PAL : PDF Automation Language

Language Reference Manual

Anshuman Singh as4916@columbia.edu
Diksha Vanvari dhv2108@columbia.edu
Vinay Gaba vhg2105@columbia.edu
Viral Shah vrs2119@columbia.edu

1. Introduction

2. Types
   2.1. Primitive Data Types
      2.1.1. boolean
      2.1.2. integer
      2.1.3. float
      2.1.4. string
      2.1.5. pdf
      2.1.6. page
      2.1.7. line
   2.2. Predefined Constructs
      2.2.1. Tuple

3. Lexical Conventions
   3.1. Identifiers
   3.2. Keywords
   3.3. Literals
   3.4. Punctuation
   3.5. Comments

4. Expressions and Operators
   4.1. Arithmetic Operators
   4.2. Logical and Relational Operators
   4.3. List Operators
   4.4. String Operators
   4.5. PDF Operators

5. Statements and Control Flow
   5.1. Declaration Statements
   5.2. Expression Statements
   5.3. Control Flow Statements

6. Built-in Functions
6.1. renderpdf
6.2. getpages
6.3. I/O
6.3.1. print
6.3.2. scan
6.4. sizeof

7. Program Structure
7.1. Import statements
7.2. Function Definition and Declaration
7.3. Function Call
7.4. Scoping
7.5. Order of operations

8. Standard Library
8.1. High Level Constructs
8.1.1. paragraph
8.1.2. image
8.2. Collections
8.2.1. Map
8.2.2. List
8.3. PDF Manipulation Functions
8.3.1. split
8.3.2. watermark
8.3.3. protect
8.3.4. chart
8.4. File I/O Functions
8.4.1. load
8.4.2. loadpdf
8.4.3. loadcsv
8.4.4. loadimage
9. References

1. Introduction:

Portable Document Format (PDF) is the file standard for the electronic exchange of documents. According to estimates by Adobe executives, there might be up to 2.5 trillion PDF documents existing in the world. The reason for its popularity is its platform-agnostic behaviour of passing and sending information that won't be skewed or altered. Our aim is to expand the range of operations performed on this popular data source through the means of PAL. There are many solutions available which are similar in nature to PAL but very often they do not fulfill the exact functionality as needed and are generally complicated, which requires a learning curve. We intend to simplify these interactions with PAL while at the same time also enable powerful operations which can fulfill operational needs.

1.1. *What doe a PAL program look like ?*

A PAL program consists of a list of statements. We will be describing this in more detail in the sections below.

2. Types

There are seven primitive types in PAL which are explained in more detail as follows:

2.1. Primitive Data Types

2.1.1. boolean (bool)

Maybe **true** or **false**. boolean types can only be used with other boolean types, any other operation involving boolean types fails.

```
boolvar : bool = true
```

2.1.2. integer (int)

An integer literal such as 5664 is a 32-bit signed integer. It takes values in the range from -2,147,483,648 to 2,147,483,647

```
intvar : int = 42;
```

2.1.3. float (float)

A float literal has an integer part followed by a fraction part. It is a 64-bit signed float.

```
floatvar : float = 42.0;
```

2.1.4. string (string)

A string literal is a sequence of ASCII characters. They are enclosed in double quotes, with special characters escaped with a backslash.

```
stringvar1 : string = "This is a string."
stringvar2 : string = "This is \"Hello\" from the other
side.
```

Escape sequences are also supported in the following manner:

\n	newline	\t	horizontal tab

2.1.5. pdf (pdf)

A pdf type represents a logical representation of a physical "PDF" document.

```
pdfvar : pdf;
```

2.1.6. page (page)

A page type represents a logical representation of a physical "PDF" page. A pdf document consists of 0 to an arbitrary number of lines.

```
pagevar : page;
```

2.1.7. line (line)

A line type represents the lowest level of physical space which is used to draw strings on a page. Line accepts a string along with fonts styles, sizes as strings and left and right margin as integers. Based on the values of input parameters and the pdf configurations, the line variable stores an index which points to the last position of the string which has been written out to the pdf.

```
line1 : line(string, string , string , int, int)
```

2.1.8. null

A null type indicates that an identifier is not initialized.

