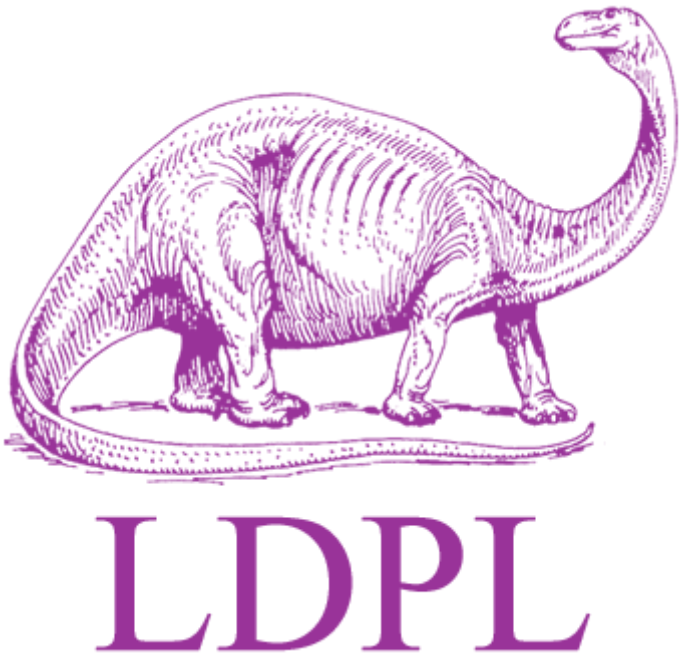
LDPL 19 Standard



Index

About This Document

The LDPL Commitee have compiled this document with the desire to teach and standarize the LDPL programming language. It should be considered a limiting boundary for any LDPL implementation, that should not include any extra commands or definitions nor lack any of the former. If it did, it should be marked accordingly with the names LDPL-ALT, Alternative LDPL or something similar that implies that it's not a standard implementation.

This document is definitive and backwards compatible, and any future revisions that build upon it will only add features to the language, not remove nor change any functionality and, as such, any code written for a previous LDPL specification should continue to run as it did with the specification it was written for.

Feedback, corrections and suggestions are welcome, both on the main LDPL repository or by e-mail to *mdelrio (at) dc.uba.ar*. New commitee members are also welcome.

The Language

LDPL (*Lartu's Definitive Programming Language*) was designed from the beginning to be an excessively expressive programming language, one that can be read from top to bottom without pausing to ponder what symbols mean or what a sentence does. As such, it is mostly written in a series of statements that mimics plain English, with the desire that it can be understood by anybody.

LDPL also aims to suppress unreadable code and redundancy by only having one way of doing anything. What a command does should never overlap with the purpose of another and, as such, every LDPL command does one and only one thing. Every line is a step further towards the completition of an algorithm, no more, no less.

Structure of a LDPL Source Code

LDPL was designed to be a rigidly structured programming language and, as such, variable declarations and the rest of the code procedure are separated in two diferent, mutually exclusive **sections** within a source file. Variable declarations should be placed within the DATA section, while the rest of the code should be placed inside the PROCEDURE section. Further subprocedures should be placed also within the PROCEDURE section, inside their own SUB-PROCEDURE subsection.

Bearing this in mind, the basic skeleton of a LDPL source code will look like this:

```
DATA:

PROCEDURE:
```

The DATA section can be obviated if no variables are declared.

Every statement in LDPL has to be on its own line. Thus, statements are separated by line breaks and it's not possible to have two statements on the same line.

Comments in LDPL are denoted with a hash symbol ('#') and can be placed both on their own line or at the end of a line that already contains a statement. Everything after the hash is considered to be part of the comment and, therefore, not executed nor compiled.

```
DATA: #This is the DATA section

PROCEDURE:
  #This is a comment within the PROCEDURE section!
```

The preferred **file extension** for LDPL source codes is '**.lsc**' (***L**DPL **S**ource **C**ode*). '**.ldpl**' should also be accepted if the later interfered with existing file extensions.

