Red-Pandas: Language Reference Manual

Amina Assal (aa4290), Ivan Barral (iab2131), Rafail Khalilov (rk2960), Myric Lehner (mhl2157)

1 Introduction

Red-Pandas is a programming language that recreates some of Numpy's core functionality natively so that compiled code can quickly churn through linear algebra problems, which could be useful in applications of robotics, data compression, and building neural networks in machine learning. Users can write programs to solve systems of equations and manipulate data in matrices. The following manual outlines lexical conventions of the language, syntax, scope, built-in functions, as well as the grammar of the language.

2 Lexical Conventions

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The identifier starts with a letter or underscore and could be followed by letters, numbers, or underscores. These identifiers are case sensitive.

```
foo;           /* correct */
fo0;           /* correct */
0of;           /* not a proper identifier*/
f_o;           /* correct */
_foo;          /* correct */
foo != Foo;
```

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

2.3.1 Keyword Identifiers

```
int                                    break
double                                 continue
string                                 if
matrix                                 else
def                                    while
return                                 for
print                                  NULL
bool                                   void
map                                    main
true                                   false
```

2.3.2 Keyword Characters

```
=     /* value binding */              >     /* greater than operator */
```

```
!     /* NOT operator */              <     /* less than operator */
&     /* AND operator */              |     /* absolute value */
(    )                                [    ]
{    }                                +     /* addition operator */
-     /* subtraction operator */      ;     /* statement separator*/
^     /* power operator */            *     /* multiplication operator */
,     /* COMMA */
```

2.4 Constants

There are several kinds of constants supported by Red-Pandas, as follows:

2.4.1 Integer constants

An integer constant is a sequence of one or more decimal digits, optionally preceded by a negative value. The type for *int* is 32-bit.

2.4.2 Double constants

A double constant consists of an integer part, a fractional part, and an exponent part. The integer part is a sequence of one or more digits, optionally preceded by a minus sign (-). The fractional part starts with a decimal point and continues with one or more digits. The exponent part starts with an optional plus or minus sign (+ || -) and is followed by one or more digits. This represents a power of 10 in scientific notation. Either the exponent part or fractional part can be excluded but not both. The type for *double* is 64-bit.

2.4.3 String constants

A string constant is an immutable sequence of one or more characters

2.4.4 Boolean constants

Booleans have two predefined constants for each truth value. *true* or *false*

```
true   false
```

2.5 Punctuation

Using the previously defined identifiers and constants, Red-Pandas can create expressions.

2.5.1 Parentheses

We can use parentheses to indicate a list of arguments in a function declaration, a function call, or a function variable declaration

2.5.2 Brackets

Curly brackets are used to define blocks of code such as those found in the body of a function or after an if/else statement.

```
def int foo(arg1, arg2){
   /* block */
}

foo(arg1,arg2);  /* function call */
```

Square brackets are used to access and initialize matrices. In this matrix, commas are used to separate the different elements of each row and column

```
matrix int foo = [ [ 0,1],              /* creates matrix foo */
                   [ 1,0] ];
foo[0][1];                              /* accesses the 2nd element in the
                                           1st row and returns value of 1 */
```

2.5.3 Semicolon and White Space

The semicolon is used to separate statements. White space is defined by the space, tab, newline, vertical tab, and form-feed characters. The use of semicolons and other separators (brackets, commas, parentheses) means that white space is ignored in Red-Pandas, except to separate tokens.

```
matrix int foo = [ [ 0,1],
               [1,0] ];
matrix int foo2=[[0,1],[1,0]];
foo == foo2;                             /* foo and foo2 are the same */

/* the following line is equivalent to the code above */

matrix int foo=[[0,1],[1,0]];matrix int foo2=[[0,1],[1,0]];foo==foo2;

matrixfoo [[0,1],[1,0]];                /* throws ERROR matrixfoo type NOT
                                            defined, need white space here */
```

3 Syntax

3.1 Variable Declarations

```
matrix-opt var-type new-id-list semicolon
```

Var-type here is one of the primitive types or a declared function type. Matrix-opt is the optional declaration for a matrix. New-id-list is a list of one or more new (or new to scope) identifier tokens, separated by commas.

3.2 Function Declarations

Functions are defined this way:

```
def matrix-opt return-type new-id (var-type list-opt) block
```

3.3 Statements

3.3.1 Assignments

```
var-obj = actual-list semicolon
```

In an assignment statement, var-obj is defined as the rvalues from actual-list. A var-obj must be either an ID token, or a validly defined matrix location pointed to by an ID token.

The rvalues - either literals or function returns - must agree with the var-obj's declared type.

3.3.2 Blocks and Control Flow

- A block is a curly braces-bounded set of zero or more statements.
- If / else
    - An If statement consists of a conditional expression followed by statements. If the expression evaluates to true, the statements are evaluated.
    - If an Else statement is present and the If statement evaluates to false, statements following the Else statement will be evaluated.