2.2. Predefined Constructs

2.2.1. Tuple

A tuple represents the association of a pdf with a page. Before using a pdf and a page as part of a tuple, they need to be defined and the page needs to be added to the pdf, otherwise the construct gives an error.

```
pdfvar : pdf;
pagevar : page;
pdfvar += pagevar;
```

The tuple: {pdf,page}

3. Lexical Conventions

3.1. *Identifiers*

Identifiers are sequences of letters, digits and underscores. All the identifiers must begin with a letter and not use any reserved keywords.

Valid variable names: isuzu bb8 r2d2
Invalid variable names: 6valid

3.2. *Keywords*

The following identifiers are used as keywords and cannot be used in any other manner.

main	bool	int
float	string	pdf
page	line	if
elif	else	for
while	return	true
false	import	split
paragraph	list	watermark
protect	loadText	loadImage
loadPDF	loadCSV	getpages
void	null	return
break	continue	

3.3. *Punctuation*

These are special characters which are neither operators nor identifiers. They have their own significance.

: -> type declarator
, -> map key-value separator
“” -> string literal delimiter
; -> end of a statement

3.4. *Literals*

A literal is a notation for representing a fixed value in source code.

3.4.1. Integer Literal

A positive integer is 1 or more digits from 0 - 9. A negative integer is a '-' followed by 1 or more digits from 0 - 9. Zero is neither positive nor negative and is represented as 0.

INT = "['0'-'9']+ | '-'['1'-'9']['0'-'9']+"

3.4.2. Float Literal

A float literal consists of an integer part, followed by a point '.' followed by the fractional part. The float literal can either be positive or negative.

FLOAT = ['+''-']?['0' - '9']* '.' ['0'-'9']*

3.4.3. Boolean Literal

A boolean literal can take only two values - true or false.

BOOL = "true|false"

3.4.4. String Literal

A string literal is zero or more ASCII characters written between two double quotes. \n, \r, \t, ", ', \ are preceded with an escape sequence character '\'.

STRING = "\"(([' '-'!' '#'-'[' ']'-'~'] | '\\' ['\\' '\"' 'n' 'r' 't']))*\""

3.5. *Comments*

PAL only support single line comments which are identified by **#.**

```
# This is a single line comment.
```

4. Expressions and Operators

4.1. *Expressions*

An expression in PAL could be any one of the following:

4.1.1. A unary operator followed by an operand
4.1.2. An operand followed by a binary operator followed by an operand
4.1.3. Declaration of a type
4.1.4. Assigning a value to a type
4.1.5. Declaration and initialization of objects
4.1.6. Any one of the four literals supported by PAL
4.1.7. An identifier of a type
4.1.8. Call to a function that returns a value

4.2. *Arithmetic Operators*

An operator is a token that would manipulate the value of operands(s).

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

The precedence of the operators is given below:

* / %
+ -

4.3. *Logical and Relational Operators*

A relational operator is used to determine how two operands compare to each other. It can be used to test for equality, inequality, and comparison of operand values.
A logical operator is used to perform logical operations on operands.

Operator	Operation
==	Equality
!=	Inequality
>	Greater Than
>=	Greater Than or Equals
<	Lesser Than
<=	Lesser Than or Equals
!	Logical NOT
&&	Logical AND
||	Logical OR

4.4. *String Operators*

PAL supports the following operations on the primitive type 'string'.

1. The '-' operator takes a string and an int as input and returns the substring of this string from position 'int' to the last index of the string.

```
str1 = str2 - 10;
```

2. The '.' operator takes two strings as its operands and returns a new string that is the concatenation of these two strings.

```
str3 = str1 + str2;
```

4.5. *PDF Operators*

PAL provides operators for performing basic PDF manipulation operations.

1. You can add a page to an existing pdf using the `+=` operator. This operator modifies an existing pdf adding the page to the end of its list of pages.

```
pdf1 += page1;

#You can also add a line to a tuple by using the += operator.
{pdf1, page1} += line1;
```

The above expression writes out a line to page1 of pdf1. The amount of data that can be written out on a line depends on the font, font size, the page size and margins and is determined at the time of writing the line to the page. This expression also sets the last_index attribute of line to the last position of the input string that could fit on a line on the pdf.

