μPerm

Language Reference Manual

1 Introduction

μPerm is a block-structured, dynamically typed, dynamically scoped, interpreted programming language for computation in the mathematical realm of the symmetric groups. In particular, μPerm provides as a primitive type the *permutation*, along with its related group-theoretic operations of composition and inversion.

1.1 Hello, world!

μPerm's hello world is simple:

```
print "Hello, world!";
```

To make the thing go, type it into a file called `hello.p` and execute it:

```
%> java uperm hello.p
Hello, world!
```

1.2 Euler's GCD Algorithm

We have to declare a recursive function first. Note that the function declaration does not take formal parameters, and that actual parameters appear in the `args` list. Type the function declaration and the two print statements into `gcd.p` and execute it.

```
fun gcd {
   if (args[0] % args[1] == 0) return n;
   return gcd (args[1], args[0] % args[1]);
}

print gcd (377, 2463);
print gcd (13, 101);
```

```
%> java uperm gcd.p
29
1
```

Notice that μPerm can print the value of the function call, which in this case is an integer.

1.3 Build a List

μPerm supports heterogeneous, dynamically-sized lists. `list.p`:

```
l = [0, 1, true, "three"];
l = l + [4]; $ append 4 to l
print l;
print l[2];
```

```
%> java uperm list.p
[0, 1, true, 'three', 4]
true
```

Lists are indexed from zero. Also note that the comment character is `$`, because I will never make any money with *μ*Perm.

1.4 A Few Permutations

*μ*Perm was created so we can explore the symmetric groups. Let's take a look at how permutations work. Note that permutation literals do *not* include commas. Don't worry yet about what the heck this all means – more on that later. `permutation.p`:

```
g=(1 2 3 4 5); p = ();

print p;
for (p = g; p != (); p = p * g) print p;
print p;
```

```
%> java uperm permutation.p
()
(1 2 3 4 5)
(1 3 5 2 4)
(1 4 2 5 3)
(1 5 4 3 2)
()
```

The loop shows us an example of a mathematical fact: that any permutation, multiplied by itself enough times, will always bring you back to the *identity permutation*, which we denote `()`.

As a convention, *μ*Perm prints permutations beginning with the smallest number appearing in the permutation.

2 Mathematical Background

2.1 Definition

A *permutation* π is a bijection from a set S to itself. For our purposes, S must be finite. We say that element $x \in S$ is *fixed* if $\pi(x) = x$; otherwise we say it is *moved* by π. Two permutations π and π' are equal if $\pi(x) = \pi'(x)$ for every element x of S. Every permutation π has a unique inverse, π^{-1}, which is the inverse mapping of π.

2.2 Image Form

Permutations were traditionally written in *image form*: two rows of numbers specifying the function with the ordered set S at top and each element's image under π at bottom. Given S={1,2,3,4,5} and π mapping 1 to 3, 2 to 4, 3 to 2, 4 to 5, and 5 to 1, we write:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix}$$

μPerm's grammar accepts permutations using a summarized image form consisting of the bottom row of numbers delimited with a backslash, as in

```
pi = \3 4 2 5 1\;
```

2.3 Cyclic Form

Permutations also admit to a more succinct *cyclic form*. Suppose that permutation ρ moves n elements, for some $n > 1$. We can arrange these n elements in a sequence $\{x\}_i$, with $x_i = \rho(x_{i-1})$, and $x_1=\rho(x_n)$. A permutation in cyclic form is just such a sequence, enclosed in parentheses. The cyclic form of π from Section 2.2 is

$$\pi = \begin{pmatrix} 1 & 3 & 2 & 4 & 5 \end{pmatrix}$$

The cyclic form is generally preferred for its succinctness:

```
pi = (1 3 2 4 5);
```

2.4 Identity and Inverse

The identity permutation is that which fixes all elements of S. A permutation composed with its inverse yields the identity.

The μPerm programmer may represent the identity permutation using the unique cyclic form `()` or the unique image form `\\`. Also, we may express the inverse of a given permutation by prepending the ~ character to a permutation or variable, as in

```
~pi
```

It is always true that a permutation multiplied by its inverse on either side will yield the identity. As permutations are mathematical functions, μPerm allows permutations to be evaluated as a function call on a single integer. Given `pi` as defined above, we might wish to know the value of

```
pi(2)
```

which of course is `4`.

