KL Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

List of Examples

Chapter 1. Introduction

KL (pronounced *kale*) is the programming language used for Fabric operators. KL stands for “kernel language”; in this context, *kernel* refers to the concept of a computational kernel as used in multithreaded programming.

KL was designed with the following goals:

- KL is designed to run *performance-critical* sections of *dynamic-language* programs. As such, KL is designed so that KL programs run as quickly as possible on modern hardware.
- KL should be easy to learn for someone who is already familiar with programming in JavaScript and other scripting languages.
- It must be possible to compile most KL programs to run on different architectures and kinds of hardware, specifically CPUs and GPUs.

As such, KL is a language with a syntax very similar to JavaScript but that is *procedural*, *strongly-typed* and with *low-level data layouts*. Being *procedural* means that, unlike JavaScript, functions (or, rather, closures) are not first-class objects that can be passed around in the language; instead, functions are always globally declared. Being *strongly-typed* means that, in a KL program, the types of all variables and function parameters are known at compile type, unlike JavaScript where types are only known at runtime. Having *low-level data layouts* means that the size of data and the way that it is laid out in memory is guaranteed and controllable by the programmer.

Before diving in to the details of the language, we will present some simple examples. We don't expect you to understand all the details of the examples, but hopefully they will give you a basic idea of what KL programs look like as well as some of what is possible with KL. All of these examples can be run by downloading and installing the `FabricEngine-KLTool-...` file that is appropriate for your operating system from `http://dist.fabric-engine.com/latest/`. Once installed, you can copy the source code to a text file and run `kl filename.kl`.

1.1. Hello World in KL

Traditionally, the first program you see in a language is one that says hello. In KL, the "Hello, world!" program is simple:

Example 1.1. Hello world

```
operator entry() {
  report("Hello, world!");
}
```

Running this program through the `kl` tool, the output is:

```
Hello, world!
```

There's not much to learn from this program, but it does introduce a few concepts:

- The `entry` operator is what is run by the `kl` tool. KL distinguishes places in the program that can be called from the outside world—in this case, the outside world is the `kl` tool itself. We discuss more about operators in Section 4.6, “Operators”
- The `report` function sends some text to wherever messages go. In the case of the `kl` tool, messages go to standard output; when running within a web browser, messages appear on the JavaScript console; for console applications (in Node.js or Python for example), messages go to standard error.

- The text `"Hello, world!"` is a *string constant*. String constants in KL follow almost exactly the same syntax as JavaScript.

1.2. Fibonacci Sequence in KL

Next, we present a somewhat more sophisticated example that computes the first few terms of the Fibonacci sequence. The Fibonacci sequence is the number sequence whose first few terms are 1 and 1, and whose further terms are the sum of the previous two in the sequence, ie. 2, 3, 5, 8, and so on. There are several ways of computing the Fibonacci sequence in a programming language but we choose the naive, recursive way in order to illustrate some more language features of KL.

Example 1.2. Fibonacci sequence generator

```
/* Recursively compute the Fibonacci sequence.
** The first term is returned with n = 0
*/
function Integer fibonacci(Integer n) {
  if (n <= 1)
    return 1; // The first two terms (n=0 or n=1) are 1
  else
    return fibonacci(n - 2) + fibonacci(n - 1);
}

operator entry() {
  for (Integer i = 0; i < 10; ++i)
    report(fibonacci(i));
}
```

Running this program through the `kl` tool, the output is:

```
1
1
2
3
5
8
13
21
34
55
```

The Fibonacci example highlights a few simple features of KL, including:

- Function and parameter declaration
- Recursion
- Conditional statements
- Loops
- Comments

1.3. Mandelbrot Set in KL

Finally, we present an example of a KL program that generates the Mandelbrot set and outputs it visually as ASCII art. The Mandelbrot set is a mathematical set defined by complex recursive functions, but that it best known be-

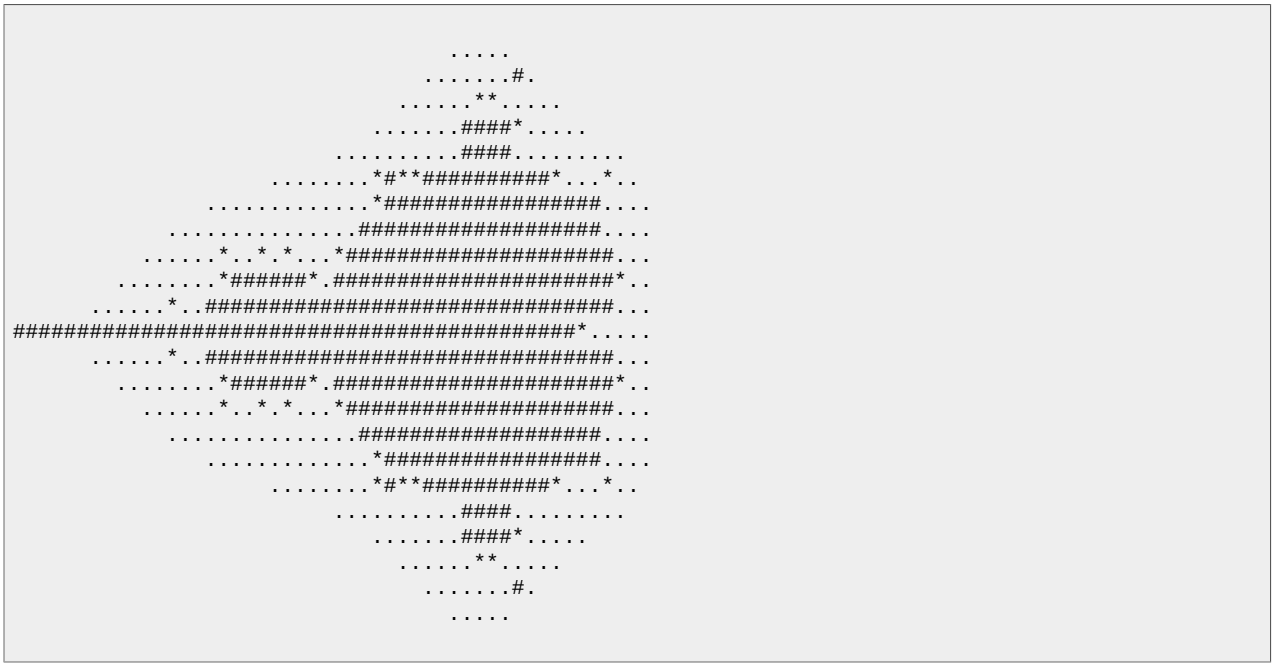
cause the patterns it contains are visually stunning. Refer to the Wikipedia entry on the Mandelbrot set at `http://en.wikipedia.org/wiki/Mandelbrot_set` for more information.

KL includes very powerful features for extending the language for computational problems such as that of computing the values of the Mandelbrot set, as seen in the source code below:

Example 1.3. Mandelbrot set generator

```
struct Complex32 {
  Float32 re;
  Float32 im;
};

function Complex32(Float32 re, Float32 im) {
  this.re = re;
  this.im = im;
}

function Complex32 +(Complex32 lhs, Complex32 rhs) {
  return Complex32(lhs.re + rhs.re, lhs.im + rhs.im);
}

function Complex32 *(Complex32 lhs, Complex32 rhs) {
  return Complex32(lhs.re*rhs.re-lhs.im*rhs.im, lhs.re*rhs.im + lhs.im*rhs.re);
}

function Float32 Complex32.normSq() {
  return this.re*this.re + this.im*this.im;
}

function Byte computeDwell(Complex32 c) {
  Complex32 z = c;
  Size count;
  for (count = 0; count < 255; ++count) {
    if (z.normSq > 4)
      break;
    z = z*z + c;
  }
  return count;
}

operator entry()
{
  Complex32 z;
  for (Size row=9; row<=31; ++row) {
    z.im = 4.0 * row / 40.0 - 2.0;
    String rowString;
    for (Size col=0; col<=78; ++col) {
      z.re = 4.0 * col / 78.0 - 2.0;
      Byte dwell = computeDwell(z);

      if (dwell & 192)
        rowString += "#";
      else if (dwell & 48)
        rowString += "*";
      else if (dwell & 12)
        rowString += ".";
      else
        rowString += " ";
    }
    report(rowString);
  }
}
```

Running this program through the `kl` tool, the output is:



The Mandelbrot set example highlights a few of the more sophisticated features of KL, including:

- User-defined types
- Constructors
- Operator overloads
- Methods
- Bitwise operators
- Variable-length arrays

1.4. How This Guide is Organized

Unlike other languages, KL is a language designed to add performance-critical sections to programs written in dynamic languages. As such, this guide assumes that you are already familiar with the basic concepts of modern programming languages, including types, expressions, statements (eg. conditional statements, loop statements, return statements) and functions.