Data Types

LDPL natively supports the **NUMBER** and **TEXT** data types. It also supports **VECTORS** of values of such types.

The **NUMBER** data type, as its name suggests, depicts numeric values. It should be represented internally as a **binary64** double-precision floating-point format number as defined by the **IEEE 754**.

Both variables and numeric constants can be members of the NUMBER type.

The **TEXT** data type, as its name suggests, represents alphanumeric strings. In the interest of supporting as many locales as possible, LDPL should be **utf-8** encoded to be compatible with Unicode. A TEXT maximum length is explicitly not defined and it should be limited only by the amount of available memory on the system.

Both variables and string constants can be members of the TEXT type.

The **VECTOR** data type is not a type itself but a collection of NUMBER or TEXT variables. This implies that only variables can be members of the VECTOR type, as collections of constants (and thus, constant VECTORs) are not present in the language.

VECTORs superficially resemble arrays in other programming languages but with fundamental differences. In LDPL, there's no need to specify the size of a VECTOR before you start to use it. Additionally, any number or string in LDPL may be used as an array index, not just consecutive integers.

VECTORs, as collections of NUMBER or TEXT variables, can only have one defined type at any given time: TEXT *or* NUMBER. A single VECTOR is not capable of storing both numeric and alphanumeric values.

DATA Section

As stated here, LDPL programs are divided in two sections, one of them being the **DATA** section. The DATA section is where variables are declared. In no other part of a LDPL source can variables be declared. If no variables are declared, the DATA section can be obviated.

All variables in LDPL are global (LDPL has no concept of scope) and have a defined data type.

The DATA section is defined and preceded by the `DATA:` keyword, as shown in this example. On every line within the DATA section (that is, on every line after the `DATA:` keyword and before the `PROCEDURE:` keyword) one and only one variable can be declared. The syntax for declaring a variable in LDPL is

```
variable name IS data type
```

Variable names can contain any character except for spaces and colons (':').

Available data types are `NUMBER`, `TEXT`, `NUMBER VECTOR` and `TEXT VECTOR`.

A DATA section cannot contain anything but variable declarations, comments and empty lines. Following the skeleton shown here, the example could be completed further like this:

```
DATA: #This is the DATA section
  myNumber IS NUMBER
  #Next I'm going to declare a text vector
  niceTextVector IS TEXT VECTOR

PROCEDURE:
  #This is a comment within the PROCEDURE section!
```

PROCEDURE Section

As stated here, LDPL programs are divided in two sections, one of them being the DATA section, the other being the **PROCEDURE** section. The PROCEDURE section is where all the code of a LDPL program that is not a variable declaration is written. A LDPL program **must** contain a PROCEDURE section, even if it's empty. Compilation will fail otherwise.

Within the PROCEDURE section, as said here, every line can contain either a comment, a statement, a statement and a comment or be empty. No two statements can be written on the same line.

```
PROCEDURE:
  #A comment
  STORE 5 IN myVariable
  STORE 6 IN myOtherVariable #A statement and a comment
```

Available statements will be explained further in the following sections of this document.

Code within the PROCEDURE section is executed from top to bottom, skipping SUB-PROCEDURE sections, unless explicitly called.

About Statements

LDPL is not a case sensitive language. Variables called `myVar` and `MYVAR` are considered to be the same variable, the same with subprocedure names and statements of any kind.

Usage of Variables

In the following sections you will see excerpts of code like this one:

```
STORE NUMBER-VAR or NUMBER IN NUMBER-VAR
```

Notice the parts in magenta. Parts of procedures colored like that mean that they should be replaced by whatever they say in that color. In the example above, the first replazable

part can be substituted with the name of a variable of NUMBER type or by a NUMBER constant.

Available replacement values are:

- `NUMBER`: A constant of type NUMBER.
- `TEXT`: A constant of type TEXT.
- `NUMBER-VAR`: A variable of type NUMBER.
- `TEXT-VAR`: A variable of type TEXT.
- `SUB-NAME`: A name of a subprocedure.