2. You can add a page to a pdf to create a new pdf using the '.' operator.

```
pdf2 = pdf1 . page1;
```

The same operator can also be used to concatenate two pdfs and store the result in a new pdf.

```
pdf3 = pdf1 . pdf2;
```

3. The '-' operator can be used to remove a page from a pdf and store the result in the a new pdf. This operator takes a pdf and an int (page number) as its operands.

```
pdf2 = pdf1 - 2;
```

4. You can use the '<>' operator to swap two pages of a pdf. The operator takes two ints (page numbers) as its operands and returns a pdf.

```
pdf1 = 4 <> 9;
```

The above expression swaps pages 4 and 9 of pdf1.

5. The **::** operator is a unary operator that operates on a line. After the line is given a string and written out to a pdf, the expression 'line**::**' returns the index of the last position that could be written out in a single line on the pdf.

5. Statements and Control Flow

5.1. Declaration Statements

A declaration statement specifies the name and datatype of the variable being declared. In addition, it may also initialize the variable.

```
pagevar : page;

pagevar : page = getFirstPage(pdf1);
```

5.2. Expression Statements

The expression statement comprises of an expression, terminated by a semicolon. An expression statement is evaluated from left to right.

```
stringvar : string = str - 15;

pdfvar : pdf = mergePDFs(pdf1, pdf2);

integervar : int = 14 % 4;
```

5.3. Control Flow Statements

Conditional execution of partial blocks of code is enabled using control flow statements that facilitate decision making, looping and branching. The decision making statements include the if, else, elif statements. The looping statements include the for and while statements. The branching statements include the break, continue and return statements.

5.3.1. If, Else, Elif

These statements enable conditional execution of partial blocks of code by evaluating the given condition and executing the corresponding block of code. If the condition is true, then it executes the statements in the first block as limited by the parentheses, else it executes the statements in the second block as limited by the parentheses.

```
if (<condition1>) {
      <statement1>
      <statement2>
} elif (<condition2>) {
      <statement3>
      <statement4>
} else {
      <statement5>
      <statement6>
}
```

5.3.2. For, While

These statements enable conditional execution of partial blocks of code by evaluating an expression against a given condition, and executing the corresponding block of code if the condition is true.

For Loop
While the condition mentioned by the second expression is true, the loop continues iterations, each time executing the statements in the block following ‘do’ limited by the ‘{}’ parentheses.

```
n : int = 10;
for (i : int = 1; i <= 10; i++) {
      <statement1>
      <statement2>
}
```

While Loop
While the condition mentioned by the expression is true, the loop continues iterations, each time executing the statements in the block following ‘do’ limited by the ‘{}’ parentheses.

```
i : int = 1;
n : int = 10;
while (i != n) {
      <statement1>
      i++;
}
```

5.3.3. Break, Continue, Return

These statements enable conditional execution of partial blocks of code by specifying the termination of execution of the corresponding block of code.

Break

A break statement within a 'for' or 'while' statement terminates the looping of the innermost looping statement it is nested within.

```
i : int = 1;
n : int = 10;
while (i != n) {
      <statement1>
      if (<condition1>) {
            <statement2>
            break;
      }
      i++;
}
```

Continue

A continue statement within a 'for' or 'while' statement skips the current iteration of the innermost looping statement it is nested within, skipping to the end of it and evaluating the conditional expression that controls the loop.

```
i : int = 1;
n : int = 10;
while (i != n) {
      <statement1>
      if (<condition1>) {
            <statement2>
            continue;
      }
      i++;
}
```

Return

A continue statement exits from the current method and the control flow returns to the point of function invocation. The return statement is followed by a return value of the type indicated in the function definition.

```
getResult (parameter : parametertype) : returntype {
      <statement1>
      <statement2>

      return result;
}
```

6. Built-in Functions

6.1. *renderpdf*

The renderpdf function takes in two arguments, a pdf and a disk location and saves this pdf to the specified location.

Input parameters:
pdf -- the pdf that you want to save
string -- the disk location you want to save the pdf to

Return Type:
void -- the function returns void

6.2. *getpages()*

The getpages function takes a pdf as input and returns a list of pages from the pdf.