2.5 Composition

Permutations naturally admit to an operation: composition of functions. Given another permutation of S, say

$$\pi' = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 5 \end{pmatrix}$$

We might write

$$\pi \cdot \pi' = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 5 & 4 \end{pmatrix}$$

The reader is left to verify that, for example, $\pi'(\pi(1)) = 2$. μPerm composes functions from left to right.

μPerm uses the asterisk to denote composition of permutations, so that, for example, the expression

```
\3 4 2 5 1\ * \4 3 2 1 5\ == \2 1 3 5 4\
```

evaluates to `true`.

Wwritten in cyclic form, $\pi \cdot \pi'$ above can be seen to be the composition of two permutations, each moving two elements of the underlying set:

$$\pi \cdot \pi' = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 5 & 4 \end{pmatrix} = (1 \quad 2) \cdot (4 \quad 5)$$

Unmoved elements, such as 3 in this example, do not appear in the cyclic form. The μPerm grammar faithfully implements the cyclic form of permutations, so that we might store the above permutation in a variable as follows:

```
rho = (1 2) * (4 5);
```

2.6 Associativity and Non-Commutivity of Composition

Permutation composition is associative, but not commutative in general. That is, $\pi\rho \neq \rho\pi$ generally. Two cycles which share no common element are called *disjoint*. Disjoint cycles commute.

2.7 No Permutation Moves a Single Element

One further remark about permutations will be useful: no permutation moves a single element. If $\pi(x) = y$, then $\pi(y) \neq y$ (because π is a bijection) and so y is also moved by π. This is an important fact, as it allows μPerm to retain the expected meaning of expressions such as `(-5)`, which simply represents the integer -5.

With that knowledge in hand, we can fully describe μPerm's *Permutation* type in Section 4.1.4.

3 Lexical Elements

3.1 Character Set

μPerm understands the regular ASCII character set, including predefined language strings for tabs and various newline formats (μPerm does not provide escape characters). μPerm is case sensitive.

μPerm scans its input according to the following lexical scheme:

Letter → ('a' .. 'z' | 'A' .. 'Z' | '_') ;
Digit → '0' .. '9' ;
Integer → *Digit*+ ;
Identifier → *Letter* (*Letter* | *Digit*)* ;

3.2 Literals

3.2.1 Boolean Literals

The Boolean literals are `true` and `false`.

3.2.2 Numeric Literals

Integer literals take the form of an optional negative sign followed by one or more digits.

3.2.3 String Literals

Strings are denoted with starting and ending double-quotes, and accept the printable ASCII characters. μPerm does not provide escape characters.

3.2.4 Permutation Literals

Permutation literals may take one of two forms: the *cyclic form* and the *image form*. See Section 4.1.4 for a detailed description of the meaning of the forms.

The image form is a backslash-delimited white-space separated list of integers: `\1 4 5 3\`. Image forms must include every positive integer less than the largest integer in the form. Zero may optionally be included. The cyclic form is a parenthetical white-space separated integer list containing at least two integers: `(1 5 7 3)`, or `(89 2)`. Cyclic forms may also include zero. In neither form may an integer included more than once. The identity permutation can be represented in either form, but expressed cyclically it is very succinct: `()`.

The cyclic form is preferable in general.

3.2.5 List Literals

List Literals are comma-separated lists of μPerm expressions enclosed in brackets [].

3.2.6 Identifiers

Identifiers have typical lexical style, a letter followed by one or more numbers and underscores.

3.2.7 Reserved Words

μPerm's reserved words are:

```
local         if            while         print         fun
return        break         continue      import        true
false         sizeof
```

3.2.8 Operators

μPerm's operators are:

```
+       -       *       /       %       ~
&&      ||
==      !=      <       >       <=      >=
=       sizeof
```

The above operators carry their usual meaning. Of note however are the unary operators ~ and `sizeof`. The ~ operator must appear before a permutation; the resulting expression resolves to the inverse of the operand. The `sizeof` operator is described in Section 5.9.

3.2.9 Comments

Comments begin with `$` and continue to the end of the line, because I will never make any money with μPerm. Comments are discarded by the μPerm parser.

4 Types

4.1 Primitive types:

μPerm's primitive types are *Boolean*, *Integer*, *String*, and *Permutation*. Their semantics are described below.

4.1.1 Boolean

Variables of *Boolean* type can take the logical values of `true` or `false`. Boolean values can be manipulated according to the Boolean calculus. The *Boolean* operators are:

- Relational == and !=
- Complement !
- Logical && and ||
- String Concatenation +

Logical

|| is inclusive.

