguous memory area, large enough for n bits. The output is the address of the first bit.

Reading `get : (KONSTANTO N' n) → (Bitsekvenco)`

The bit sequence is the n bits at and after the address.

Writting `set : (Bitsekvenco) → ()`

Writes the bit sequence at the address (first bit) and the following ones.

4.2.2 N

This class represents the mathematical set $\mathbb{N}$. The number must be a built-in number of the system (for example, a 64 bits number in an x64 computer), but also depends on the Progvo implementation.

In the source code, one can create a constant object of class $\mathbb{N}$ with a word of the set

$$K_n = (\{0\} \cup \{1; 2; ...; 9\}\{0; 1; ...; 9\}^*)\{n\} \subset K$$

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the number (its length is constant).

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The bit sequence replaces the one of the object. Their length must be the same, else an undefined behavior is caused.

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid natural number or is too big, an undefined behavior is caused.

Incrementation `alk : () → ()`

For a number `n` of class `N`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1n)`.

Decrementation `dek : () → ()`

For a number `n` of class `N`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1n)`.

Related functions

Comparisons `/= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a = b$, else false.

`/<  : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a < b$, else false.

`/<= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a \leq b$, else false.

`/>  : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a > b$, else false.

`/>= : (KONSTANTO N' a, KONSTANTO N' b) → (Buleo)`, true if $a \geq b$, else false.

Addition `/+ : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a - b$.

Multiplication `/× : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a \times b$.

Division `/÷ : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = \left\lfloor \frac{a}{b} \right\rfloor$.

Division remainder `/mod : (KONSTANTO N' a, KONSTANTO N' b) → (N c)`, with $c = a - b \left\lfloor \frac{a}{b} \right\rfloor$.

Special cases The comparisons and operations are built-in and computed by the machine/processor. Situations like overflows or division by zero cause undefined behavior.

4.2.3 Z

This class represents the mathematical set $\mathbb{Z}$. The number must be a built-in number of the system with the same length as `N`.

In the source code, one can create a constant object of class $\mathbb{Z}$ with a word of the set

$$K_z = (\{0\} \cup \{\epsilon, -\}\{1;2;...;9\}\{0;1;...;9\}^*)\{z\} \subset K$$

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the number (its length is constant).

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The bit sequence replaces the one of the object. Their length must be the same, else an undefined behavior is caused.

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid integer or is too big, an undefined behavior is caused.

With a natural number `KONSTRUILO (KONSTANTO N')`

The represented number of the created object corresponds to the number in input.

Incrementation `alk : () → ()`

For a number `n` of class `Z`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1z)`.

Decrementation `dek : () → ()`

For a number `n` of class `Z`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1z)`.

Related functions

Comparisons `/= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a = b$, else false.

`/<  : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a < b$, else false.

`/<= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a \leq b$, else false.

`/>  : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a > b$, else false.

`/>= : (KONSTANTO Z' a, KONSTANTO Z' b) → (Buleo)`, true if $a \geq b$, else false.

Opposite `/- : (KONSTANTO Z' a) → (Z b)`, with $b = -a$.

Addition `/+ : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a - b$.

Multiplication `/×` `: (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a \times b$.

Division `/÷` `: (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = \left\lfloor \frac{a}{b} \right\rfloor$.

Division remainder `/mod : (KONSTANTO Z' a, KONSTANTO Z' b) → (N c)`, with $c = a - b \left\lfloor \frac{a}{b} \right\rfloor$.

Special cases The comparisons and operations are built-in and computed by the machine/processor. Situations like overflows or division by zero cause undefined behavior.

4.2.4 R

This class represents the mathematical set $\mathbb{R}$. The number must be a built-in real number of the system (for example an *IEEE 754 Extended Precision* number), but can also depend on the Progvo implementation.

In the source code, one can create a constant object of class `R` with a word of the set

$$K_r = (K_Z' \cup K_Z'\{.\}K_N')\{r\} \subset K$$

A number can be infinite (∞) or invalid (Not a Number, or NaN, the Ekio character 251 represents it).

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the number (its length is constant).

Methods

Constructors

With a bit sequence `KONSTRUILO (KONSTANTO Bitsekvenco')`

The bit sequence replaces the one of the object. Their length must be the same, else an undefined behavior is caused.