Since it's difficult to provide meaningful examples in a computer language without using many concepts of the language, we will often use or even refer to concepts that aren't discussed until later in the guide. We leave it up to you to decide whether to jump ahead or to just keep reading and see where things go.

The remainder of the guide is laid out as follows:

- Chapter 2, *KL Syntax* describes the syntax of KL. KL has a syntax very similar to JavaScript (like JavaScript, it is a “C-like” language) so this syntax will be familiar to many programmers. If you are already familiar with the syntax of JavaScript or C, we recommend that you skip this chapter and move on to the next; you can always come back to it later for reference.
- Chapter 3, *The KL Type System* introduces the type system of KL. KL comes with a rich set of base types that includes integers, floating-point numbers, strings and booleans, as well as derived types including structures, arrays and dictionaries.

- Chapter 4, *Functions and Other Global Declarations* describes how functions and other global declarations are made in KL. KL includes support for basic functions (also referred to as procedures in other languages), but also support more complex function concepts such as prototypes, methods and overloaded operators. It also supports global constants, as well as a facility to import functionality from external modules.

- Chapter 5, *Operators and Expressions* describes the set of operators supported by KL and the different kinds of expressions that can be formed using these operators. It also discusses the rules that govern scoping for variables and other symbols.

- Chapter 6, *Statements* describes the different statements that are available for function bodies in KL.

Chapter 2. KL Syntax

When KL was developed it was designed to have a syntax as close as possible to JavaScript, which itself is what is referred to as a "C-like language" (as far as syntax is concerned). This chapter goes into the details of KL syntax, and can act as a reference for you if you are not familiar with C-like languages. On the other hand, if you are already familiar with the syntax of JavaScript and/or C, we recommend that you skip this chapter and move on to the next.

Some of the key characteristics of C-like languages are:

- Programs are plain, human-readable text files.
- A program is a sequence of *tokens*. There are four major types of tokens: keywords, identifiers, symbols and constants.
- Tokens may be separated by arbitrary whitespace and *must* be separated by whitespace in the case that it would make two adjacent tokens appear to be a single, different token. By whitespace, we mean spaces (ASCII 32), newlines (ASCII 10), tabs (ASCII 9), carriage returns (ASCII 13) and vertical tabs (ASCII 11).
- Programs can contain comments anywhere and comments are treated like whitespace when the program is processed by the computer. There are two types of comments: block comments, which begin with `/*` and end with `*/`, and line comments, which begin with `//` and continue to the next newline (ASCII 10).
- Blocks are delimited by `{` and `}`
- Statements are terminated with `;`

We delve into some more details of the KL syntax below.

2.1. Encoding

Although KL programs are plain, human-readable text files, KL does specify an encoding for the text files: UTF-8. The KL compiler assumes that all KL source code is encoded as UTF-8 without any encoding marks. KL programs which contain invalid UTF-8 sequences cannot be compiled and will result in syntax errors.

If you don't understand exactly what this means, don't worry: plain 7-bit ASCII text files, such as used for C source code (and often for JavaScript source code) are by default UTF-8 encoded. The only thing to keep in mind is that text files saved on Windows in "Unicode format" (which technically means they are encoded as UCS-2) cannot be read by the KL compiler. If you want to insert foreign language characters into a KL source file you must use an editor that can write UTF-8 files.

The benefit of specifying an encoding is that high-bit characters, such as foreign language characters, can be inserted directly into string constants in KL source files.

2.2. Line Continuations

Like C, any line of a KL program that ends with a `\` (backslash) character causes the line to be joined with the next line. This is useful for breaking long lines into multiple lines when a single token needs to be split; usually this token would be a long string.

Example 2.1. Line continuations

```
operator entry() {
  report("\
This is a really, really long string \
that spans multiple lines.  Note that \
     <-- the SPACES at the beginning \
of the next line are preserved!");
}
```

Output:

```
This is a really, really long string that spans multiple lines.  Note that      <-- the SPACES at
 the beginning of the next line are preserved!
```

2.3. Comments

As mentioned previously, KL supports the same two types of comments as C and JavaScript: block comments and line comments.

2.3.1. Block Comments

A block comment in KL is an arbitrary sequence of characters that begins with the characters `/*` and ends with the characters `*/`. Like C, KL ends a block comment as soon as it encounters the first sequence `*/`. Also like C, KL does not recognize nested comments. The following examples illustrate acceptable and erroneous bad block comments in KL.

Example 2.2. Acceptable block comments

```
/* A simple, one-line block comment */

/* This is
   a multi-line
   block comment
   */

/*
** And so is this
*/

operator entry() {
  /* comments can appear in source code */
  report("Hello!"); /* pretty much anywhere */
  foo( 32 /*, 53*/ ); /* block comments are a simple way to temporarily remove code */
}
```


Example 2.3. Erroneous block comments

```
/* KL ends the comment
 * as soon as the first */
 * is seen, so this is
 * a bad block comment that
 * results in a syntax error
 */

/* Like C, block comments
 * cannot nest in KL,
   /* so this is also
    * a bad comment that
    * will result in a
    * syntax error
    */
 */
```

Be careful with block comments; if either of the errors of the type given above occur in your program it can be very difficult to figure out what has gone wrong; the syntax errors you are given will seem to have nothing to do with your program. For this reason, we recommend that you use line comments, which are described next.

2.3.2. Line Comments

A line comment in KL begins with the character sequence `//` and continues until the end of the line (ASCII character 10). Unlike block comments, there are no “gotchas” with line comments: they work exactly as expected. However, they can't be used to comment out sections of code in the middle of a line

Example 2.4. Line comments

```
// A simple, one-line line comment

// This is
// a multi-line
// line comment

operator entry() {
  // comment can appear in source code
  report("Hello!"); // pretty much anywhere
  foo( 32 //, 53 ); // can't comment out code in the middle of the line
          ); // but you can continue it on the next line.
}
```

2.4. Tokens

A KL program is *parsed* as a sequence of tokens. A token is a sequence of characters that begins with something other than whitespace and is not a comment. There are four categories of tokens in KL: keywords, identifiers, symbols, and constants.

2.4.1. Keywords

The following is the list of all the keywords in KL:

```
alias               do                  in                  switch
```

```
break       else        io          throw
case        false       operator    true
const       for         require     var
continue    function    return      while
default     if          struct
```

Keywords cannot be used as identifiers, ie. cannot be used for variable, parameter, function, constant or type names.

2.4.2. Identifiers

Identifiers are a sequence of one or more characters that:

- begin with any of the characters a...z, A...Z or _; and
- are followed by zero or more of the characters a...z, A...Z, 0...9 or _; and

Identifiers are used in KL for variable names, parameter names, function names, constant names, method names and type names.

The built-in base types in KL (eg. `Boolean`, `String`, `Float32`) are not keywords but rather identifiers. This is a technical detail that is important in the design of the language grammar but doesn't make much difference in practice; it simply changes certain syntax errors into semantic errors.

Some examples of valid identifiers in KL:

- `foo`
- `someVariable`
- `MyType`
- `MyTypeVersion2`
- `variable_with_underscores`
- `piBy2`
- `_`

Some examples of invalid identifiers:

- `2by4`
- `my%Share`
- `peter.zion@fabric-engine.com`

2.4.3. Symbols

A small set of non-alphanumeric, non-underscore characters or short sequences of such characters are the symbols in KL. They are specifically:

```
=       -=      %       &=      ]       .       >=      ,
==      --      %=      &&      (       <       >>      ?
+       *       ^       |       )       <=      >>=     ;
+=      *=      ^=      |=      {       <<      ~
++      /       ^^      ||      }       <<=     !
```



```
-       /=      &       [       ;       >       !=
```

The KL compiler is “greedy” when looking for symbols: it looks for the longest sequence of characters that are a valid symbol. This is why `==(` is treated as the two symbols `==` and `(`: there is no symbol starting with `==(` but `==` is a symbol.

2.4.4. Constants

A constant is a token that is interpreted as a fixed value of a specific type. KL supports five types of constants: boolean constants, integer constants, floating-point constants, and string constants. Each of these are explained below.

2.4.4.1. Boolean Constants

The two keywords `true` and `false` are the two boolean constants. They are each a value of type `Boolean`.

