PLT Language Reference Manual

1) Group Members

Siddhi Mittal - sm3210
Sahil Yakhmi - sy2348
Damien Fenske-Corbiere - dpf2117
Dan Aprahamian - dha2108

2) Introduction

The **NUMLANG** programming language is designed to make numerical computation easy. One of the key features of this language is that it allows mathematical functions to be entered as literals. It allows computation with matrices and other common mathematical operations. The language is intended to be suitable for compilation as well as interpreting. The reference implementation is, however, a compiler.

3) Lexical Conventions

3.1) Comments

There is only one type of comment in this language, a block comment. A block comment is defined as anything in between the starting character sequence '/*', and the first occurrence of '*/'. Nothing within a comment is used by the compiler to generate code.

3.2) Identifiers

An identifier may be any alpha-numeric sequence of characters that begins with a letter character.

3.3) Keywords

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

`match`
`done`
`cont`
`loop`
`any`
`pass`
`sub`
`include`
`const`
`->`

4) Literals

4.1) Numerical Literals

Numerical literals can be specified as a sequence of decimal digits, optionally with a decimal point in any position. Additionally, an 'E' character and an exponent can optionally be specified following a number to multiply the number by 10 to a certain power. A numerical literal may contain no whitespace. For example: '`.00005332`', '`3234.0`', '`0.1`', '`1000.`', '`1.0E-4`', and '`.009E12`' are valid numerical literals.

4.2) *String* Literals
A *string* literal is anything between single quotation marks. Special characters can be escaped using a slash character.

4.3) Function Literals
Mathematical functions must be specified as literals using the following format:

```
(x) -> x + 1
(x, y) -> x + y
```

The functions are mappings from the comma-delimited list of variables in parenthesis to a single value, the value of the expression to the right of the -> keyword.
A set of parenthesis may optionally be included around the function literal, and may be required in certain contexts (such as when the function is part of a larger expression).

4.4) Array Literals
Single and multidimentional arrays can be declared in-line by writing the comma-delimited array elements in between braces using the following syntax:

```
arr = { 1, 2, 3, 4};
2darr = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

Additionally, arrays can be initialized with default values by specifying a size in between square brackets. For example:

```
arr = [4]
2darr = [2, 4]
```

Arrays elements will by default hold 0 values.

4.5) Matrices
Matrix literals are specified in much the same way as Arrays. If an m is written directly before the first brace with no intervening whitespace, the array declaration will be treated as a matrix. This will entail some additional checks for validity. For example, a matrix is required to have two dimensions, and the columns must be of consistent length. For example, `m{{1, 2, 3}, {1, 2, 3, 4}}` would be an invalid matrix literal, and `m{{{1, 2, 3}, {1, 2, 3}, {1, 2, 3}}, {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}}, {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}}}` would also be invalid.

Matrices can also be declared with default values by specifying their size between square brackets, prefixing an m. For example:

```
matrix = m[2, 3]
```

However, a matrix must always be two-dimensional.

5) Types
NUMLANG contains four fundamental types corresponding to the above literals. They are:

5.1) *num*
A numerical always stored with floating-point precision.

5.2) *string*
A sequence of zero or more ASCII characters.

5.3) *mFunc*
A mathematical function representing a mapping of one or more numerical variables to a uniquely corresponding set of numerical values.

5.4) *array*
An *array* structure that can contain any other type. *Arrays* can be multi-dimensional, and can contain varying types. A multi-dimensional *array* is simply an array of *arrays*, and jagged multi-dimensional arrays are legal.

5.5) *matrix*
A matrix is a special type of array that contains only *num* types and has two dimensions with columns of consistent length.

6) Syntax notation
In the syntax notation used in the manual, syntactic categories are indicated by the *italic* type.

7) Expressions
The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. Otherwise the order of evaluation of expressions is undefined.

<u>7.1) Unary operators</u>
Expressions with unary operators group right-to-left.

7.1.1 − expression
This is the numerical negation operator.

- If expression evaluates to a *num*, the result is a *num* equal to the negative of the *num*.
- If expression evaluates to an *mFunc*, the result is an *mFunc* that represents the negation of the expression *mFunc*.
- If expression evaluated to an *matrix*, the result is a *matrix* that represents the negated *matrix*.
- − applied to any other expression is illegal.

7.1.2 ! expression
This is the logical negation operator.

- If the expression evaluates to a *num*, the result is 0 if the *num* is non-zero, and 1 if the *num* is 0.

- If the expression evaluates to a *string*, the result is 1 if non-empty, and 0 if empty.
- If the expression evaluates to an *mFunc*, the result is a new *mFunc* that represents the logical negation of the *mFunc*.
- ! applied to any other expression is illegal.

7.2) Multiplicative operators
The multiplicative operators *, / , and % group left-to-right.
7.2.1 expression * expression
The binary * operator indicates multiplication.

- If both expressions evaluate to *num* then the result is a *num*
- If either of the expression is a *matrix* and the other is *num*, the result is a matrix where each element is the element from the original matrix multiplied with *num*.
- If both expressions are of the type *matrix*, then the result is a matrix where each element at a location is the multiplication of elements from the original matrices at the same location, provided that the matrices are the same size.
- If one operand is a *num* and the other is *mFunc*, the result is an *mFunc*.
- * applied to any other pair of expressions is illegal.

7.2.2 expression / expression
The binary / operator indicates division.

- The same type considerations as for multiplication apply, except attempting to divide a *num* by a *matrix* will result in an error.
- Attempting to divide by zero will also result in an error.

7.2.3 expression % expression
The binary % operator yields the remainder from the division of the first expression by the second.

- The same type considerations as for division apply.

7.2.4 expression # expression
The binary # operator yields the *matrix* multiplication of two *matrices*.

- If the first element is an nxm *matrix*, and the second element is an mxp *matrix*, then, then it returns an nxp *matrix* that is the result of the mathematical matrix multiplication.
- # applied to any other expression is illegal.

7.3) Additive operators
The additive operators + and − group left-to-right.
7.3.1 expression + expression
The result is the sum of the expressions.

- If any operand is a *string*, it is treated as the *num* 0 if it is empty, and 1 if it is non-empty.
- If both operands *num*, the result is also a *num*.
- If one operand is a *num*, and the other is an *mFunc*, the result is an *mFunc*.
- If both expressions are of the type *matrix*, then the result is a *matrix* where each element at a location is the addition of elements from the original *matrices* at the same location, provided that the *matrices* are the same size.
- If one operand is a *num*, and the other is a *matrix*, then the result is a *matrix* with each element = *old-element* + *num*.
- No other type combinations are allowed.

7.3.2 expression − expression
The result is the difference of the expressions.

- If both operands *num*, the result is also a *num*.
- If one operand is a *num*, and the other is an *mFunc*, the result is an *mFunc*. If both operands are matrices, the result is a matrix if the operands are the same size, else an error occurs.
- If the first element is a matrix, and the second element is a *num*, then the result is a new *matrix* where each element = *old-element - num*.
- No other type combinations are allowed.

7.4) Relational operators
The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to.
7.4.1 expression < expression
7.4.2 expression > expression
7.4.3 expression <= expression
7.4.4 expression >= expression
The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

- Relational operators are only valid where the operands are either *num* or *mFunc*.
- The result always is a *num*, unless one more more operand is an *mFunc*, in which case the result is another *mFunc*.

7.5) Equality operators
7.5.1 expression == expression
7.5.2 expression != expression
The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

- The equality operator is only valid where both operands are *num*s and when both operands are *string*s. The result is al

7.6) Concatenation Operator
7.6.1 expression . expression
The concatenation operator ‘.’ is only valid for situations where both operands are *string*s. The result is always a *string*.

7.7) Assignment operators
There is only one assignment operator, which groups right-to-left. It requires an lvalue (variable or array/matrix element) as its left operand. The value of the evaluated expression (right operand) is the value stored in the left operand after the assignment has taken place.
7.7.1 lvalue = expression
The value of the expression replaces the value stored in lvalue.

8) Declarations and Initializations:
In our language, variables follow are dynamically typed. The type of a variable could be any time at a given moment, and the type of a variable cannot be resolved until runtime.

8.1) Declaring and initializing a scalar:

All scalars are declared and initialized the first time they are referenced. Typically, this is via assignment:

```
lvalue1 = num; /* Declares lvalue1 as a num and assigns value num
                    */
lvalue2 = string;   /* Declares lvalue2 as a string and assigns value
                       string */
lvalue3 = mFunc;    /* Declares lvalue3 as a mFunc and assigns value
                       mFunc */
```

If the scalar is not explicitly initialized, it defaults to a *num* of value 0:

```
lvalue1;                  /* Declares lvalue1 as a num and assigns
                             value 0 */
lvalue3 = lvalue2 + 1;    /* Declares lvalue2 as a num and assigns
                             value 0. Treats lvalue 2 as num for
                             purpose of computing lvalue3 */
```

A scalar can also be declared as constant as such:

```
const lvalue1 = num;      /* Declares lvalue1 as a num, and assigns
                             value num. lvalue1 can no longer change
                             its type or value */
```

8.2) Changing and re-declaring a scalar:

As long as a scalar has not been declared as const, its value can be changed. Furthermore, it can be assigned a value that is of a different type than its first value. This results in re-declaring the scalar to the new type:

```
lvalue = 3;          /* Declares lvalue as num, assigns 3 */
lvalue = lvalue + 1; /* Assigns lvalue + 1 */
lvalue = -1;         /* Assigns -1 */
lvalue = "foo";      /* Re-declares lvalue as string, assigns "foo" */
lvalue = "bar";      /* Assigns "bar" */
lvalue = (x)->(x + 1);          /* Re-declares lvalue as func, assigns
                                   (x)->(x + 1) */
const lvalue = (x)->(2x - 7);   /* Re-declares lvalue as const func,
                                   assigns (x)->(2x-7) */
lvalue = 3;               /* Error: const cannot change type */
lvalue = (x) -> (x / 2);  /* Error: const cannot change value */
```

8.3) Declaring and initializing an array:

An array is a sequential list of values. Each value can be a *scalar*, *matrix*, or another array.
To declare an array:

array:

lvalue **=** *array-declaration*;

array-declaration:

[*array-size-declaration***]**

{ *array-element-list* **}**

array-size-declaration:
array-size
array-size **,** *array-size-declaration*

array-element-list:
array-element
array-element **,** *array-element-list*

array-element:
element
{ *array-element-list* **}**

Example:

```
arr1 = [5];              /* Declares an array of length 5
                            with default values */
arr1 = [4, 3];           /* Declares a two dimensional
                            array of size 4x3 with default values */
arr1 = [2, 7, 4, 10];    /* Declares a four-dimensional
                            array of size 2x7x4x10 */
arr1 = {1, 2, 3, 4};     /* Declares a size-4 array with
                            the listed values */

/* Declares an array of arrays with the listed values*/
arr1 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

8.4) Declaring a matrix:

A matrix is akin to a mathematical matrix, a two-dimensional representation of a list of equally-sized vectors.

To declare a matrix:

matrix:
lvalue = *matrix-declaration*;

matrix-declaration:
m[*num-rows*, *num-columns*]
m{*matrix-row-list*}

matrix-row-list:
matrix-row
matrix-row, matrix-row-list

matrix-row:
num

num, matrix-row

All rows in a matrix must have the same number of elements. Also, all elements in a matrix must be numbers. Matrices are therefore more restrictive than arrays, but have the benefit of being allowed to perform mathematical operations on them.

8.5) Declaring a subroutine

To declare a subroutine:

subroutine:

`sub` *subroutine-name* **`(`***parameter-list***`)`** *statement*

parameter list:

parameter-name

parameter-name **`,`** *parameter-list*

9) Statements:

The program is made up of a series of statements. A statement is in the following format:

statement:

expression-statement

block-statement

match-statement

null-statement

Unless otherwise specified, all statements are executed sequentially.

9.1) Expression Statement:

An expression statement simply consists of an expression and an expression terminator. Most statements are expression statements.

expression-statement:

expression **`;`**

9.2) Block Statement:

Block Statements allow one to group multiple statements into one statement, useful for when only one statement is expected. The Block Statement is defined as follows:

block-statement:

`{`*statement-list***`}`**

statement-list:

ε

statement statement-list

9.3) Match Statement:

Match Statements are used for control flow. They incorporate features normally found in languages in if, switch, and while statements. They are defined as follows:

match-statement:
  `match(`*expression$_a$***`){`***match-list***`}`**

match-list:
  ε
  match-command match-list

match-command:
  flow-type match-condition **`?`** *statement*

flow-type:
  ε
  `cont:`
  `done:`
  `loop:`

match-condition:
  expression$_b$
  match-comparator expression$_b$
  match-type

match-comparator:
  `>`
  `>=`
  `<`
  `<=`
  `!=`

match-type:
  `SCALAR`
  `STRING`
  `MFUNC`
  `TRUE`
  `ANY`

The way the match works is as follows:

1) Start
2) For each *match-command* in the *match-list*, do the following:
   a. Determine if the condition matches

i. If the -match-condition is *expression*$_b$, the condition matches if *expression*$_a$== *expression*$_b$

ii. If the match-condition is *match-comparator expression*$_b$, the condition matches if **(***expression*$_a$ *match-comparator expression*$_b$**) != 0**

iii. If the match-condition is a *match-type*, the condition matches in the following cases:

