μPerm

Language Reference Manual

By: Gregory Kip

COMS W4115 Programming Languages and Translators
Fall 2007
Prof. Stephen Edwards

1 Lexical Elements

1.1 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 insensitive to case.

1.2 Literals

1.2.1 Boolean Literals

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

1.2.2 Numeric Literals

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

1.2.3 String Literals

String
No escape characters. Instead we provide reserved strings CR, FF, NL, QT, TB.

1.2.4 Set Literals

Literal syntax is a comma separated list between curly braces: **`{1, 4, 5, 9}`**. The null set is denoted **`{}`**.

1.2.5 Permutation Literals

Permutation literals are white-space separated integer lists containing at least two integers, enclosed in parentheses: **`(1 5 7 3)`**. Also, we can specify permutations using the image format, a backslash-delimited white-space separated list of integers: **`\1 4 5 3\`**. Use of the backslash differentiates from the division operator **`/`**.

1.2.6 Identifiers

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

1.2.7 Reserved Words

μPerm's reserved words are:

```
and         or          xor         not
mod
cr          ff          nl          qt          tb
in          subset      sizeof
if          then        else        elsif
begin       end         print
loop        break       continue    for         while
function    procedure   return
```

1.2.8 Operators

μPerm’s operators are:

```
+              -              *              /              **
&              |              ~              ..
:=
=              <=             >=
+=             -=             *=             /=
```

Also, permutations may be composed by juxtaposition.

1.2.9 Comments

Comments begin with `--` and continue to the end of the line.

2 Types

2.1 Primitive types:

The primitive types are *Boolean*, *Integer*, *String*, *Set*, and *Permutation*. Their semantics are described below. *μ*Perm manipulates values of each type internally using a Java class that implements the semantics of the type. The type of a μPerm object is determined statically upon declaration and first assignment, which *must* coincide. Coercion between types may be allowable and is decided at runtime according to the rules described herein.

2.1.1 Boolean

The Boolean operators are:

And	**`and`**
Or	**`or`**
Xor	**`xor`**
Unary negation	**`not`**

Boolean values are represented internally using Java's *Boolean* class.
Boolean values may be coerced to integer values with **`true`** mapping to **`1`** and **`false`** mapping to **`0`**, or to upper-case strings.

2.1.2 Integer

The integer operators are

Addition	**`+`**
Subtraction	**`-`**
Multiplication	**`*`**
Division	**`/`**
Modulus	**`mod`**
Equality	=
Inequality	!=
Negation	**`-`**
Exponentiation	**`**`**

Integers are represented internally using Java's *bigint* package.
Integer values may be coerced to Boolean values with 0 mapping to false and all other values mapping to true. Integers may also be coerced to strings and singleton sets.

2.1.3 String

String operations are:

Concatenation	+
Assignment	:=
Equality	=
Inequality	!=

Strings are represented as Java Strings.
The empty string is denoted **`""`**. Characters are strings of length 1.

Strings also admit to array-style indexing, and the built-in functions **`length() and print()`**. To wit:

```
s := "Hello, uPerm!";
print s[2]; -- prints e.
a = length(s); -- a is now 13.
```

2.1.4 Set

A variable of set type represents a set of positive integer values.

Sets admit to the following operations:

Assignment	**`:=`**
Union	**`|`**
Intersection	**`&`**
Complement	**`~`**
Integer addition to	**`+`**
Integer removal from	**`-`**
Equality testing	**`=`**
Element testing	**`in`**
Subset testing	**`subset`**

The **`sizeof`** built-in function returns an integer representing the cardinality of the set.

A μPerm programmer can iterate a set using the **`in`** keyword, viz:

```
for i in {1,2,5,6,9} loop
   visit(i);
end loop;
```

Also, the **`in`** keyword can help with element testing:

```
if 1 in {3,4,5,9} then
   print "We have unity.";
end if
```

The reserved word **`subset`** can be used for subset testing:

```
if {5,6} subset {5,6,7} then
   print "Good enough.";
end if;
```

Two sets are equal if they have exactly the same elements.

A range of values may be assigned to a set using the range operator **`..`**, viz

```
S = {1 .. 10};
```

Looping is thusly accomplished.

An integer value can be coerced into a singleton set by enclosing it in curly braces, viz. **`{i}`**, hence

`S + i` is a sugary **`S | {i}`**.

2.1.5 Permutation

Permutations admit to

▪ Assignment	:=	
▪ Equality testing	=	
▪ Composition	**`*`**	(or juxtaposition)
▪ Inversion	~	

Permutations also admit to a function invocation, which reflects the action of the permutation on a member of its underlying set. Permutation invocation returns an integer value. Hence:

```
p := (1 2 5 8 3);
p(5);
```

returns 8, the action of **`p`** on **`5`**

2.2 Composite Types

µPerm provides a single composite type, *Array*.