Example 2.5. Boolean constants

```
operator entry() {
  Float32 value = 1.1;
  Index steps = 0;
  Boolean done = false;
  while (!done) {
    value *= value;
    if (value > 2)
      done = true;
    ++steps;
  }
  report("took " + steps + " steps");
}
```

Output:

```
took 3 steps
```

2.4.4.2. Integer Constants

A integer constant can take one of three forms:

- The single digit `0`
- A decimal constant that starts with a single digit in the range `1`...`9` followed by zero or more digits in the range `0`...`9`
- The two characters `0x` or `0X` followed by one or more characters in one of the ranges `0`...`9`, `a`...`f` and `A`...`F`

The type of an integer constant depends on its value:

- If it is less than 2^{31}-1 then it is of type `Integer`;
- If it is at least 2^{31} but less than 2^{32}-1 (on 32-bit platforms) or 2^{64}-1 (on 64-bit platforms) then it is of type `Size`;
- Otherwise, the integer constant is too large and the compiler will generate a syntax error.

All integer constants are positive; a `-` (minus sign) followed by an integer constant is interpreted as the unary minus operator applied to the integer constant.

Example 2.6. Integer constants

```
operator entry() {
  report(0);
  report(1);
  report(123456789);
  report(0xA9);
  report(0xdeadBEEF);
}
```

Output:

```
0
1
123456789
169
3735928559
```

2.4.4.3. Floating-Point Constants

A floating-point constant takes the following form:

- Either a single `0` or one digit in the range `1`...`9` followed by zero or more digits in the range `0`...`9` (the *whole part*; the same syntax as a decimal integer constant)...
- ...optionally followed by a `.` and zero or more characters in the range `0`...`9` (the *fractional part*)...
- ...optionally followed by an `e` or an `E`, then optionally by a `+` or a `-`, then by one or more characters in the range `0`...`9` (the *exponent*).

C accepts floating-point constants that start with a `.` but floating-point constants in KL must start with a digit

However, for a constant to be floating-point (and not integer) it must have a fractional part, an exponent, or both.

It is not possible to represent many decimal fractional expressions in IEEE floating point format. As such, do not be surprised if the output values of such expressions appear to be slightly off, as seen in the example below with the constant `8.9e-6`.

In KL, all floating-point constants are of type `Float64`, regardless of value.

All floating-point constants are positive; a `-` (minus sign) followed by a floating-point constant is interpreted as the unary minus operator applied to the floating-point constant.

Example 2.7. Floating-point constants

```
operator entry() {
  report(0.0);
  report(1.4142);
  report(3.14159265358979);
  report(8.9e-6);
  report(4356.123E42);
}
```


Output:

```
0
1.4142
3.14159265358979
8.899999999999999e-06
4.356123e+45
```

2.4.4.4. String Constants

A *string constant* is a constant value of type `String`. It takes one of the following forms:

- A sequence of characters, possibly including string constant escape sequences (described below), enclosed in a pair of `"` (“double-quote”) characters; or
- A sequence of characters, possibly including string constant escape sequences (described below), enclosed in a pair of `'` (“single-quote”) characters; or

A *string constant escape sequence* is one of the following sequences of characters, describing what the sequence is replaced by in the string:

`\n` a single newline (ASCII 10) character

`\f` a single form feed (ASCII 12) character

`\r` a single carriage return (ASCII 13) character

`\t` a single carriage tab (ASCII 9) character

`\v` a single vertical tab (ASCII 11) character

`\a` a single bell (ASCII 7) character

`\b` a single backspace (ASCII 8) character

`\0` a single null (ASCII 0) character

`\"` a single double-quote (ASCII 34) character

`\'` a single single-quote (ASCII 39) character

`\\` a single backslash (ASCII 92) character

`\x`*`HH`* A single character whose ASCCI code is given in hexadecimal. The two characters *`HH`* must each be a hexadecimal character, ie. in one of the ranges `0`...`9`, `a`...`f` and `A`...`F`. For example, the escape sequence `0x0A` (decimal 10) is equivalent to the escape sequence `\n` (ASCII 10).

Example 2.8. String constants

```
operator entry() {
  report("double-quoted!");
  report('single-quoted\nwith a newline');
  report("double-quoted containing \"double quotes\"");
}
```

Output:

```
double-quoted!
single-quoted
with a newline
double-quoted containing "double quotes"
```

2.5. Program Structure

A KL program consists of zero or more *global declarations* in sequence. A global declaration is one of:

- A declaration of a function, a function prototype, an operator, a method or an operator overload; see Section 4.1, “Function Definitions”.
- A declaration of structure; see Section 3.2.1, “Structures”.
- A declaration of global constant; see Section 4.11, “Global Constants”.
- A `require` statement; see Section 4.12, “Importing Functionality With `require`”.

In turn, functions, operators, and other similar constructs consist of zero or more *statements*. Each statement is terminated with a `;` (semicolon) character. For more detailed information on all the statements available in KL, see Chapter 6, *Statements*.

A sequence of statements, surrounded by braces (a `{` and a `}`), can take the place of a single statement. This also introduces a nested scope; for more information, see Section 5.4.1.3, “Compound Statement Scope”.

Chapter 3. The KL Type System

Unlike most languages, KL has a dynamic type system that is inherited from the Fabric environment. In most cases, compound types are registered from the Fabric environment running in a dynamic language (eg. JavaScript or Python); those types are then automatically made available to KL programs running in that environment. This does not, however, affect the semantics of the language; the KL type system can still be explained purely from the point of view of the language itself.

Like most programming languages, KL has support for both a fixed set of base types from which other types are derived as well as different kinds of derived types.

3.1. Base Types

The base types in KL are the following:

`Boolean` can be either `true` or `false`

`Byte` an 8-bit unsigned integer

`Integer` a 32-bit signed integer

`Size` an unsigned integer that is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms. `Size` is usually used to either index into or count the number of elements in an array

`Index` an alias for `Size`

`Float32` a 32-bit IEEE floating point

`Scalar` an alias for `Float32`

`Float64` a 64-bit IEEE floating point

`String` a sequence of zero or more characters

3.1.1. The `Boolean` Type

The value of an expression of `Boolean` type is either logical true or logical false. The type has the following properties:

- The constants `true` and `false` are `Boolean` values with logical values true and false, respectively.
- All other base types cast to Boolean as follows:
  - `Byte`, `Integer` and `Size` values cast to true if and only if the value is non-zero
  - `Float32` and `Float64` values cast true if and only if the value is not equal to `0.0` or `-0.0`
  - `String` values cast to true if and only if their length is greater than zero
  - arrays and dictionaries cast to true if and only if they are non-empty
  - by default, structures do not cast to `Boolean`, but you can implement the cast if desired by creating a `Boolean` constructor that takes the structure as a parameter; see Section 4.7, “Constructors”
- For operators:
  - None of the arithmetic operators (binary `+`, `-`, `*`, `/`, `%` as well as unary `-` and `+`) are valid for `Boolean` values
  - Only the `==` and `!=` comparison operators are valid for `Boolean` values

- All of the bitwise binary operators (`|`, `&`, `^` and `~`) are valid for `Boolean` values and treat the value as if were a single bit

Example 3.1. The `Boolean` type

```
operator entry() {
  Boolean a = true;
  report(a);
  Boolean b = a & false;
  report(b);
  report(a != b);
}
```

Output:

```
true
false
true
```

3.1.2. Integer Types

The `Byte`, `Integer` and `Size` types (collectively known as *integer types*) represent whole integers. These types differ only in their bit width and whether they are signed or unsigned, as follows:

`Byte` an 8-bit unsigned integer

`Integer` a 32-bit signed integer

`Size` an unsigned integer that is 32-bit on 32-bit systems and 64-bit on 64-bit systems

`Index` an alias for `Size`

Integer types behave as follows:

- Integer constants (eg. `43562`) are of type `Integer` if their value is less than 2^{31}-1, otherwise they are of type `Size`. For more details, see Section 2.4.4.2, “Integer Constants”.
- All of the arithmetic, logical and bitwise operators work as expected for all integer types.

The result of applying the unary `-` (minus) operator to a `Byte` or a `Size` is still a positive number; it is the resulting unsigned integer that, when added to the original, results in 0 because of the type “wrapping around”.

Example 3.2. Integer types

```
operator entry() {
  Byte b = 64;
  report(b);
  Size s = 45 * Size(b) + 32;
  report(s);
  Integer i = -75 * Integer(s) + 18;
  report(i);
}
```


Output:

```
64
2912
-218382
```

3.1.3. Floating-Point Types

The `Float32`, `Float64` and `Scalar` types (collectively known as *floating-point types*) represent IEEE floating-point numbers. These types differ only in their bit width, as follows:

`Float32` a 32-bit IEEE floating-point number

`Float64` a 64-bit IEEE floating-point number

`Scalar` an alias for `Float32`

The `Scalar` alias will probably be removed in a future version of KL. We recommend that you not use it in any new code.