1. **`NUM`**: *expression*$_a$ returns a number
2. **`STRING`**: *expression*$_a$ returns a *string*
3. **`MFUNC`**: *expression*$_a$ returns a func
4. **`TRUE`**: *expression*$_a$ returns a non-zero value
5. **`ANY`**: always matches

b. If the condition matches, do the following:

i. Perform the *statement*

ii. Depending on the *flow-type*, do the following:

1. **`cont:`**: proceed to the next iteration of Step 2.
2. **`done:`**: proceed to step 3
3. **`loop:`**: proceed to step 1
4. **`ε:`**: treat as cont

3) Finish

9.4) Null Statement:

The Null Statement is useful for places where you need a placeholder that does nothing. It is defined as follows:

null-statement:
```
pass;
```

10) Scope rules

A program consists of one or more files (via include) that are compiled together. Variables declared in the top level of a file are in the global scope. Otherwise, the language implements block level scope. For example, if a variable is first declared in a match statement, it will not be accessible once the match statement has finished.

A subroutine may only be declared in the top level of the program, and cannot be nested within another subroutine.

11) More on Types

11.1) Scalar Types

11.1.1. *num*

- A *num* is a basic floating point or integer number. Basic arithmetic rules apply.
  - `a + b`: add b to a
  - `a - b`: subtract b from a

        - `a * b`: multiply a by b
        - `a / b`: divide a by b. b cannot equal 0
        - `a % b`: returns the remainder of a / b. b cannot equal 0.
        - `-a`: returns the negative value of a.
    - In addition, *nums* are also used as boolean types. 0 is false, non-zero is true.
        - Integer to boolean operations
            - `a == b`: returns 1 if a is equal to b, 0 otherwise
            - `a != b`: returns 0 if a is equal to b, 1 otherwise
            - `a > b`: returns 1 if a is greater than b, 0 otherwise
            - `a >= b`: returns 1 if a is greater than or equal to b, 0 otherwise
            - `a < b`: returns 1 if a is less than b, 0 otherwise
            - `a <= b`: returns 1 if a is less than or equal to b, 0 otherwise
        - Boolean to boolean operations
            - `!a`: returns 0 if 1, 1 if 0
            - to achieve AND and OR operations, use * and + respectively
                - ex: `a + b === a OR b`
                - ex: `a * b === a AND b`

11.1.2 *string*

- A *string* is a series of characters (ex: “Hello”, “Goodbye”)

11.1.3 mFunc

- A mFunc is a mathematical function that takes in certain values and returns a *num*
- Literal: (*input-params*) -> *function-of-input-params*
    - Ex: (x) -> 2x + 3;
- Assigning function to variable: *lvalue* = *literal*
    - Ex: f = (x) -> 2x + 3;
- Evaluating function at value: *function*(*value*)
    - Ex: f(3); /*Returns 9*/
- Operators on functions all return new functions that combine both operands.
    - Valid operations: +, -, *, /, %, >, >=, <, <=, ==, !=
    - Ex:
        - f = (x) -> (x + 1);
        - g = f + 1; /* g == (x) -> x + 1 + 1 */
        - h = f * g; /* h == (x) -> (x + 1) * (x + 2) */
- Can combine functions
    - Ex:
        - f = (x) -> 2x - 3;
        - g = (x) -> x + 1;
        - h = f(g); /* h == (x) -> 2(x + 1) - 3*/

<u>11.4) Subroutines</u>

- Subroutines must be defined on a global level.
- Defining a subroutine: `sub` *subroutine-name* (*parameter-list*) *statement*
- Ex:

- `sub mySum(a, b) return a + b;`
- ```
  sub lotsofstuff(a, b, c)
  {
        a = b + c;
        b = b + b;
        c = a + a;
        return c; }
  ```
- `mySum(5,2); /*Calling the subroutine mySum*/`
- `lotsofstuff(1,2,3) /*Calling the subroutine lotsofstuff*/`

<u>11.5) Important functions</u>

- `return` *expression*;
  - Exits a subroutine with the value returned by expression. If used on global level, ends program.
- `str(`*num*|*func*|*array*|*matrix*`)`
  - Returns *a string* representing the passed in *num, func, array,* or *matrix*.
- `num(`*string*`)`
  - Returns the number value of a *string*
- `floor(`*num*`)`
  - Returns the integer value of a *num*, going downward
- `ceil(`*num*`)`
  - Returns the integer value of a *num*, going upward
  ```