

# Maxwell Language Design

F. Schuiki (fschuiki@student.ethz.ch)

February 21, 2014

## Contents

<b>1 Sequential Expressions aka Blocks</b>	<b>1</b>
1.1 Future Work . . . . .	1
<b>2 Functions</b>	<b>1</b>
2.1 Syntax . . . . .	1
2.2 Bound Arguments . . . . .	2
2.3 Contextual Calls . . . . .	2
<b>3 Miscellaneous</b>	<b>2</b>
3.1 Array Literals . . . . .	2
3.2 Iterators . . . . .	2

```
// implicit return value
// a = 2, Real
var a = { var k = 4; sqrt(k); };

// explicit single return value
// b = 2, Real
var b = r { r = sqrt(4); };
var b = r Real { r = sqrt(4); };

// explicit tuple return value
// c = (2,4), (Real,Real)
var c = (r,s) { r = 2; s = r+2; };
var c = (r Real, s Real) { r = 2; s =
    r+2; };
```

Listing 1: Sequential Expression examples

## 1 Sequential Expressions aka Blocks

Since everything is an expression in Maxwell, there are no statements. Blocks `{ ... }` in classic C-like languages are statements, i.e. they describe a computational step and do not have a value. In Maxwell, blocks also have a return value, depending on the syntax used for their definition. The following possibilities exist:

*Implicit return value.* In its simplest form, a block in Maxwell looks the same as in C. A list of semicolon-separated expressions enclosed in curly braces. The return value corresponds to the last statement in the block. `{ expr0; expr1; ... exprn; } = exprn;`

*Explicit single return value.* The block expression may be prefixed by a return variable name which may be assigned a value inside the block body. The variable may be used like any other variable or function argument, and will be returned as the block's value. `r { ... r = exprn; ... } = r;`

*Explicit tuple return value.* Instead of a single return variable, the block expression may define a tuple of variables that will be returned as value. `(r,s) { ... r = exprn; s = exprm; ... } = (r,s);`

Refer to listing 1 for an example of each type of sequential expression.

### 1.1 Future Work

It might actually be possible to unify the definition of variables, function arguments and block return

```
-> expr // no arguments
a0 -> expr // one argument
(a1,a2) -> expr // multiple arguments
```

Listing 2: Function definition syntax

variables into one entity.

## 2 Functions

### 2.1 Syntax

Functions represent the core concept of the Maxwell programming language. They map a set of input arguments to an output expression. This expression can be anything from a simple arithmetic operation to a complex algorithm. Unlike most members of the C family of languages, functions in Maxwell are anonymous and need to be assigned to a variable or given a name in order to be called. The syntax for a function definition looks as shown in listing 2.

Functions are objects that may either be defined in global scope (as you would do in classic C), or that may be assigned to a variable which may be called later. The two possibilities are outlined in listing 3.

Note that the concept of return types the reader might be familiar with from the C language, arises from Maxwell's sequential expression. By using a sequential expression as the function's expression, any number of return types may be realized. Refer

```
square: x -> x*x;
var f = x -> x*x;
// callable as square(...) and f(...)
```

Listing 3: Giving names to functions

```
square: x -> x*x
fibonacci: (x,y) -> x+y
invsquare: x -> r { r = x*x; r = 1/r; }
swapinc: (x,y) -> (s,t) { s = y+1; t = x+1; }
```

Listing 4: Function return values

to the example in listing 4.

## 2.2 Bound Arguments

Maxwell shall support the notion of Bound Arguments. That is, certain arguments of a function may be set to a constant value, thus yielding a new function which lacks those arguments. This concept is similar to Haskell's *currying*. This feature gives rise to contextual function calls (which C++ calls *member function calls*) and encourages software design that relies on callback functions, as well as functional design in general. The concept is illustrated in an example in listing 5.

## 2.3 Contextual Calls

Consider C++'s member functions. They are functions which have an implicit function argument representing the object they are called on. For example `A a; B b; a.add(b);` is restructured to `A::add(&a,b)`, implicitly passing `a` as the first function argument. A similar concept is possible in Maxwell, yet in a more generic fashion. Typing `var x = a.add(b);` in Maxwell encompasses an argument bind as well as a function call. The expression `a.add` is restructured to `add.bind(0,a)`, thus creating a new function which has its first argument bound to `a`. Of course this expression by itself may already be assigned to a variable or given a name in order to call it later. Yet in this example the call expression `... (b)` immediately calls the function and

```
add: (a Int, b Int) -> a+b;
var increment = add.bind("b", 1);
// add is a mapping (Int, Int) -> (Int)
// increment is now a mapping (Int) -> (Int)

var x = add(1,2); // x = 3
var y = increment(1); // y = 2
```

Listing 5: Example of function argument binding

```
// The following are equivalent:
a.add(b);
add(a,b);
add.bind(0,a)(b);

// Callback functions are easily
// defined:
var cb = a.add;
cb(5); // actually calls add(a,5)
```

Listing 6: Contextual call examples

```
type Iterator interface {
  next (iterator * @) -> nil;
  get (iterator @) -> any;
  end (iterator @) -> Bool;
}
```

Listing 7: Iterator interface definition

returns its result. See listing 6 for an example.

## 3 Miscellaneous

This section acts as a placeholder for random information.

### 3.1 Array Literals

An array literal is a constant, static array defined in source code. It is not intended to be modified and supports only a rather simplistic interface:

- `get (index Int) -> A`
- `length -> Int`

This corresponds to the *ConstArray* interface, which declares a function to access individual elements, and a function to obtain the overall length of the array.

### 3.2 Iterators

An iterator is a structure which walks through elements of a collection; i.e. an array, set, map or string. Walking through a linear collection such as a vector array is simple, as the elements may be accessed by a continuously increasing index. In case of more complex structures, such as sets or maps, stepping from one element to another is not as simple anymore. Iterators provide a way to abstract this operation, allowing each collection to implement their most efficient iteration technique.

The language shall define an *Iterator* interface as shown in listing 7. The *next* function advances the iterator to the next element, or does nothing if the iterator went past the last element of its collection.

The *get* function provides access to the element the iterator is currently pointing at.

## Listings

1	Sequential Expression examples . . .	1
2	Function definition syntax . . . . .	1
3	Giving names to functions . . . . .	2
4	Function return values . . . . .	2
5	Example of function argument binding	2
6	Contextual call examples . . . . .	2
7	Iterator interface definition . . . . .	2