Floating-point types behave as follows:

- Floating-point constants have the same syntax as in JavaScript and C, and are of type `Float64`. For more details, see Section 2.4.4.3, “Floating-Point Constants”.
- All of the arithmetic and comparison operators are valid for floating-point values. None of the bitwise operators are valid for floating-point values.

Example 3.3. Floating-point types

```
operator entry() {
  Float32 x = 3.141;
  report(x);
  Float64 y = 2.718;
  report(y);
  Float32 z = x*x + y*y;
  report(z);
}
```

Output:

```
3.141
2.718
17.2534
```

3.1.4. The `String` Type

The `String` type represents a text string, ie. a sequence of zero or more characters. A value of type `String` is referred to as a *string value*.

The semantics of the `String` type in KL are important to understand. Strings have the following key properties:

- A string is a sequence of zero or more characters.

- The length of a string is value of type `Size`, and the maximum length of a string is the maximum value of type `Size`.
- String constants can be specified inline in KL source files using single- or double-quotation marks, just as JavaScript. For more details and examples of string constants, see Section 2.4.4.4, “String Constants”.
- Strings are *reference-counted* and *copy-on-write*. This means that if you assign a string value to multiple variables, there is only one copy of the actual string until someone changes one of the strings.
- Strings support the following operations and properties, which are all exactly the same as JavaScript:
  - They have a .length property which returns the number of characters in the string
  - The `+=` assignment operator is used to append another string to a given string
  - A new string can be created by concatenating two other strings using the `+` binary operation
  - Strings can be compared using the usual `==`, `!=`, `<`, `<=`, `>` and `>=` logical operators.
- Unlike C or C++, strings can contain the null character (ASCII 0).
- Strings have no notion of encoding; they are just sequences of bytes. String encodings are determined by the application space where the strings are used. Note that everything in Fabric itself uses the UTF-8 encoding, but Fabric extensions may need to convert strings into other encodings.
- All other types in KL can be converted to strings through a cast; this conversion simply creates a string that is a human-readable version of the value.

Example 3.4. String type

```
operator entry() {
  String a = "A string";
  report(a);
  report("a has length " + a.length);
  String b = "Another string";
  report(b);
  String c = a + " and " + b;
  report(c);
  b += " now includes " + a;
  report(b);
}
```

Output:

```
A string
a has length 8
Another string
A string and Another string
Another string now includes A string
```

3.2. Derived Types

In addition to the base types, KL supports three classes of derived types: structures, arrays and dictionaries.

3.2.1. Structures

A *structure* is a collection of typed values that are placed together in memory.

Structures are usually defined outside of KL using Fabric's *registered type system*, but they can also be declared in KL source code itself using the `struct` keyword:

```
struct NewType {
  Float32 firstMember;
  String secondMember;
  Integer thirdMemberVarArray[];
};
```

Note the use of the variable-size array as the last member; derived types can nest arbitrarily.

All structure declarations in KL must be in the global scope; it is not possible to declare a structure within a function scope.

More details about structures:

- Access to structure members is through the `.` (dot) operator, as in JavaScript.
- Currently, the structure members are *packed*, meaning that there is no space in member between the structure members. In a future version of Fabric it will be possible to explicitly specify structure member alignment and packing.
- It is possible to overload operators and add *methods* to structures; see Section 4.9, “Methods”.

Example 3.5. Structure use

```
struct MyNewType {
  Integer i;
  String s;
};

function entry() {
  MyNewType mnt;
  mnt.s = "Hello!";
  mnt.i = 42;
  report(mnt);
}
```

Output:

```
{i:42,s:"Hello!"}
```

3.2.2. Arrays

An *array* is a sequence of values of the same type (referred to as the array's *element type*) that are indexed by integers and placed sequentially in memory. KL supports three types of arrays: variable-size arrays, fixed-length arrays, and sliced arrays. The details of each array type are discussed below.

Regardless of specific type, arrays in KL have several common behaviours:

- Arrays are indexed using the `[..]` operator, exactly as in JavaScript and C. The indexing of arrays is 0-based, again just as in JavaScript and C.

```
```

```
Size sizes[];    // Declare a variable-size array
sizes.push(42);  // Push some elements onto the end of the array
sizes.push(21);
sizes.push(3);
report(sizes[1]); // outputs "21"
```

- The size of an array is of type `Size` and the indexing operator takes an index of type `Index` (which is an alias for `Size`).
- Array declarations can be nested, and can be co-nested with other array types.

```
Integer b[][];  // A variable-size array of variable-size arrays of integers
Boolean a[2][]; // An array of 2 variable-size arrays of booleans
String c<>[];   // A sliced array of variable-size
```

- Arrays are *passed by reference* into functions and operators, ie. they are not copied. This means that it takes just a long to pass an array with one million elements to a function as it does to pass an array with one element.
- When running within a web browser, indexing into arrays using the indexing operator is bounds-checked; if the index runs off the end of the array, an exception is thrown. When running Fabric from the command line, arrays are not bounds-checked.

3.2.2.1. Variable-Size Arrays

A *variable-size array* is an array whose size can be changed at runtime. Variable-size arrays are declared by appending `[]` to the name of the variable, parameter or structure member where they are declared, eg. `String strings[]`.

Variable-size arrays have all the properties of arrays(page 27) as well as the following additional properties:

- The maximum size of a variable-size array is the maximum value of the `Size` type. This means that variable-size arrays can be much larger on 64-bit machines than they can on 32-bit machines.
- Variable-size arrays are *duplicate-on-modify*, meaning that it is inexpensive to assign one variable-size array to another but as soon as one of the “copies” is modified the contents are duplciated.
- Variable-size arrays support the following methods:
  - The `push(element)` method appends an element to the end of the variable-size array
  - The `size()` method returns the number of elements in the variable-size array
  - The `resize(newSize)` method resizes the array. Any new elements at the end are initialized with the default value for the underlying type.

Example 3.6. Variable-size arrays

```
operator entry() {
  Integer a[];
  report("The array a has size " + a.size + " and value " + a);
  a.push(42);
  a.push(84);
  report("The array NOW has size " + a.size + " and value " + a);
  a.resize(4);
  report("The array NOW has size " + a.size + " and value " + a);
}
```


Output:

```
The array a has size 0 and value []
The array NOW has size 2 and value [42,84]
The array NOW has size 4 and value [42,84,0,0]
```

3.2.2.2. Fixed-Size Arrays

A *fixed-size array* is an array whose size is fixed at runtime. Fixed-size arrays have much faster performance characteristics than variable-size arrays, therefore should be used in place of variable-sized arrays when the size of an array is known at compile time. Fixed-size arrays are declared by appending `[size]` to the name of the variable, parameter or structure member where they are declared, eg. `String strings[4]`.

Fixed-size arrays have all the properties of arrays(page 27) as well as the following additional properties:

- The maximum size of a fixed-size array is the maximum value of the `Size` type.

Since fixed-size arrays are allocated on the stack (instead of the heap), using very large fixed-size arrays may result in a stack overflow. It is recommended that fixed-size arrays only be used for arrays that are reasonably small.

- Fixed-size arrays are copied when they are assigned.

Unlike variable-size arrays, this behaviour of fixed-size arrays will never change.

Example 3.7. Fixed-size arrays

```
function Float32 det(Float32 mat[2][2]) {
  return mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0];
}

operator entry() {
  Float32 mat[2][2];
  mat[0][0] = 3.5;
  mat[0][1] = -9.2;
  mat[1][0] = -2.1;
  mat[1][1] = 8.6;
  report("The determinant of " + mat + " is " + det(mat));
}
```

Output:

```
The determinant of [[3.5,-9.2],[-2.1,8.6]] is 10.78
```

3.2.2.3. Sliced Arrays

A *sliced array* is an array whose size is fixed when it is created. Sliced arrays have the unique feature that new sliced arrays can be created that access a sub-range of the elements of an existing sliced array. Sliced arrays are primarily used for operator parameters bound to sliced data inside Fabric's dependency graph, but can also be used on their own within

KL. Sliced arrays are declared by appending <> to the name of the variable, parameter or structure member where they are declared, eg. `String strings<>`.