String Coercion

Boolean values may be coerced into *String* values using the string concatenation operator +. It is a runtime error to apply + to two values if one has *Boolean* type and the other does not have *String* type.

4.1.2 Integer

Variables of Integer type represent whole number values from –2147483648 to 2147483647, inclusive. The *Integer* operators are:

- Arithmetic +, –, *, /, and %
- Relational ==, !=, <, <=, >, and >=
- String Concatenation +

String Coercion

Integer values may be coerced into *String* values using the string concatenation operator. It is a runtime error to apply + to two values if one has *Integer* type and the other does not have *String* or *Integer* type.

4.1.3 String

Values of *String* type represent sequences of those characters accepted by μPerm. The *String* operators are:

- Relational ==, !=, <, <=, >, and >=
- String Concatenation +

Relational

Two *String* values are equal if and only if they represent the same sequence of characters. *String* A is less than string value B when A precedes B in the standard lexicographic ordering.

4.1.4 Permutation

See Section 2 for a discussion of the mathematical background of permutations.

Values of *Permutation* type represent mathematical permutations. Integers mapped by permutations must lie between 0 and 100, inclusive. Permutation literals may not have more than 101 elements, none of which may be negative or greater than 100.

A Permutation value `pi` that does not contain in its array some integer `y` is called *ill-formed*. Such a mapping π has no x for which $\pi(x) = y$, and is not a permutation. Otherwise, a Permutation `pi` value is *well-formed*. Upon detection of an ill-formed value, μPerm immediately aborts program execution and displays an error message. *Permutation* values are internally initialized to the identity, and the methods of the underlying Java class are robust. Errors will only occur upon introduction of an ill-formed literal.

The Permutation operators are:

- Relational `==` and `!=`
- Composition `*`
- Inversion `~`
- String Concatenation `+`

Relational

Two permutations are equal if and only if they represent the same mapping.

Composition

Composition is carried out from left to right. Composition occurs only between permutations; it is a runtime error to apply `*` to two values if one has *Permutation* type and the other does not.

String Concatenation

Permutation values may be coerced into *String* values using the string concatenation operator. It is a runtime error to apply + to two values if one has *Permutation* type and the other does not have *String* type.

Permutations as Functions

Permutation variables may also be called as functions taking a single positive integer.

4.2 Composite Types

μPerm provides a single composite type: *List*.

4.2.1 Lists

The *List* type represents an ordered collection of values. μPerm lists are heterogeneous: each element in a list may take on any value of any type, including *List* values. Lists are *indexed* by a single non-negative *Integer*. Lists are *sliced* by zero, one, or two non-negative *Integers*. List slices may not be assigned to. The List operators are:

- Relational == and !=
- Append +
- String Concatenation +

Relational

Two lists are equivalent when they contain exactly the same ordering of the same elements.

Append

Applying the + operator to two objects of *List* type results in a list containing the elements of the first list followed by the elements of the second list.

String Concatenation

List values may be coerced into *String* values using the string concatenation operator. It is a runtime error to apply + to two values if one has *List* type and the other does not have *List* or *String* type.

4.3 Type Coercion

μPerm provides type coercion from any type to the *String* type. This coercion is achieved through use of the string coercion operator +. μPerm will convert to a string any arithmetic expression involving a string, given that the expression does not violate any semantics of the language.

5 Expressions

5.1 The Expression Grammar

μPerm interprets expressions according to the following grammar:

Expression → *Boolean ;*
Boolean → *Join (|| Join)* ;*
Join → *Equality (&& Equality)* ;*
Equality → *Relational ((== | !=) Relational)* ;*
Relational → *Arithmetic ((< | <= | > | >=) Arithmetic)* ;*
Arithmetic → *Multiplicative ((+ | -) Multiplicative*)* ;
Multiplicative → *Unary ((* | / | %) Unary*)* ;
Unary → - *Atom*
| - *Atom*
| ~ *Atom*
| ! *Atom*
| `sizeof` *Atom*
| *Atom ;*
Atom → *PermutationLiteral*
| *FunctionCall*
| *List*
| *Slice*
| *Identifier*
| *IntegerLiteral*
| *StringLiteral*
| *BooleanLiteral*
| (*Expression*) ;
PermutationLiteral → (*(Integer | Identifier | Slice | FunctionCall)*
(Integer | Identifier | Slice | FunctionCall)+)
| \ *(Integer | Identifier | Slice | FunctionCall)*
(Integer | Identifier | Slice | FunctionCall)+ \
| ( ) ;
FunctionCall → *Identifier* (*Expression**) ;
List → [*Expression**];
Slice → *Identifier* [*Expression? :? Expression?*] ;
StringLiteral → " *Letter | Digit | Space | Tab | …* " ;
BooleanLiteral → `true` | `false` ;

5.2 Order of Evaluation

μPerm expressions are evaluated in a left-to-right, bottom-up order.

