μPerm

A Small Permutation Group Engine

by: Gregory Kip

COMS W4115 *Programming Languages and Translators*
Prof. Stephen Edwards

Abstract

Given the abstract character of much of modern physics and mathematics, and the computational tasks demanded by work within the domain of abstract algebra, a general-purpose computing language capable of expressing and manipulating the carbon atoms of modern algebra – permutations – may be useful. We propose an interpreted, structured, dynamically typed programming language with Pascal-like syntax for the expression and manipulation of permutations within permutation groups. We call the language μPerm.

Mathematical Background

Given a set **S** the Symmetric group *Sym*(**S**) is a group whose elements are the bijections from **S** onto itself, the *permutations* on **S**. The operator of *Sym*(**S**) is composition of functions. For our purposes, **S** will always be finite.

We denote the action of an element g of *Sym*(**S**) on $s \in \mathbf{S}$ as $g(s)$. Given $g_1, g_2 \in Sym$ (S), we denote $g_1(g_2(s))$ as $g_1 \quad g_2\,(s)$, $s \in \mathbf{S}$. We will sometimes be unconcerned with the result of a permutation on a single element of **S**, in which case we will simply refer to $g_1 g_2$.

Programming Examples

μPerm will provide the ability to manipulate permutations using such statements as:

```
p1 : perm := (1 2) (5 3 4);
p2 : perm := (3 5 1 2);
print p1 * p2; -- * denotes composition
```

Output:

```
(1 3 4)
```

```
for p in (1 2) (3 4 5) to (1 2 3 5) do
begin
   print p;
end;
```

The output of the above program is:

```
(1 3) (4 5)
```

because (1 2) (3 4 5) * (1 3) (4 5) is (1 2 3 5).

Primitive Types

Common primitive types will be supplied in as simple a form as possible: `int`, `float`, `string`. Numeric types will have the four arithmetic operators; strings will admit concatenation.

The Permutation Type

In addition, μPerm introduces its special primitive type, the permutation. Permutation objects will be denoted using the keyword `perm`, as seen above. perm will admit two operators: `*` for composition of permutations, and `/` for decomposition. Operations between perm objects and other primitive types will not be provided. For example, the sentence `5 + (1 9)` is not a valid μPerm expression.

All operators on objects of primitive type will be left associative, with a possible exception noted below.

One syntactic ambiguity that could arise is the interpretation of such strings as `(7)`. Does this express the integer 7 or the permutation g with $g(7) = 7$? μPerm will always interpret such expressions as integer expressions. However μPerm will treat `()`, if encountered, as the identity permutation. (We may encounter a syntactic conflict between the identity permutation and function calls with no parameters. We will deal with that possibility as it comes.)

Another syntactic issue that may arise is the shorthand expression of composition – the empty string. Mathematicians are perfectly comfortable notating the composition of (1 3 5) with (2 4) as (1 3 5) (2 4). This may not be a problem if we can come up with a way for the compiler to recognize such statements as being compositional. If we cannot, we can fall back on the explicit (1 3 5) * (2 4).

We would like to interpreter in such a way that it assumes the underlying set is only as large as the largest integer encountered in a given permutation literal. But it may become necessary to force the programmer to specify the size of the permutation group he intends to work with. If so, we will introduce a right-associative operator to be applied at the end of a permutation literal, such as:

```
p : perm := (1 3 5)#6;
```

which will direct the compiler to work within $Sym(\mathbf{Z}_6)$.

It may become difficult to specify permutation literals using space-separated lists. If this becomes the case, we can introduce commas as separators.

Objects of type perm will also admit a built-in functional notation for the purpose of applying a permutation $g \in Sym(\mathbf{S})$ to $s \in \mathbf{S}$. The program

```
p : perm := (1 3 5);
```

```
print p(3);
print p(4);
```

will produce the output

```
5
4
```

We will also allow the use of C-like enumerations within a permutation declaration. The compiler will assign the smallest unused integer to each enumeration literal it encounters, or the programmer may specify values for the compiler to use. Thus the program

```
p : perm := (1 Milk 3 Eggs => 5 Cheese);
print p(3);
print p(Eggs);
print p;
```

will produce the output

```
5
2
```

We will consider allowing the programmer to specify a previously declared variable identifier in a given permutation literal, and introducing square brackets as a syntactic mechanism to get at the value – of any type – of the variable. For example,

```
h : string := "Hello";
pi : float := 3.14159;
i : int := 2;

p : perm := (h, i, pi, 4);

print p;
print p[1];
print p(p[i]));
```

Would yield output

```
("Hello" 2 3.14159 4)
```

This may prove overly-complicated and not useful. An alternative is to allow for the creation of arrays in which the programmer can store the desired data.

Language Constructs

Control-flow

µPerm will provide the usual control flow mechanisms: **if-then-else**, **for**, **while**, **do-while**.

To visit elements of the underlying set in the order specified by the permutation, a programmer might write:

```
p : perm := (3 4 1 2);
i : int := 1;

loop
   visit (p(i));
   i := p(1);
   if i != 1 exit loop;
end loop;
```

Block-structure

Programmers will also be able to define functions and procedures. Passing semantics will be by-reference. Blocks will be delimited with **begin** and **end** keywords, and can be named.

Compilation / Interpretation

Our aim is to produce a Java program to interpret µPerm code.

Notes:

I came about this idea more in the mindset of producing a language for computational group theory – particularly the permutation groups, represented using so-called *strong-generating sets*. I quickly became sidetracked with the lower-level ideas communicated herein, and decided that, given my tardy proposal along with the potential to write a decent, moderately-sized language, I had best just get on with the ideas I had.

Time permitting, I would also like to include some higher-level abstractions, such as the notion of a symmetry group itself, generators within that group, and so on. Computational group theory is a relatively young field, albeit with some mature results, especially regarding the computational complexities of representing and working with permutation groups. It is my hope that I may be able to implement certain of the simpler algorithms in the field (e.g. strong generating set enumeration) using μPerm.