Sliced arrays have all the properties of arrays(page 27) as well as the following additional properties:

- Within KL, sliced arrays can only be created using one of two syntaxes, shown in the example below:

```
Size sizes<>(8);                // Create a sliced array of 8 elements of type Size
for (Size i=0; i<8; ++i)
  sizes[i] = i;

Size subSizes<>( sizes, 2, 4 ); // A sliced array of elements of type Size that accesses
                                // elements 2 through 5 (inclusive) of the underlying
                                // sliced array
report(subSizes[0]);            // will output '2'
report(subSizes.size);          // will output '4'
```

- Sliced arrays as a whole cannot be the target of an assignment. However, an element of a sliced array (accessed through the `[...]` index operator) can be assigned to.
- Sliced arrays support the following methods:
  - The `size()` method returns the number of elements in the sliced array. If a sliced array refers to a subrange of another sliced array, the value returned is the size of the subrange.

Example 3.8. Sliced arrays

```
operator entry() {
  String strings<>(8);
  for (Size i=0; i<8; ++i)
    strings[i] = "string " + (i+1);
  report("strings = " + strings);

  String subStrings<>(strings, 2, 4);
  report("subStrings = " + subStrings);

  report("Replacing subStrings element...");
  subStrings[3] = "replaced string";
  report("subStrings = " + subStrings);
  report("strings = " + strings);
}
```

Output:

```
strings = ["string 1","string 2","string 3","string 4","string 5","string 6","string 7","string
 8"]
subStrings = ["string 3","string 4","string 5","string 6"]
Replacing subStrings element...
subStrings = ["string 3","string 4","string 5","replaced string"]
strings = ["string 1","string 2","string 3","string 4","string 5","replaced string","string
 7","string 8"]
```

3.2.3. Dictionaries

KL supports key-value pair dictionaries. The type of the key of a dictionary can be any of the KL base types (e.g. `Boolean`, `String`, or any integer or floating-point type) and the type of the value can be any type. Dictionaries are declared by appending `[KeyType]` to the variable, parameter or member name. For example:


```
String scalarToString[Float32];     // A Float32-to-String dictionary
Boolean integerToBoolean[Integer];  // An Integer-to-Boolean dictionary
```

Dictionaries in KL have the following properties:

- Dictionaries are *duplicate-on-modify*, meaning that it is inexpensive to assign one dictionary to another but as soon as one of the "copies" is modified the contents are duplciated.
- Dictionaries can be nested, and can be co-nested with array types. For example:

```
Integer b[String][2]; // An String-to-Fixed-Length-Integer-Array dictionary
Boolean a[][Integer]; // A variable array of Integer-to-Boolean dictionaries
```

- There is no limit to the size of dictionaries other than available memory
- Dictionaries support indexing using the `[key]` indexing operator to both return the element at the give key or assign to the element at the given key. When retrieving a value from a dictionary, if there is no value for the given key then an exception is thrown.
- Dictionaries support the `has(key)` method that returns a `Boolean` value indicating whether there is a value in the dictionary for the given key.
- Dictionaries support the `delete(key)` method that deletes the value for the given key. If there is no value for the given key, nothing happens.
- Dictionaries can be iterated over using JavaScript-like `in` iteration:

```
String dict[String];
for (k in dict)
  report("dict[" + k + "] = " + dict[k]);
```

  For improved performance, both the key and value can be made available through `in` iteration. The value can be assigned to if and only if the dictionary can be assigned to. For example:

```
String dict[String];
for (k, v in dict)
  report("dict[" + k + "] = " + v);
```

- Insertion order (not sort order!) is the iteration order for dictionaries, just as for JavaScript objects.

```
operator entry() {
  String numbers[Integer];
  numbers[3] = "three";
  numbers[2] = "two";
  report(numbers); // outputs '{3:"three",2:"two"}'
  numbers[1] = "one";
  report(numbers); // outputs '{3:"three",2:"two",1:"one"}'
}
```

Example 3.9. Dictionaries

```
operator entry() {
  Float32 a[String];
  a['pi'] = 3.14;
  a['e'] = 2.71;
  report("a is:");
  for ( k, v in a ) {
    report("a['" + k + "'] = " + v);
  }
  a.delete('pi');
  report("a is now:");
  for ( k, v in a ) {
    report("a['" + k + "'] = " + v);
  }
}
```

Output:

```
a is:
a['pi'] = 3.14
a['e'] = 2.71
a is now:
a['e'] = 2.71
```

3.2.4. Map-Reduce Types

There are two additional derived types used exclusively for work within Fabric's map-reduce framework, namely:

- `ValueProducer<`*Type*`>`
- `ArrayProducer<`*Type*`>`

For more information, see Chapter 7, *Map-Reduce*.

3.3. Type Aliases

The `alias` statement can be used to alias a type to make code more readable. Its syntax is the same as a variable declaration:

```
alias Integer Int32;        // Int32 is now an alias for Integer
alias Float32 float;        // float is now an alias for Float32
alias Float32 Mat22[2][2];  // Mat22 is now an alias for Float32[2][2], ie. a size-2-array-of-
size-2-arrays-of-Float32
```

`alias` statements must appear within the global scope of a KL program.

Example 3.10. Type aliases

```
alias Float32 Mat22[2][2];

operator entry() {
  Mat22 mat22;
  report(mat22);
}
```

Output:

```
[[0,0],[0,0]]
```

3.4. The `Data` Type and the `data` and `dataSize` Methods

When interfacing with external libraries such as OpenGL, it is sometimes necessary to get direct access to the data underlying a value. An example is a library call that takes a pointer to data. KL itself has no notion of pointers; instead, KL has the concept of the `Data` type whose value is a pointer to data which can be passed to an external library call.

Most values in KL have a built-in method called `data` that returns a value of type `Data`, and a built-in method called `dataSize` that returns a value of type `Size`. The value returned by the `data` method is a pointer to the data underlying the value, and the value returned by the `dataSize` method is the number of bytes the value occupies in memory. The only values which do not support the `data` and `dataSize` methods are dictionaries as well as other derived types that do not lay out their elements or members contiguously in memory.

```
Integer integers[];
report(integers.data);  // OK: integers are contiguous in memory
String strings[];
report(strings.data);   // ERROR: string data is not contiguous in memory
```

Unlike pointers in C, the values returned by `data` methods cannot be inspected or used in any expressions; the only thing which can be done is a cast to `Boolean`, which will be `true` if and only if the `Data` value points to a value whose size is greater than zero. However, these `Data` values can be passed directly to external library functions provided by Fabric itself or Fabric extensions, where they are used as pointers to data in memory.

For values of type `String`, the value returned by `dataSize` includes a null terminator that is automatically appended to the string by Fabric; this is so that the string data can be directly used in C library calls as a regular C string. If you want to pass the number of characters in the string, pass `string.length` instead.

Example 3.11. `Data` values and the `data` and `dataSize` methods

```
operator entry() {
  String s;
  report("s = '" + s + "'");
  report("s.data = " + s.data);
  report("Boolean(s.data) = " + Boolean(s.data));
  report("s.dataSize = " + s.dataSize);
  s = "Hello";
  report("s = '" + s + "'");
  report("s.data = " + s.data);
  report("Boolean(s.data) = " + Boolean(s.data));
  report("s.dataSize = " + s.dataSize);
}
```

Output:

```
s = ''
s.data = <Opaque>
Boolean(s.data) = false
s.dataSize = 1
s = 'Hello'
s.data = <Opaque>
Boolean(s.data) = true
s.dataSize = 6
```


Chapter 4. Functions and Other Global Declarations

4.1. Function Definitions

Function definitions in KL are much the same as the “traditional” function definition syntax in JavaScript, with the following key differences:

- The return type and the type of each function parameter must be explicitly declared. If a function does not return a value, the return type must be omitted.
- The parameter declarations may additionally declare the parameter as input (read-only; the default) by preceding the type by `in` or input-output (read-write) by preceding the value by `io`.

Example 4.1. Function definitions

```
// Function returning a value and using only
// input parameters

function Float32 add(Float32 lhs, Float32 rhs) {
  return lhs + rhs;
}

// Function not returning a value and using both
// input and input-output parameters

function add(in Float32 lhs, in Float32 rhs, io Float32 result) {
  result = lhs + rhs;
}
```

4.2. Function Invocations

Function invocations (“calls”) are made using the same syntax as JavaScript, namely by appending a comma-delimited list of arguments, surrounded by parentheses, to the function name.

Example 4.2. Function call

```
function Integer add(Integer lhs, Integer rhs) {
  return lhs + rhs;
}

operator entry() {
  report("2 plus 2 is " + add(2, 2));
}
```

Output:

```
2 plus 2 is 4
```

4.3. Function Prototypes

A *function prototype* in KL is a function declaration that is missing a body. Providing a function prototype allows the function to be called before it is defined. This is useful under two circumstances:

- When two or more functions call each other. Such functions are sometimes referred to as *co-recursive*.