- While
    - Similar to an If statement, While statement is a conditional expression followed by a series of one or more statements. These statements are evaluated in total and in repeating sequence for as long as the expression evaluates to true.
- For
    - Similarly to C / Java / etc, a For loop in Red Pandas has an expression that includes a conditional and a series of statements that are evaluated as long as the condition is evaluated as true. Unlike a While, a For loop's calling expression includes its own user defined iteration function.

- Break / Continue
    - Break
        - A break statement exits any For or While loop that it occurs within. Specifically, it exits the loop level that contains the break statement.
    - Continue
        - A continue statement terminates the execution of the current sequence of statements within its containing loop and moves directly to the next evaluation of the loop's conditional.

4 Scoping

Red-Pandas uses lexical scoping to define the scope of a variable. Depending on the block within which the variable is declared, no other block has access to a variable that isn't within the scope of the block. References from lower scope level to variables that are initialized in the higher scope level are allowed, however. If there are variables with the same identifiers, then it overrides the variable that is in a higher scope.

4.1 Scope and Brackets

Curly brackets are used to define the blocks found in the body of a function, as well as nested blocks within the body of a function. Nested blocks include blocks defined by if/else, for, and while statements. Variables within a block will override any other variables in a higher scope that share the same identifier if the variable is assigned a new value. However, if the variable is declared explicitly within a smaller scope block, and there are variables that share the same identity outside of the block, the two variables will have different values, depending on what they are assigned, and each will be treated as a separate variable.

```
int i = 5;   /* i = 5 */
int k = 2;   /* i = 2 */

def void foo(){
    int i = 3;   /* new variable, i = 3 */
    int k = 20;  /* new variable, k = 20 */
    while (i<8){
        i++;    /* this will loop until i = 7 when foo() is executed */
    }
}

/* i = 5 * /

def int boo(){
    k = 10;        /* overrides the global variable  */
    foo();

    int j = k+i;   /* new variable, scope is function's body */
    return j;      /* returns 15 */
}

int var = boo();
/* i = 5, k = 10, new variable var = 15 */


def void goo(){
    var = j + i;   /* compiler will reject, j is not defined */
}
```

4.2 Race Conditions

Since Red-Pandas overrides any variables in the higher scope that share the same identifier with a variable in a lower scope (unless explicitly declared), the programmer must be aware of potential race conditions they might create, and ensure that the statements and variable names executed are done so in a consistent sequence.

5 Type Conversions

In Red-Pandas, the programmer must explicitly convert operands to match the data types of an expression. Red-Pandas is strongly typed, and as a result, will not do any implicit type

conversions. Thus, when an operator has operands of different types, the compiler will reject the expression. It is up to the programmer to convert such operands to the correct type, using the built-in functions.

```
int num1 = 3;
double num2 = 3.0;
int f = num1 + num2  /* data types are not the same, compiler will reject */
```

```
int num1 = 3;
double num2 = 3.0;
int f = num1 + double2int(num2)    /* success! */
```

6 Built-in Functions

6.1 Conversion Functions

Red Pandas has two conversion functions to and from the int type:

- double2int(double x): truncates the fractional part of x and returns the integer part of x as an int.
    - double2int(matrix double x): performs double2int across all cells of x
- int2double(int x): returns x as a double where the fractional part of x is 0 and the integer part of x is equal to x
    - int2double(matrix int x): performs int2double across all cells of x
- mat2str(matrix x): returns matrix x as a string that is formatted so each row of the matrix is on a new line
- int2str(int x): returns int x as a string
- double2str(double x): returns double x as a string

6.2 Print Function

Red Pandas performs printing using the print function.

- The function prints the concatenation of the string representation of each expression to standard output
- Matrix objects have a mat2str() function that the user calls when printing matrices
- The print function is polymorphic

```
matrix int x = populate(1, 3, 3) /*creates a 3x3 matrix of 1s */
print(“matrix x = ” + x) /*prints formatted matrix x as [[1, 1, 1],
                                                          [1, 1, 1],
                                                          [1, 1, 1]]*/
```

6.3 Basic Matrix Functions

Red Pandas includes these basic matrix functions:

- Map(matrix x, function): applies the function to each element in matrix x
- transpose(matrix x): returns the transpose of matrix x
- populate(int x, int y, int z): creates an y by z matrix and fills each element of the matrix with x
- concat(matrix x, matrix y): intelligently concatenates x and y in the order given. Depending on which dimensions match, concatenates horizontally or vertically.
- sizeOfRow(matrix x): returns number of rows of matrix x
- sizeOfCol(matrix x): returns number of columns of matrix x

6.4 Library Functions

Red Pandas has a library of common matrix manipulation functions:

- det(matrix x): calculates and returns the determinant of matrix x
- inv(matrix x): returns the inverse of matrix x
- rank(matrix x): calculates and returns the rank of matrix x
- rref(matrix x): returns the reduced row echelon form of matrix x