2.2.1 Arrays

*µ*Perm provides very simple array semantics. Arrays may be declared by specifying an identifier for the array, and the array size.

```
id[10];
```

Apart from subprogram invocations, array declaration is the only *µ*Perm statement that does not include an operator of some sort.

Arrays hold references to primitive types, hence

```
p := (1 4 3 5 2);
i = 10;
a[2];
a[1] = p;
a[2] = i;
```

is a valid program. Notice that arrays are indexed from **`1`**, not **`0`**.

2.3 References

The underlying representation of all *µ*Perm objects of primitive type is an object of underlying Java type *Reference*. *Reference* has a subtype for each primitive type, determined and instantiated during parsing by variable declaration and assignment. Coercions are performed upon assignment only. Assignment to an array index coerces every reference subtype to *Reference*.

3 Expressions

3.1 Boolean Expressions

Boolean expressions consist of identifiers or literals related together using the Boolean operator keywords: **`and`**, **`or`**, **`xor`**, **`not`**. Order of operations and grouping are typical of the Boolean domain.

3.2 Integer Expressions

Integer expressions consist of identifiers or literals related together using unary and binary integer operators and the modulus operator, expressed with the reserved word **`mod`**. The modulus operator has multiplicative precedence.

3.3 Array Indexing Expressions

Array indexing expressions consist of an identifier followed by a bracket-enclosed integer expression. All array (index) expressions resolve to integer type.

3.4 String Expressions

String expressions consist of identifiers or string literals related together using the concatenation, equality, and inequality operators.

3.5 Set Expressions

Set expressions consist of identifiers or set literals related together using the operators **`=`** (equality), **`!=`** (inequality), **`|`** (union), **`&`** (intersection), **`in`** (element evaluation), and **`subset`** (subset evaluation).

3.6 Permutation Expressions

Permutations expressions consist of identifiers or literals related together using composition (whitespace or **`*`**), equality, or inequality. Permutation identifiers also admit to a built-in functional notation wherein the identifier is followed by a parenthetical single integer; these expressions are called *permutation evaluations*, and have integer type.

3.7 Semantics

Each expression has a type. It denotes the final computational type of the evaluated expression. See the Types section.

4 Statements

µPerm provides simple statement syntax and semantics.

4.1 Assignment and Declaration Statements

Assignment statements take the form

```
lhs := rhs;
```

Upon reading an assignment statement, μPerm evaluates the type and value of the right hand side **rhs**. It then locates the left hand side variable lhs in the symbol table, creating a new symbol and assigning **rhs** if one does not already exist. If **lhs** exists and is of the same type as **rhs**, the assignment is completed (if **rhs** also has an entry in the symbol table, a *full copy* is completed, not a reference copy). A coercion is required if **lhs** and **rhs** are not of identical type. If the coercion is illegal, μPerm halts execution, frees all memory allocated for the evaluation of **rhs**, outputs an error message. If the coercion is permitted, μPerm performs it and completes the assignment. If **lhs** is an array index expression, then the *Reference* to **rhs** is assigned.

4.2 If Statements

If statements have the general form:

```
if expression1 then
   --sequence of statements
elsif expression2 then
   --sequence of statements
else
   --sequence of statements
end if.
```

The **expression$_i$** must evaluate or be coerced to a Boolean value. The **elseif** and **else** portions are not required. The **if**, **then**, and **end if** portions are.

4.3 Loop Statements

Loop statements may take one of three forms:

```
loop
   -- sequence of statements
end loop


for <identifier> <iteration-scheme> loop
   -- sequence of statements;
end loop;


while expression loop
   -- sequence of statements
end loop
```

The first form produces an infinite loop.

The second form iterates *identifier* through certain integer values according to the *iteration-scheme*, such as that provided by sets and permutations. To visit the values 1 through 10, we might write:

```
for i in {1 .. 10} loop
   visit i;
end loop;
```

The third form is a typical while loop.

Upon execution of a **`break`** statement, μPerm exits the innermost loop of execution. Upon execution of a **`continue`** statement, μPerm jumps immediately to the next iteration of the loop.

5 Subprograms

5.1 Blocks

A μPerm block is a sequence of statements enclose by the **begin** and **end** keywords. A block causes a stack push and has its own lexical scope.

5.2 Functions

A function is a subprogram that returns a value of a primitive type. The syntax for declaring a function is:

```
function identifier (argument-list)
begin
   -- sequence of statements, with return statement
end
```

The return statement must specify a value of local scope to be returned (by value) to the calling statement.

5.3 Procedures

A procedure is a subprogram that does not return a value.

```
function identifier (argument-list)
begin
   -- sequence of statements, with return statement
end
```

The return statement must not specify a value.

5.4 Value-Passing Semantics

All values are passed into functions and procedures using their *Reference*.

6 Errors

Lexical, syntactical, and semantic errors will result in the termination of execution, immediate return from any function or procedure, and the output of an error message. Where possible, the error message will contain a reference to the appropriate section of this manual to assist the programmer in debugging.