Example 4.3. Co-recursion using a prototype

```
// Function prototype for 'two', so that 'one' can call it before it is defined
function two(Integer n);

// The function 'one' calls 'two' even though it is not yet defined
function one(Integer n) {
  report("one");
  if (n > 0)
    two(n - 1);
}

// The definition of the function 'two' comes after its prototype
function two(Integer n) {
  report("two");
  if (n > 0)
    one(n - 1);
}

operator entry() {
  one(4);
}
```

Output:

```
one
two
one
two
one
```

- When a function definition is provided by a Fabric extension. The name of the symbol of the function in the Fabric extension is provided by appending `= "symbol name"` or `= 'symbol name'` to the function prototype.

Example 4.4. External function

```
// The prototype 'libc_perror' is linked to an external function 'perror'
function libc_perror(Data cString) = 'perror';

// The KL function 'perror' is what KL functions actually call
function perror(String string) {
  libc_perror(string.data);
}

operator entry() {
  perror("something that caused an error");
}
```


4.4. Built-in Functions

KL has several built-in functions that are available to all KL programs.

4.4.1. The `report` Function

The `report` function outputs a message to wherever messages are sent from KL; when KL is used from Fabric running in a browser this is to the JavaScript console, whereas when Fabric is used from the command line or when the KL tool is used the output is sent to standard error and standard output respectively. A newline is appended to the message when it is sent.

The prototype of the `report` function is:

```
function report(String message);
```

Within Fabric the report function is primarily used for debugging, whereas it is used for general output from the KL tool.

4.4.2. Numerical Functions

KL has support for many of the “standard library” numerical functions from C. Each of these functions has a version that takes a parameter or parameters of type `Float32`, and another that takes a parameter or parameters of type `Float64`. The one that is called is chosen using polymorphism best-match rules; see Section 4.5, “Function Polymorphism”.

4.4.2.1. Trigonometric Functions

`sin(x)` Returns the sine of the argument `x`.

`cos(x)` Returns the cosine of the argument `x`.

`tan(x)` Returns the tangent of the argument `x`.

`asin(x)` Returns the arcsine of the argument `x`.

`acos(x)` Returns the arccosine of the argument `x`.

`atan(x)` Returns the arctangent of the argument `x`. `atan(x)` doesn't work for large `x` and can only return values in the range $(-\pi/2,\pi/2]$; use the `atan2` function instead when possible.

`atan2(y, x)` Returns the arctangent of the ratio y/x; the result is in the range $(-\pi,\pi]$.

4.4.2.2. Exponential and Logarithmic Functions

`pow(x, y)` Returns the value of x^y.

`exp(x)` Returns the value of e^y where e is the base of the natural logarithm (2.7182818...).

`log(x)` Returns the natural logarithm of `x`.

`log10(x)` Returns the common (base 10) logarithm of `x`.

4.4.2.3. Non-Transcendental Functions

`abs(x)` Returns the absolute value of `x`.

`round(x)` Returns the value of `x` rounded to the nearest whole (fractional part of zero) floating-point number.

`floor(x)` Returns the greatest whole floating-point number less than or equal to `x`.

`ceil(x)` Returns the smallest whole floating-point number greater than or equal to `x`.

4.5. Function Polymorphism

KL support *compile-type function polymorphism*. This means that you can have multiple functions with the same name so long as they have a different number of parameters or those parameters differ by type and/or their input versus input-output qualification.

It is an error to have two functions with the same name that take exactly the same parameter types but return different types

When a function call is made in KL source, if there are multiple functions with the same name then the KL compiler uses a best-match system to determine which function to call. Exact parameter type matches are always prioritized over type casts. If the compiler is unable to choose a unique best match then an error will be reported showing the ambiguity.

Example 4.5. Function polymorphism

```
function display(Integer a) {
  report("integer value is " + a);
}

function display(String s) {
  report("string value is '" + s + "'");
}

operator entry() {
  Integer integer = 42;
  display(integer);

  String string = "hello";
  display(string);

  Byte byte = 64;
  display(byte);
}
```

Output:

```
integer value is 42
string value is 'hello'
integer value is 64
```

4.6. Operators

The `operator` keyword in KL is used to mark functions that are to be used as entry points into KL from the Fabric dependency graph. Operators are declared in the same way as functions except that they must not return a value. Fabric does special type-checking to ensure that operators are bound properly to nodes in a Fabric dependency graph.

Example 4.6. Operator definition

```
operator addElements(io Float32 lhs, io Float32 rhs, io Float32 result) {
  result = lhs + rhs;
}
```

4.7. Constructors

A *constructor* for a user-defined type is a function that initializes a value with the given the type from other values.

4.7.1. Constructor Declarations

A constructor is declared as a function whose name is the name of the user-defined type. The function can take any number of parameters, all of which must be input parameters; constructors cannot take input-output parameters. Constructors cannot return values.

Within the body of a constructor definition, the value being initialized is referred to with the `this` keyword; its members are accessed using the `.` operator. In this context, `this` is always read-write, ie. its members can be modified.

Example 4.7. Constructor declarations

```
struct Complex32 {
  Float32 re;
  Float32 im;
};

// The empty constructor;
function Complex32() {
  this.re = this.im = 0.0;
}

// Construct a Complex from a Float32
function Complex32(Float32 x) {
  this.re = x;
  this.im = 0.0;
}

// Construct a Complex from two Float32s
function Complex32(Float32 x, Float32 y) {
  this.re = x;
  this.im = y;
}

operator entry() {
  report(Complex32());
  report(Complex32(3.141));
  report(Complex32(3.141, 2.718));
}
```

Output:

```
{re:0,im:0}
{re:3.141,im:0}
{re:3.141,im:2.718}
```

4.7.2. Constructor Invocation

Constructors are invoked in one of several ways:

- If a variable is declared without any initialization, the *empty constructor* (ie. the constructor that takes no parameters) is invoked to initialize the variable. This is referred to as *naked initialization*.

Example 4.8. Constructor invocation using naked initialization

```
struct MyType {
  Integer n;
  Float32 x;
};

// The empty constructor
function MyType() {
  this.n = 42;
  this.x = 3.141;
}

operator entry() {
  MyType myType; // invokes the empty constructor
  report(myType);
}
```

Output:

```
{n:42,x:3.141}
```

- If a variable is assigned to as part of its declaration, a single-parameter constructor is invoked. This is referred to as *assignment initialization*. If there isn't an exact match for the type of the value assigned, best-match polymorphism rules are used to choose the constructor to invoke.

Example 4.9. Constructor invocation using assignment initialization

```
struct MyType {
  String string;
};

// Construct from a string
function MyType(String string) {
  this.string = "The string was '" + string + "'";
}

// Construct from a scalar
function MyType(Float64 float64) {
  this.string = "The float64 was " + float64;
}

operator entry() {
  // Construct MyType from String value
  MyType myTypeFromString = "foo";
  report(myTypeFromString);

  // Construct MyType from Float64 value
  MyType myTypeFromFloat64 = 2.718;
  report(myTypeFromFloat64);

  // There is no constructor that takes a Boolean but
  // there is a cast from Boolean to String
  MyType myTypeFromBoolean = true;
  report(myTypeFromBoolean);
}
```

Output:

```
{string:"The string was 'foo'"}
{string:"The float64 was 2.718"}
{string:"The string was 'true'"}
```

- If a variable is “called” (ie. using function call syntax) as part of its declaration, the constructor taking the given arguments is invoked. This is referred to as *invocation initialization*. If there isn't an exact match for the arguments passed to the call, best-match polymorphism rules are used to choose the constructor to invoke.

Example 4.10. Constructor invocation using invocation initialization

```
  struct Vec2 {
    Float64 x;
    Float64 y;
  };

  // Construct from two scalars
  function Vec2(Float64 x, Float64 y) {
    this.x = x;
    this.y = y;
  }

  operator entry() {
    Vec2 vec2FromFloat64s(3.141, 2.718);
    report(vec2FromFloat64s);
    Vec2 vec2FromIntegers(42, -7);  // Uses best-match polymorphism to convert Integer to
 Float64
    report(vec2FromIntegers);
  }
```

Output:

```
  {x:3.141,y:2.718}
  {x:42,y:-7}
```

- If a function call is performed where the name of the function is the name of the type, the constructor taking the given arguments is invoked to create a temporary value of the named type. If there isn't an exact match for the arguments passed to the call, best-match polymorphism rules are used to choose the constructor to invoke.

KL does not distinguish between construction and casting. Casting a value to a different type is the same as constructing a temporary value of the given type and initializing it, using the appropriate constructor, from the given value.