With an Ekio string `KONSTRUILO (KONSTANTO Ekio')`

The represented number of the created object corresponds to the string. If it does not represent a valid real number or is too big, an undefined behavior is caused.

With an integer `KONSTRUILO (KONSTANTO N')`

`KONSTRUILO (KONSTANTO Z')`

The represented number of the created object corresponds to the number in input.

Incrementation `alk : () → ()`

For a number `n` of class `R`, its value after `n.alk()` will be the same as the value after `n ← +(n, 1r)`.

Decrementation `dek : () → ()`

For a number `n` of class `R`, its value after `n.dek()` will be the same as the value after `n ← -(n, 1r)`.

Related functions

Comparisons `/= : (KONSTANTO R' a, KONSTANTO R' b) → (Buleo)`, true if $a = b$, else false.
`/<  : (KONSTANTO R' a, KONSTANTO R' b) → (Buleo)`, true if $a < b$, else false.
`/<= : (KONSTANTO R' a, KONSTANTO R' b) → (Buleo)`, true if $a \leq b$, else false.
`/>  : (KONSTANTO R' a, KONSTANTO R' b) → (Buleo)`, true if $a > b$, else false.
`/>= : (KONSTANTO R' a, KONSTANTO R' b) → (Buleo)`, true if $a \geq b$, else false.

Opposite `/- : (KONSTANTO R' a) → (Z b)`, with $b = -a$.

Addition `/+ : (KONSTANTO R' a, KONSTANTO R' b) → (N c)`, with $c = a + b$.

Subtraction `/- : (KONSTANTO R' a, KONSTANTO R' b) → (N c)`, with $c = a - b$.

Multiplication `/×` `: (KONSTANTO R' a, KONSTANTO R' b) → (N c)`, with $c = a \times b$.

Division `/÷` `: (KONSTANTO R' a, KONSTANTO R' b) → (N c)`, with $c = \frac{a}{b}$.

Special cases They depend on the machine and the Progvo implementation.

4.3 Built-in functions

4.3.1 Input/Output functions

Input in terminal

Of a number `enigi : (N') → ()`
`enigi : (Z') → ()`
`enigi : (R') → ()`
The input must be valid. Else, the number is NaN.

Display

Of an address `printi : (KONSTANTO Adreso') → ()`
This prints the address number.

Of a number `printi : (KONSTANTO N') → ()`
`printi : (KONSTANTO Z') → ()`
`printi : (KONSTANTO R') → ()`

5. Graphics library

Progvo can propose graphical features. Classes and functions of the graphics library are in the namespace `Baz` and can be external. The Progvo implementation would provide the include file `/Bazo/Grafiko.Pvo` containing the declarations and definitions of the classes and functions of the graphics library.

5.1 Classes

The classes will be improved and completed in future versions of Progvo.

5.1.1 Canvas

This class represents canvasses. A canvas is a two-dimensional array of pixels. The definition is not more precise, to allow Progvo implementations to adapt to the system (a canvas could be a window in an operating system with windows, or the full screen in an embedded system with a very basic graphical interface).

Attributes

The class has one attribute of class `Bitsekvenco`, storing the bit sequence representing the data of the pixels (its length is constant).

Methods

Constuctors

With dimensions `KONSTRUILO (KONSTANTO Ind/N' w, KONSTANTO Ind/N' h)`

Creates a canvas of width w and height h (pixels).

Valid positions are horizontally 0-$(w-1)$ and vertically 0-$(h-1)$.

Set a pixel `setBilderon : (KONSTANTO Ind/N' r, KONSTANTO Ind/N' g, KONSTANTO Ind/N' b, KONSTANTO Ind/N' x, KONSTANTO Ind/N' y) → ()`

Changes the color of the pixel at position (x, y). If the position is invalid, an undefined behavior is caused.

Get a pixel `getBilderon : (KONSTANTO Ind/N' x, KONSTANTO Ind/N' y) → (Ind/N r, Ind/N g, Ind/N b)`

Gets the color of the pixel at position (x, y). If the position is invalid, an undefined behavior is caused.

5.2 Widgets

Will come in future versions of Progvo.