Input parameters:
pdf -- the pdf from which you want to extract pages

Return Type:
list page -- the function returns a list of pages of the pdf

6.3. *I/O*

6.3.1. print

The print function accepts a type as input and prints out the string representation of the type.

Input parameters:
string -- the value that needs to be printed to the output stream

Return Type:
void -- the function returns void

```
#Prints "ABC" on the output stream.
print "ABC";
```

6.3.2. scan

The scan function accepts a string literal from the input stream.

Input parameters:
string -- the string literal to be read from input stream

Return Type:
void -- the function returns void

```
#Reads a string literal from the input stream.
scan s : string;
```

6.4. *sizeof*

The sizeof function is an overloaded function that takes either a string, list or a map as input and returns the number of characters in the string, number of elements in the list or the number of key value pairs in the map respectively.

Input parameters:
string -- The string whose length needs to be returned
| map -- The map whose size needs to be returned
| list -- The list whose size needs to be returned

Return Type:
int -- The size of the datatype returned as an integer

7. Program Structure

7.1. *Import statements*

These statements are used to import the libraries that the language can use. If there are any import statements, then they must appear at the beginning of program. It makes all the functions and fields from the module accessible in the current program, and can be accessed without prepending the module name. The statements are included as follows:

```
import stdlib
listvar : list;
```

7.2. *Function Definition and Declaration*

Users can define their own function in PAL. A function is declared and defined at the same time. A function is defined by specifying the function name followed by the function parameters and finally the function return type. Functions are declared by specifying a list of statements after the function definition. Functions don't necessarily need to be defined before they are used.

```
function_name (parameter : parametertype) : returntype {
      <statement1>
      <statement2>

      return result;
}
```

7.3. *Function Call*

Functions are called using the following syntax. Users can specify the function name and the list of input parameters. Once a function is called, the program execution is halted until the function execution is completed.

```
variable : type;
variable = function_name (parameter : parametertype);
```

7.4. *Scoping*

Scope refers to the variables and functions available at any given instance in the program. All variables are local and are available within the function in which they have been declared. In addition, a variable declared within a given block of code is available only within the scope of that block. The scope of a block is limited by the surrounding parentheses. Thus, variables declared within a control flow statement is available only within the scope of the control flow statement, as opposed to variables declared in the beginning of a function definition, which are available throughout the function body. If more than one variable is declared with the same identifier, then the variable declared in the most nested scope, limited by parentheses, prevails the variable declared before it, within the scope.

A function is available throughout the file in which it has been defined and can be invoked by another function irrespective of the order of function definition.

Every program must have a main function. The program starts execution from the main function. On successful execution, the main function returns an integer value as specified in the return statement, else it returns -1.

```
main() : int {
	<statement1>

	result : int = getResult(<expression>);

	n : int = 10;
	while (result != n) {
		<statement1>
		result++;
	}

	return 0;
}
```

```
getResult (parameter : int) : int {
      <statement1>
      <statement2>

      return result;
}
```

7.5. *Order of operations*

In an expression containing multiple operators, the evaluation is based on the following order of operator precedence:

- Function Invocations
- Unary Operators nested from right to left
- Multiplication, Division, Modulo
- Addition, Subtraction
- Relational Operators
- Equality and Inequality Operators
- Logical Operators
- Assignments nested from right to left

8. Standard Library

The standard library consists of collections, high level constructs and pre-defined functions which help in simplifying common operations for the user of this language.

8.1. *High Level Constructs*

8.1.1. paragraph

A paragraph is a high level construct which prints a given string to a pdf. This takes care of printing the entire string, which is different from the line construct which prints only one line at a time.

Input parameters:
string -- The string to be printed out to the pdf
string -- The font style to be used
string -- The font size to be used
int -- left margin
int -- right margin

Return Type:
paragraph -- The paragraph type that represent the string input

The implementation of paragraph would be similar to:

```
str : string;

while(str != null){
      line1: line(str, "Helvetica", "12", 5, 5);
      {pdf,page}+=line1;
      str = str - line1::;
}
```

8.1.2. image

An image is a high level construct that lets you hold a representation of a jpeg/png image. It will also be helpful in generation of charts that would also be returning an image of the chart which can then be placed in the pdf.

```
#Initialize an image
imgVar2: image;

#Initialize an image with specific height and width
imgVar2: image(int,int);

#Loading an image
imgVar = loadImage("imagePath");

#Adding an image to a page
{pdf1, page1} += imgVar;
```

8.2. *Collections*

PAL supports two collection data types that provide an architecture to store and manipulate a group of data types supported by PAL.