Example 4.11. Constructor invocation using temporary value

```
struct Vec2 {
  Float64 x;
  Float64 y;
};

// Construct from two scalars
function Vec2(Float64 x, Float64 y) {
  this.x = x;
  this.y = y;
}

operator entry() {
  report(Vec2(3.141, 2.718));
  report(Vec2(42, -7));  // Uses best-match polymorphism to convert Integer to Float64
}
```

Output:

```
{x:3.141,y:2.718}
{x:42,y:-7}
```


4.8. Destructors

A destructor is a function that is called when a variable goes out of scope and its resources are freed. Destructors are declared by prepending `~` in front of the name of the type and using it as a function. Destructors cannot take any parameters or return values. The destructor is called before the value is freed so that its members are still accessible. In the body of the destructor the value is referred to using the `this` keyword; the value is input-ouput, ie. it can be modified in the destructor.

Example 4.12. Destructor

```
struct MyType {
  String s;
};

// Empty constructor
function MyType() {
  this.s = "foo";
  report("Creating MyType: this.s = " + this.s);
}

// Destructor
function ~MyType() {
  report("Destroying MyType: this.s = " + this.s);
}

operator entry() {
  MyType myType;
}
```

Output:

```
Creating MyType: this.s = foo
Destroying MyType: this.s = foo
```

4.9. Methods

A *method* is a function that operates on a user-defined structure. It uses a slightly different (and more suggestive) syntax than plain function calls for the case that the function call is strongly tied to a value whose type is a user-defined structure.

4.9.1. Method Definitions

If *`Type`* is a structure or alias, then a method named *`methodName`* can be added to the type using the following syntax:

```
// A method that returns a value
function ReturnType Type.methodName(parameter list) {
  method body
}

// A method that does not return a value
function Type.methodName(parameter list) {
```

```
  method body
}
```

Within the method body, `this` refers to the value on which the method is called. `this` is read-only if the method returns a value and is read-write if the method does not return a value.

4.9.2. Method Invocation

If *`value`* is a value of type *`Type`* then the method *`methodName`* can be invoked on *`value`* using the expression *`value.methodName`*(*`argument list`*). If a method takes no parameters, then it is optional to use parentheses when calling it.

Just as there can be multiple functions with the same name, a given type can have multiple methods with the same name. When deciding which method to invoke, the usual best-match rules apply.

Example 4.13. Method definition and invocation

```
struct Type {
  Integer a;
  Float32 b;
};

// Add method desc to Type
function String Type.desc() {
  return "a:" + this.a + "; b:" + this.b;
}

operator entry() {
  Type t;
  t.a = 1;
  t.b = 3.14;
  // Both report 'a:1; b:3.14'
  report(t.desc());
  report(t.desc);
}
```

Output:

```
a:1; b:3.14
a:1; b:3.14
```

4.9.3. Methods Taking Read-Only or Read-Write Values for `this`

Whether `this` is read-only or read-write (in compiler terms, an r-value or an l-value) can be controlled on a per-method basis. By default, `this` is read-only when the method returns a value and read-write when the method does not return a value. This can be controlled by suffixing the method name with `!` (exclamation mark) to force `this` to be read-write or `?` (question mark) to force `this` to be read-only.

Example 4.14. Explicit read-only or read-write `this` in methods

```
struct Vec2 {
```



```
  Float64 x;
  Float64 y;
};

function Vec2(in Float64 x, in Float64 y) {
  this.x = x;
  this.y = y;
}

// Explicitly make 'this' read-only
function Vec2.getComponents?(io Float64 x, io Float64 y) {
  x = this.x;
  y = this.y;
}

function Float64 Vec2.normSq() {
  return this.x*this.x + this.y*this.y;
}

function Float64 Vec2.norm() {
  return sqrt(this.normSq());
}

function Vec2./=(in Float64 value) {
  this.x /= value;
  this.y /= value;
}

// Explicitly make 'this' read-write
function Float64 Vec2.normalizeAndReturnOldNorm!() {
  Float64 oldNorm = this.norm();
  this /= oldNorm;
  return oldNorm;
}

operator entry() {
  Vec2 vec2(3.14, 2.71);

  Float64 x, y;
  vec2.getComponents(x, y);
  report("vec2.getComponents: x=" + x + ", y=" + y);

  report("vec2.normalizeAndReturnOldNorm returned " + vec2.normalizeAndReturnOldNorm());
  report("vec2 is now " + vec2);
}
```

Output:

```
vec2.getComponents: x=3.14, y=2.71
vec2.normalizeAndReturnOldNorm returned 4.147734321289154
vec2 is now {x:0.7570398093926274,y:0.6533687526923632}
```

4.10. Overloaded Operators

KL allows overloading of binary operators and compound assignment operators for custom types (ie. specified through `struct`).

4.10.1. Binary Operator Overloads

Binary operators can be overloaded using the following syntax:

Example 4.15. Binary operator overload

```
struct Type {
  Integer a;
  Float32 b;
};

function Type +(Type lhs, Type rhs) {
  Type result;
  result.a = lhs.a + rhs.a;
  result.b = lhs.b + rhs.b;
  return result;
}

operator entry() {
  Type t1; t1.a = 42; t1.b = 3.14; report(t1);
  Type t2; t2.a = 7; t2.b = 2.72; report(t2);
  Type t3 = t1 + t2; report(t3);
}
```

Output:

```
{a:42,b:3.14}
{a:7,b:2.72}
{a:49,b:5.86}
```

Any of the binary arithmetic (+, -, *, / and %), bitwise (|, &, ^, << and >>) and comparison (==, !=, <, <=, > and >=) operators can be overloaded.

Binary operator overloads are subject to the following restrictions:

- They must take exactly two parameters. The two parameters may be of any type and the two types may be different but they must both be input-only parameters.
- They must return a value. However, the return type can be any type.

4.10.2. Compound Assignment Overloads

KL provides a default plain assignment for custom types which simply assigns each of the members. It also provides a default *compound assignment* operator (ie. +=, -=, *=, /=, %=, |=, &=, ^=, <<= and >>=) by composing the associated binary operator, if available, with an assignment.

However, it is also possible to provide an overload for any of the compound assignment operators using the following syntax:

Example 4.16. Compound assignment overload

```
struct Type {
  Integer a;
  Float32 b;
};

function Type.+=(Type that) {
  this.a += that.a;
  this.b += that.b;
}

operator entry() {
  Type t1; t1.a = 42; t1.b = 3.14; report("t1 is " + t1);
  Type t2; t2.a = 7; t2.b = 2.72; report("t2 is " + t2);
  t1 += t2; report("t1 is now " + t1);
}
```

Output:

```
t1 is {a:42,b:3.14}
t2 is {a:7,b:2.72}
t1 is now {a:49,b:5.86}
```

Compound assignment overloads are subject to the following restrictions:

- They must take exactly one parameter. The parameter may be of any type but it must be an input-only parameter.
- They must not return a value.

4.11. Global Constants

A *global constant* in KL is a global declaration of the form:

```
const Type name = value;
```

Type must be an integer or floating-point type; *name* must be an identifier; and *value* must be constant of the appropriate type.

It is a compile-time error to try to do any of the following:

- assign to a global constant
- pass a global constant to a function as an input-output parameter
- declare a global constant with the same name as a function, operator or another global constant

Example 4.17. Global constants

```
const Size twoToTheSixteen = 65536;
const Float64 pi = 3.14159365358979;

operator entry() {
  report("twoToTheSixteen = " + twoToTheSixteen);
  report("pi = " + pi);
}
```

Output:

```
twoToTheSixteen = 65536
pi = 3.14159365358979
```

4.12. Importing Functionality With `require`

Through integration with Fabric, it is possible for derived KL types and/or Fabric extensions to provide KL code that is defined externally to the current source file. To use these types and code within the current source file, the `require` statement is provided; it is similar to the `import` statement in Python.

The `require` statement should be followed by the name of the registered type or extension. For example, to include the functionality provided by the extension named “Math” and the registered type named “RegType”, the program should start with:

```
require Math, RegType;
```

Any `require` statements must appear at the top of the KL program that uses the associated functionality. You can have as many `require` statements as you would like.

Chapter 5. Operators and Expressions

This chapter explains the possible operators and resulting expressions in KL. Generally, KL has the same operator and expression syntax as JavaScript and C, and in particular follows exactly the same precedence and associativity rules.

5.1. Operators

KL supports the same basic set of operators as JavaScript and C. These operations are broadly categorized as arithmetic operators, logical operators, bitwise operators and assignment operators.