5.3 Boolean Expressions

Boolean expressions consist of expressions related together using any number of the Boolean relational, complement, or logical operators. Operator binding and parenthetical expression grouping obey semantics typical of the Boolean domain.

5.4 Integer Expressions

Integer expressions consist of identifiers or literals related together using the unary and binary integer operators. The modulus operator has multiplicative precedence.

5.5 String Expressions

String expressions consist of identifiers or literals of any type, at least one of which evaluates to String type, related together using any of μPerm's arithmetic or relational operators. μPerm attempts to coerce the constituents of any such expression to *String* type, so that the final value of the expression has *String* type.

5.6 Permutation Expressions

Permutation expressions consist of identifiers or *Permutation* literals related together using the *Permutation* operators. *Permutation* identifiers also admit to a functional notation wherein the identifier is followed by a parenthetical single integer; these expressions are called *permutation evaluations*, and have *Integer* type.

5.7 List Expressions

List expressions consist of identifiers or List literals related together using the List operators. List identifiers also admit to indexing and slicing notations, wherein the identifier is followed by a series of one or two integers enclosed in square braces, and possibly incorporating a colon. If the series includes a colon, the expression is a Slice. Lists are indexed and sliced from zero, using non-negative integers. List indexes and slices may not be assigned to.

Given

```
L = [0, 1, 2, true, (1 2 3 4), [1, 2 ,3]];
```

`L[0]` evaluates to the integer `0`.
`L[0:3]` evaluates to the list `[0, 1, 2, true]`.
`L[:2]` evalueates to the list `[0, 1, 2]`.
`L[4:]` evaluates to the list `[(1 2 3 4), [1, 2, 3]]`.
`L[:]` evaluates to the whole list `L`.

The empty list `[]` is a valid object.

5.8 Function Calls

Functions may be invoked by issuing a function call (see Section 6.3 for information regarding functions called as statements). Permutation identifiers may be called as well, using

The *Expression*s enclosed in parentheses are evaluated left-to-right and stored *by value* in a List named `args`, which is itself automatically created in the associated function *Block*. The μPerm programmer may pass any number of values into a function.

Upon parsing a function call, the interpreter executes all *Expression* parameters, stores them in `args`, and executes the associated *Block*. When a *ReturnStatement* is executed within the function, execution of the block immediately halts, and the function call takes on the returned value within the enclosing *Expression*. *ReturnStatements* without a return value cause the *Boolean* value `true` to be returned.

Dynamic scoping semantics allow functions to have side effects, by allowing access to externally scoped variable in the course of execution.

5.9 `sizeof` Expressions

The `sizeof` keyword can be placed before any *Atom*. The resulting expression always resolves to an integer. Applied to objects of *Boolean* or *Integer* type, the `sizeof` operator yields `1`. Applied to strings, the expression resolves to the length of the string; to permutations, the highest integer moved by the permutation; to lists number of elements in the list.

6 Statements

μPerm provides simple statement syntax:

Statement → *AssignmentStatement*
| *FunctionDeclaration*
| *FunctionCallStatement*
| *PrintStatement*
| *IfStatement*
| *ForStatement*
| *WhileStatement*
| *ReturnStatement*
| *BreakStatement*
| *ContinueStatement*
| *ImportStatement*
| *BlockStatement*

6.1 Assignment Statements

AssignmentStatement → `local`*? Identifier* = *Expression* `;`

Upon parsing an assignment statement, *μ*Perm first evaluates and temporarily stores the type and value of *Expression*. It then searches for *Identifier* in the symbol table, according to semantics described in Section 7.2. If it is found, *μ*Perm associates with it the value of *Expression*. If *Identifier* is not found, *μ*Perm creates a new symbol in the local symbol table.

Variables come into existence upon first assignment, and are destroyed when execution exits their enclosing block (see Section 7.2).

*μ*Perm allows programmers to force creation of a variable in the immediately enclosing block by using the keyword local. So, the output of the following program is `200` *newline* `100`.

```
i = 100;
{
   local i = 200;
   print i;
}
print i;
```

Identifiers have no affinity with type. A value of any type can be assigned to any identifier at any time.