Usage of Vectors

Vectors in LDPL aren't more than a collection of variables. When you declare a vector, you declare a structure that lets you store something of its type on any subindex of the variable. For example, say you declare the vector `myVector`:

```
DATA:
 myVector IS NUMBER VECTOR
```

Then you can use `myVector` as a multivariable with various indexes where you can store NUMBERs.

```
DATA:
 myVector IS NUMBER VECTOR
PROCEDURE:
 STORE 5 IN myVector:1 #Stores 5 in the subindex 1 of myVector
 STORE -10.2 IN myVector:5 #Stores -10.2 in the subindex 5 of myVector
```

Vector subindexes can't just be constant NUMBERs, though. They can also be NUMBER variables, TEXT and TEXT variables, or even subindexes of other arrays. For example:

```
DATA:
 myVector IS NUMBER VECTOR
 myOtherVector IS NUMBER VECTOR
 myVar IS NUMBER

PROCEDURE:
 STORE 17 IN myVar
 STORE 1 IN myVector:"hello" #Stores 1 in the subindex "hello" of
myVector
 STORE 7 IN myVector:myVar #Stores 7 in the position of index value of
myVar
 STORE 3 IN myVector:myOtherVector:4
 #Stores 7 in the position of index value of myVar of myOtherVector
```

Plase note that as a VECTOR is a collection of variables, a single index of a VECTOR is a variable in itself. This means that a VECTOR with a subindex can be used in any position where you could use a variable of the same type of the vector.

Statements: Control Flow

STORE

- **Syntax:**

  ```
  STORE NUMBER-VAR or NUMBER IN NUMBER-VAR
  ```
  or
  ```
  STORE TEXT-VAR or TEXT IN TEXT-VAR
  ```

- **Description:** assigns a value to a variable.

IF

- **Syntax:**

```
IF NUMBER-VAR or NUMBER IS REL-OP-NUM NUMBER-VAR or NUMBER THEN
 #Code goes here (positive branch)
ELSE
 #Code goes here (negative branch)
END-IF
```

or

```
IF TEXT-VAR or TEXT IS REL-OP-TEXT TEXT-VAR or TEXT THEN
 #Code goes here (positive branch)
ELSE
 #Code goes here (negative branch)
END-IF
```

or

```
IF NUMBER-VAR or NUMBER IS REL-OP-NUM NUMBER-VAR or NUMBER THEN
 #Code goes here (positive branch)
END-IF
```

or

```
IF TEXT-VAR or TEXT IS REL-OP-TEXT TEXT-VAR or TEXT THEN
 #Code goes here (positive branch)
END-IF
```

- **Possible values of `REL-OP-NUM` (number relational operator):**
  - `EQUAL TO`
  - `NOT EQUAL TO`
  - `GREATER THAN`
  - `LESS THAN`
  - `GREATER THAN OR EQUAL TO`
  - `LESS THAN OR EQUAL TO`
- **Possible values of `REL-OP-TEXT` (text relational operator):**
  - `EQUAL TO`
  - `NOT EQUAL TO`
- **Description:** evaluates if the condition given by the relational operator between the first and second values is positive. If it is, the code in the positive branch is executed. If it is not, the code in the negative branch is executed (if available). Execution then continues normally.

WHILE

- **Syntax:**

```
WHILE NUMBER-VAR or NUMBER IS REL-OP-NUM NUMBER-VAR or NUMBER DO
 #Code goes here
REPEAT
```

or

```
WHILE TEXT-VAR or TEXT IS REL-OP-TEXT TEXT-VAR or TEXT THEN
 #Code goes here
REPEAT
```

- **Possible values of `REL-OP-NUM` (number relational operator):**
  - `EQUAL TO`
  - `NOT EQUAL TO`
  - `GREATER THAN`
  - `LESS THAN`
  - `GREATER THAN OR EQUAL TO`
  - `LESS THAN OR EQUAL TO`
- **Possible values of `REL-OP-TEXT` (text relational operator):**
  - `EQUAL TO`
  - `NOT EQUAL TO`

- **Description:** evaluates if the condition given by the relational operator between the first and second values is positive. While it is, the code between the `WHILE` and `REPEAT` statements is repeatedly ran.