8.2.1. Map

The Map datatype takes a key value pair as an input and supports all the common functionalities that a map is expected to support. It cannot contain duplicate keys and each key can only contain one value.

```
#Initialization
mapvar : map keyType,valueType;

#Adding a Key Value pair to a Map
mapvar += key,value;
```

```
#Removing a Key from a Map
mapvar = mapvar - key;
```

8.2.2. List

The List datatype is used to store an ordered collection of datatypes. It allows storing duplicate values as well.

```
#Initialization
listvar : list listType;

#Adding a List Element to a List
listvar += listElement;

#Removing a List Element from the List
listVar = listVar - index;

#Accessing an element in the List
```

8.3. PDF Manipulation Functions

PDF Manipulation Functions are a set of predefined functions that help in simplifying common PDF Manipulation tasks.

8.3.1. split

The split function helps in splitting a pdf file into multiple pdf's.

```
#Usage
pdfList = split(pdf,listVar);
```

Input parameters:
pdf -- The pdf file that needs to be split
list -- A list of integers that specify which page numbers to split the pdf on

Return Type:
list -- A list of pdf files that were split from the original pdf

8.3.2. watermark

The watermark function helps in adding a watermark to all the pages of the pdf file.

```
#Usage
pdfVar = watermark(pdf,filepath);
```

Input parameters:
pdf -- The pdf file on which the watermark will be added
string -- The filepath of the image which will serve as the watermark

Return Type:
pdf -- The pdf file which has the watermark overlayed on it

8.3.3. protect

The protect function helps in password protecting the pdf file.

```
#Usage
pdfVar = protect(pdf,password);
```

Input parameters:
pdf -- The pdf file on which needs to be password protected.
string -- The password that will be protecting the pdf file.

Return Type:
pdf -- The pdf file which has been password protected.

8.3.4. chart

The chart function helps in creating compelling charts. The charts supported right now are 'Bar-Chart' and 'Pie-Chart'.

```
#Usage
imagevar = chart(map,list);
```

Input parameters:
map -- The key value pairs required for creating the chart such as label names, legend information etc.
list -- The data that will be used to populate the chart

Return Type:
image -- The image representation of the chart created

The map variable consists of the following attributes that can be passed as key value pairs using a map:

Chart type
Chart title
X-Axis label
Y-Axis label
Include legend

8.4. *File I/O Functions*

File I/O Functions are a set of predefined functions that help in simplifying common file loading tasks.

8.4.1. load

The load function helps in loading a text file into the program.

```
#Usage
str : string;
str = load("Area51.txt");
```

Input parameters:
string -- location of the file

Return Type:
string -- string representation of text

8.4.2. loadPDF

The loadPDF function helps in loading a PDF file into the program.

```
#Usage
pdfvar : pdf;
pdfvar = loadPDF("Area52.pdf");
```

Input parameters:
string -- location of the file

Return Type:
string -- pdf object of file

8.4.3. loadCSV

The loadCSV function helps in loading a CSV file into the program.

```
#Usage
listvar : list list string;
listvar = loadCSV("Area53.csv");
```

Input parameters:
string -- location of the file

Return Type:
list -- list representation of values in the file

8.4.4. loadImage

The loadImage function helps in loading a JPG file into the program.

```
#Usage
imagevar : image;
imagevar = loadImage("Area54.jpg");
```

Input parameters:
string -- location of the file

Return Type:
image -- image representation of file

9. References

[1] Ritchie, Dennis M. https://www.bell-labs.com/usr/dmr/www/cman.pdf, C Reference Manual
[2] https://docs.oracle.com/javase/specs/jls/se8/html/index.html The Java Language Specification.
[3] Edwards, Stephen. "Programming Language and Translators." Lecture