6.2 Function Declarations

FunctionDeclaration → `fun` *Identifier Block*

Functions are declared using the `fun` keyword, which serves to associate *Identifier* with the *Block* of code. The association is stored in the current symbol table.

μPerm does not allow the programmer to specify input parameters to functions. See Section 6.3 for information on passing values into functions.

6.3 Function Call Statements

FunctionCallStatement → *Identifier* `(` *Expression** `)` `;`

Functions may be invoked by issuing a function call statement (see Section 5.8 for information regarding function calls as part of an *Expression*).

Functions are called as statements only for their side effects; any values returned from a function called as a statement are discarded.

The *Block* associated with *Identifier* is executed in the context of the *FunctionCallStatement*; all declarations, assignments, variable references, expressions, inner function definitions and function calls therein are evaluated as if the *Block* program text (including enclosing braces) had directly replaced the *FunctionCallStatement*. This is the essence of dynamic interpretation.

The Expressions enclosed in parentheses are evaluated left-to-right and stored *by value* in a List named args, which is itself automatically created in the associated function *Block*. The μPerm programmer may pass any number of values into a function.

6.4 Print Statements

PrintStatement → `print` *Expression* `;`

*PrintStatement*s have the effect of printing the value of *Expression* to the standard output stream of the local shell or operating system. Strings appearing inside of lists are output enclosed in single-quotes.

6.5 If Statements

IfStatement → `if` `(` *Expression* `)` *Statement* (`else` *Statement*)*?* `;`

Regular if-then-else semantics apply. *Expression* must evaluate to a Boolean value.

6.6 For Statements

ForStatement → `for` `(` *AssignmentStatement* `;`
Expression `;`
(*AssignmentStatement* | *FunctionCallStatement*) `)` *Statement*

The usual for loop semantics apply. The *Expression* must evaluate to a Boolean.

Variables created in the *AssignmentStatement*s are local to the loop and are destroyed upon termination.

6.7 While Statements

WhileStatement → `while (` *Expression* `)` *Statement*

Expression must have Booolean type.

6.8 Return Statements

ReturnStatement → `return` (*Expression*)? `;`

Has the effect of setting the value of a surrounding function to *Expression* and executing a *BreakStatement*. If *Expression* is omitted, `true` is returned. *ReturnStatements* executed within a non-function context have an effect equivalent to the execution of a *BreakStatement*.

6.9 Break Statements

BreakStatement → `break ;`

Has the effect of halting execution until the statement immediately following the innermost surrounding block or function context.

6.10 Continue Statements

ContinueStatement → `continue ;`

Suspends execution for the remainder of the immediately enclosing loop context and resumes execution at the top of the loop.

6.11 Import Statements

ImportStatement → `import` *"filename"* `;`

Import statements cause the μPerm interpreter to evaluate the file indicated by *filename*. Filename must be a string acceptable to the local shell or operating system environment. The evaluation occurs in the context of the *ImportStatement* itself; the effect is equivalent to replacing the *ImportStatement* with the exact contents of *filename* and evaluating the result.

6.12 Block Statements

BlocStatement → `{` *Statement* * `}`

Causes the creation and stack push of a new symbol table, against which the *Statement*s are executed. When execution is complete, the symbol table is destroyed.

7 Scope

7.1 Structure

μPerm is block-structured and dynamically scoped.

7.2 Blocks

A μPerm block is a sequence of statements enclosed by `{` and `}` characters. Each block represents a unique namespace in which names may be associated with values to form variables, or with code blocks to form functions.

Each block maintains a symbol table; as program control enters and exits blocks, symbol tables are pushed and popped onto an execution stack. Names are resolved by searching the stack beginning at the current symbol table and proceeding downward in the stack until either a value is found or the bottom of the stack is reached. The latter causes an error and immediate halt to program interpretation.

7.3 Functions

All values are assigned, evaluated, and passed into functions by *value* in the form of a *List* `args` that is created in the symbol table of the function when it is called.

7.4 Control-Flow Constructs

If the Statement of a particular control-flow construct resolves to a BlockStatement, then that block and all of its local declarations are created and destroyed with each iteration of the construct. In the case of for loops, any variables created during initialization are survived by the creation and destruction of the blocks.

8 Errors

All lexical, syntactic, and semantic errors result in the throwing of a Java exception, along with a short description of the error, and immediate halt of program interpretation.