SUB-PROCEDURE

- **Syntax:**

```
SUB-PROCEDURE procedure name
 #Code goes here
RETURN
```

- **Description:** A SUB-PROCEDURE is a piece of code that can be called and executed from other parts of the script. SUB-PROCEDURE subsections must be declared within the PROCEDURE section of the code and end with at least one RETURN keyword. Bear in mind that you can't define a SUB-PROCEDURE within a SUB-PROCEDURE.

CALL SUB-PROCEDURE

- **Syntax:**

```
CALL SUB-PROCEDURE procedure name
```

- **Description:** Executes a SUB-PROCEDURE. Once the SUB-PROCEDURE returns, the execution continues from the line following the `CALL SUB-PROCEDURE`.

RETURN

- **Syntax:**

```
RETURN
```

- **Description:** Returns from a SUBPROCEDURE. Will give a compiler error if used outside one.

Statements: Arithmetic Operations

ADD

- **Syntax:**

```
ADD NUMBER-VAR or NUMBER AND NUMBER-VAR or NUMBER IN NUMBER-VAR
```

- **Description:** adds two NUMBER values and stores the result in a NUMBER variable.

SUBTRACT

- **Syntax:**

```
SUBTRACT NUMBER-VAR or NUMBER FROM NUMBER-VAR or NUMBER IN NUMBER-
VAR
```

- **Description:** subtracts two NUMBER values and stores the result in a NUMBER variable.

MULTIPLY

- **Syntax:**

```
MULTIPLY NUMBER-VAR or NUMBER BY NUMBER-VAR or NUMBER IN NUMBER-VAR
```

- **Description:** multiplies two NUMBER values and stores the result in a NUMBER variable.

DIVIDE

- **Syntax:**

```
DIVIDE NUMBER-VAR or NUMBER BY NUMBER-VAR or NUMBER IN NUMBER-VAR
```

- **Description:** divides two NUMBER values and stores the result in a NUMBER variable.

MODULO

- **Syntax:**

```
MODULO NUMBER-VAR or NUMBER BY NUMBER-VAR or NUMBER IN NUMBER-VAR
```

- **Description:** calculates the remainder of the modulo operation between two NUMBER values and stores the result in a NUMBER variable.

ABS

- **Syntax:**

```
ABS NUMBER-VAR
```

- **Description:** calculates the absolute value of a NUMBER variable and stores it in that same variable.

Statements: Text Operations

JOIN

- **Syntax:**

```
JOIN NUMBER-VAR or NUMBER or TEXT-VAR or TEXT AND NUMBER-VAR or
NUMBER or TEXT-VAR or TEXT IN TEXT-VAR
```

- **Description:** concatenates two values and stores them in a TEXT variable. If any value is a number, it is converted to a string before concatenation.

GET CHARACTER AT

- **Syntax:**

```
GET CHARACTER AT NUMBER-VAR or NUMBER FROM TEXT-VAR or TEXT IN
TEXT-VAR
```

- **Description:** gets the character at the position indicated by the NUMBER value from the TEXT value and stores it in a TEXT variable.

Statements: Input / Output

DISPLAY

- **Syntax:**

```
DISPLAY multiple NUMBER, TEXT, TEXT-VAR, NUMBER-VAR or CRLF
```

- **Description:** outputs the values passed to the output stream. `CRLF` means line break.
- **Example:**

```
DISPLAY "Hello World! " myVariable CRLF
```

ACCEPT

- **Syntax:**

```
ACCEPT TEXT-VAR or NUMBER-VAR
```

- **Description:** the ACCEPT command is used to gather input from the user. If a TEXT variable is specified, anything the user enters before pressing the 'return' key will be accepted. If a NUMBER variable is specified, the user must enter a number (if any non-numeric key is entered, the error message "Redo from start" will be output and the ACCEPT command rerun).