7 Grammar

```
program:
  decls EOF    { $1 }

decls: /* nothing */  { [], [] }
  | decls vdecl      { ($2 :: fst $1), snd $1 }
  | decls fdecl      { fst $1, ($2 :: snd $1) }

fdecl:
  DEF typ VARIABLE LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
    { { typ = $2; fname = $3; formals = $5;
       locals = List.rev $8; body = List.rev $9 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list    { List.rev $1}

formal_list: typ VARIABLE                    { [($1,$2)] }
        | formal_list COMMA typ VARIABLE { ($3,$4) :: $1 }

typ: INT    { Int }
  |  BOOL   { Bool }
  |  VOID   { Void }
  |  STRING { String }
  |  DOUBLE { Double }
  |  MATRIX { Matrix }

vdecl_list: /* nothing */    { [] }
        | vdecl_list vdecl { $2::$1 }

vdecl: typ VARIABLE SEMI { ($1, $2) }

stmt_list:
  /* empty */   { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI                                          { Expr $1             }
  | BREAK SEMI                                         { Break Noexpr        }
  | CONTINUE SEMI                                      { Continue Noexpr     }
  | RETURN expr_opt SEMI                               { Return $2           }
  | LBRACE stmt_list RBRACE                            { Block(List.rev $2)  }
  | IF LPAREN expr RPAREN stmt %prec NOELSE            { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt               { If($3, $5, $7)      }
  | MAP LPAREN expr COMMA expr RPAREN                  { Map($3, $5)         }
  | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt
                                                       { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt                      { While($3, $5)       }
```

```
expr_opt:
  /* nothing */ { Noexpr }
  | expr          { $1 }


expr:
    expr PLUS   expr                { Binop($1, Add, $3) }
  | expr MINUS  expr                { Binop($1, Sub, $3) }
  | expr TIMES  expr                { Binop($1, Mul, $3) }
  | expr DIVIDE expr                { Binop($1, Div, $3) }
  | expr POWER expr                 { Binop($1, Pow, $3) }
  | expr EQUALS expr                { Binop($1, Equals, $3) }
  | expr NOTEQ expr                 { Binop($1, Noteq, $3) }
  | expr LESS expr                  { Binop($1, Less, $3) }
  | expr GREAT expr                 { Binop($1, Great, $3) }
  | expr LEQ expr                   { Binop($1, Leq, $3) }
  | expr GEQ expr                   { Binop($1, Geq, $3)}
  | expr AND expr                   { Binop($1, And, $3) }
  | expr OR expr                    { Binop($1, Or, $3) }
  | expr COLON expr                 { Range($1, $3)}
  | VARIABLE ASSIGN expr            { Equi($1, $3) }
  | MINUS expr %prec NEG            { Unop(Neg, $2) }
  | NOT expr                        { Unop(Not, $2) }
  | ABS expr ABS                    { Unop(Abs, $2) }
  | VARIABLE LPAREN actuals_opt RPAREN
                                    { Call($1, $3) }
  | TRUE                            { Bool(true) }
  | FALSE                           { Bool(false) }
  | VARIABLE                        { Var($1) }
  | DOUBLE_LITERAL                  { Dub($1)}
  | STRING_LITERAL                  { Str($1)}
  | LITERAL                         { Lit($1) }
  | LPAREN expr RPAREN              { $2 }

actuals_opt:
  /* empty */  { [] }
  | actuals_list { List.rev $1 }

actuals_list:
    expr { [$1] }
  | actuals_list COMMA expr {$3 :: $1 }
```

8 Sample Code

```
def matrix int foo(){
    int i = 3;   /* new variable, i = 3 */
    matrix int k = [[0,1,2],[2,1,5]]; /* new variable, k = 20 */
    while (i<8){
        i++;    /* this will loop until i = 7 when foo() is executed */
    }
    return k
}

def void printSquares(int x) {
      int s = x * x;
      print(x + " squared is " + s);
}

/* Replicating representative program from MathLight pg. 37 -> method for
linear regression */

def matrix double bar(matrix int f, matrix int g) {
      matrix int one = populate(1, sizeOfRow(f), sizeOfCol(f));
      new_f = concat(f,one);

      /* inv() needs to return double */
      matrix double i = inv(transpose(new_f) * int2double(new_f));
      i = i * int2double(transpose(new_f)) * int2double(g);

      return new_f * i;
}

void main() {

      matrix int x = [[1],[2],[3]];
      matrix int y = [[4],[5],[6]];

      print(bar(x,y));
}
```

9 References

1. Dennis M. Ritchie, C Reference Manual
2. The Funk language by 4115 students Naser AlDuaij, Senyao Du, Noura Farra, Yuan Kang, and Andrea Lottarini. http://www.cs.columbia.edu/~sedwards/classes/2012/w4115-fall/reports/Funk.pdf
3. The MicroC Compiler and documentation
4. OCaml Documentation and User's Manual. https://caml.inria.fr/pub/docs/manual-ocaml/