EXECUTE

- **Syntax:**

```
EXECUTE multiple NUMBER, TEXT, TEXT-VAR or NUMBER-VAR
```

- **Description:** concatenates all the passed values as a single command and executes the specified command, dumping any resulting text to the output stream.
- **Example:**

```
EXECUTE "echo " myVariable " >> myFile"
```

Language Grammar

The grammar is listed below in Backus-Naur form. In the listing, an asterisk ("*") denotes zero or more of the object to its left; parentheses group objects; and an epsilon ("ε") signifies the empty set. As is common in computer language grammar notation, the vertical bar ("|") distinguishes alternatives, as does their being listed on separate lines. The symbol "CRLF" denotes a carriage return (usually generated by a keyboard's "Enter" key).

```
# DATA TYPES
digit ::= 1 | 2 | ... | 9 | 0
letter ::= A | B | C | ... | a | b | c | ... | á | ñ | あ | ... | 仮 |
... | ر | ...
procedure-name ::= letter letter*
string ::= " letter* "
number ::= (ε | - ) digit digit* (ε | (. digit digit*))

# VARIABLE DEFINITIONS
num-var-name ::= letter letter*
str-var-name ::= letter letter*
var ::= num-var | str-var
```

```
num-var-definition ::= num-var-name IS NUMBER (VECTOR | ε)
str-var-definition ::= str-var-name IS TEXT (VECTOR | ε)
num-var ::= num-var-name (: (number | num-var | string | str-var))*
str-var ::= str-var-name (: (number | num-var | string | str-var))*
var-definition ::= num-var-definition | str-var-definition

# STATEMENTS
block ::= (statement \n)*
statement ::= display | accept | join | store | while | if | addition |
subtraction | multiplication | division | modulo | abs | subroutine-call
| get-char | sys-exec
display ::= DISPLAY (var | string | number | CRLF) (var | string |
number | CRLF)*
accept ::= ACCEPT var
join ::= JOIN (var | string | number) AND (var | string | number) INTO
str-var
str-store ::= STORE (str-var | string) IN str-var
num-store ::= STORE (num-var | number) IN num-var
store ::= str-store | num-store
num-while ::= WHILE (num-var | number) (relop1 | relop2) (num-var |
number) DO block REPEAT
str-while ::= WHILE (str-var | string) relop1 (str-var | string) DO
block REPEAT
while ::= num-while | str-while
num-if ::= IF (num-var | number) (relop1 | relop2) (num-var | number)
THEN block ((ELSE block) | ε) END-IF
str-if ::= IF (str-var | string) relop1 (str-var | string) THEN block
((ELSE block) | ε) END-IF
if ::= num-if | str-if
addition ::= ADD (num-var | number) AND (num-var | number) IN (num-var)
subtraction ::= SUBTRACT (num-var | number) FROM (num-var | number) IN
(num-var)
multiplication ::= MULTIPLY (num-var | number) BY (num-var | number) IN
(num-var)
division ::= DIVIDE (num-var | number) BY (num-var | number) IN (num-
var)
modulo ::= MODULO (num-var | number) BY (num-var | number) IN (num-var)
abs ::= ABS num-var
subroutine-call ::= CALL SUB-PROCEDURE procedure-name
get-char ::= GET CHARACTER AT (num-var | number) FROM (string | str-var)
IN str-var
sys-exec ::= EXECUTE (string | str-var)

# RELATIONAL OPERATORS
relop1 ::= (IS EQUAL TO) | (IS NOT EQUAL TO)
relop2 ::= (IS GREATER THAN) | (IS LESS THAN) | (IS GREATER THAN OR
EQUAL TO) | (IS LESS THAN OR EQUAL TO)

# CODE SECTIONS
data-section ::= DATA: \n (var-definition \n)*
procedure-section ::= PROCEDURE: block
sub-procedure-section ::= SUB-PROCEDURE procedure-name: block RETURN

# VALID CODE
valid-code ::= (data-section | ε) procedure-section sub-procedure-
section*
```