5.1.1. Arithmetic Operators

5.1.1.1. Add and Subtract

When `+` or `-` appear between two expressions they are referred to as the *add* and *subtract* binary operators. These operators are pre-defined for all integer and floating point types, where they perform the usual arithmetic operations, but they can also be overloaded to apply to user-defined structures or different combinations of types. For example, it is possible to define an operator that takes a user-defined `Rect` on the left and a user-defined `Point` on the right.

In addition to being pre-defined for integer and floating-point types, the `+` (add) operator (but not the `-` [subtract] operator) is defined for the `String` type; the result of adding two strings is the two strings concatenated together.

5.1.1.2. Multiply, Divide and Remainder

When `*`, `/` or `%` appear between two expressions they are referred to as the *multiply*, *divide* and *remainder* binary operators. These operators are pre-defined for all integer and floating point types, where they perform the usual arithmetic operations, but they can also be overloaded to apply to user-defined structures or different combinations of types. For example, it is possible to define a multiply operator that takes a `Float32` on the left and a user-defined `Vec3` on the right.

5.1.1.3. Unary Plus and Minus

When `+` or `-` appears in front of an integer or floating-point expression without an expression to the left they are referred to as the *unary plus* and *unary minus* operators. The unary plus operator doesn't do anything to the value it operates on, but the unary minus operator returns the value that, when added to the original, produces zero (for unsigned integer expressions); for signed integer and floating-point expressions, this is value of the expression with its sign reversed.

It is currently not possible to overload the unary plus and minus operators; this will be fixed in a future version of Fabric.

5.1.1.4. Increment and Decrement Operators

The `++` (increment) and `--` (decrement) operators can be used to add one to or subtract one from a variable. These operators have the following properties:

- The operators only work on variables or input-output parameters; then cannot operate on constants or input parameters. This is because they change the value of the expression.
- They only operate on integer values.

- Each operator can appear either *precede* or *follow* the variable it operates on. When it precedes the variable, the value of the expression is the value of the variable *after* incrementing; this is referred to as a *prefix* increment (or decrement). When it follows the variable, the result is the value of the variable *before* incrementing; this is referred to as a *postfix* increment (or decrement).

- They cannot be overloaded.

5.1.2. Logical Operators

5.1.2.1. Equality Operators

When `==` or `!=` appear between two expressions they are referred to as the *equal-to* or *not-equal-to* binary operators, respectively; collectively, they are referred to as the *equality operators*.

The equality operators are pre-defined for all integer and floating-point types as well as the `Boolean` and `String` types. They can also be overloaded to apply to user-defined structure types or combinations of different types.

5.1.2.2. Relational Operators

When `<`, `<=`, `>` or `>=` appear between two expressions they are referred to as the *less-than*, *less-than-or-equal-to*, *greater-than* or *greater-than-or-equal-to* binary operators, respectively; collectively, they are referred to as the *relational operators*.

The relational operators are pre-defined for all integer and floating-point types as well as the `String` type. They can also be overloaded to apply to user-defined structure types or combinations of different types.

5.1.2.3. Logical AND

When `&&` appears between two expressions it is referred to as the *logical AND* binary operator. Logical AND operates as follows: the left operand is cast to a `Boolean` (ie. a `Boolean` value is constructed from the left hand operand). If the result is `true`, the result is the right operand, otherwise the result is the left operand.

The behaviour of logical AND is the same as in JavaScript but different than C. In C, the result value of a logical AND is always an integer (bool in C++).

It is not possible to overload the logical AND operator. However, you can “enable” it for custom types (structures) by creating a `Boolean` constructor with a single parameter whose type is the type of the left operand.

5.1.2.4. Logical OR

When `||` appears between two expressions it is referred to as the *logical OR* binary operator. Logical OR operates as follows: the left operand is cast to a `Boolean` (ie. a `Boolean` value is constructed from the left hand operand). If the result is `true`, the result is the left operand, otherwise the result is the right operand.

The behaviour of logical OR is the same as in JavaScript but different than C. In C, the result value of a logical AND is always an integer (bool in C++).

It is not possible to overload the logical OR operator. However, you can “enable” it for custom types (structures) by creating a `Boolean` constructor with a single parameter whose type is the type of the left operand.

5.1.2.5. Logical NOT

When `!` (exclamation mark) appears in front of an expression it is referred to as the *logical NOT* unary operator. Logical NOT inverts the logical value of expression; more specifically, it constructs a new `Boolean` value from the expression and then inverts its logical value. Therefore, logical not can be applied to any expression that has a `Boolean` constructor that takes a single parameter whose types is the type of the expression.

It is not possible to overload the logical NOT operator. However, you can "enable" it for custom types (structures) by creating a `Boolean` constructor with a single parameter whose type is the structure.

5.1.2.6. The Conditional Operator

When three expressions are separated by `?` and `:` it is referred to as the *conditional operator* (or *ternary operator*). The conditional operator constructs a `Boolean` from the first operand; if it has value `true`, the result is the second operand, otherwise it is the third.

It is not possible to overload the conditional operator.

5.1.3. Bitwise Operators

5.1.3.1. Bitwise AND, OR and XOR

When &, | or ^ appear between two expressions they are referred to as *bitwise AND*, *bitwise OR* or *bitwise XOR* binary operators, respectively.

Bitwise AND, OR and XOR are predefined for all integer types; they perform the usual bitwise operation on the two values. They are also predefined for the `Boolean` type, which is treated as if it was a single bit with value 1 (if true) or 0 (if false).

Bitwise AND, OR and XOR can be overloaded for user-defined structures or combinations of different types.

5.1.3.2. Bitwise NOT

When ~ (tilde) appears in front of an expression it is referred to as the *bitwise NOT* unary operator.

Bitwise NOT is predefined for all integer types; it inverts the state of the bits of the value. It is also predefined for the `Boolean` type, which is treated as if it was a single bit with value 1 (if true) or 0 (if false).

It is not currently possible to overload the bitwise NOT operator; this will change in a future version of Fabric.

5.1.3.3. Left and Right Shift

When << or >> appear between two expressions they are referred to as the *left shift* or *right shift* binary operators, respectively. These operators are pre-defined for all integer types, where they perform a left or right bit shift of the left operand by the number of bits given in the right operand.

A right-shift of a signed integer value will fill the left most bits with the sign bit, not with zeros. Right shifts of unsigned integer values and left shifts of any integer values always fill with zeros.

It is possible to overload the shift operators for user-defined types, and even provide non-integer types as right-hand operands.

5.1.4. Assignment Operators

5.1.4.1. Direct Assignment Operator

When `=` appears between two expressions it is referred to as the *direct assignment operator*. The direct assignment operator is predefined for all types; see Chapter 3, *The KL Type System* for details on how direct assignment operates for a given type.

5.1.4.2. Compound Assignment Operators

Any of the arithmetic or bitwise (but not logical) binary operators can be combined with `=` to form a *compound assignment operator*; these are specifically `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=` and `|=`.

A compound assignment operator is predefined for a given type if and only if the corresponding binary operator is predefined for the type. It is also possible to overload the compound assignment operator for any type, and it is even possible to have different types for the left and right operands. See Section 4.10.2, “Compound Assignment Overloads”.

5.1.5. Operators and Polymorphism

Operator invocations are subject to the same rules as function calls with respect to polymorphism. If an exact match for an operator with the parameter types equal to the operand types is not found, KL will find the best-match among the existing implementations of the operator. This makes it possible, for instance, to add an integer and a string; the result is that the integer is cast to a string and then the strings are concatenated.

For more information on polymorphism and best-match rules, see Section 4.5, “Function Polymorphism”.

5.2. Expressions

There are two types of expressions in KL: simple expressions and compound expressions.

5.2.1. Simple Expressions

Simple expressions are the expressions from which more complex expressions are derived. The simple expressions are:

- Symbols that refer to variables, function arguments or global constants. The type of the expression is the type of the entity referred to. Examples: `foo`, `myParam`, `mathPI`. See Section 5.4, “Scoping Rules” for how symbol names are resolved.
- Boolean, integer, floating-point and string constants. The type of the expression is the type of the constant. Examples: `true`, `42`, `3.14159`. See Section 2.4.4, “Constants”.

5.2.2. Compound Expressions

Compound expressions are built from simple expressions and/or other compound expressions using operators.

The following table lists all the different compound expressions in KL. Compound expressions are grouped by *type*; all expressions of the same type are of the same precedence and share the same associativity. Compound expression types are listed from highest to lowest precedence.

Table 5.1. Compound expressions

Type	Associativity	Expression(s)	Reference
Postfix	left-to-right	*function*(*args*)	Section 4.2, “Function Invocations”
		expr[*expr*]	Section 3.2.2, “Arrays”; Section 3.2.3, “Dictionaries”
		expr.*member*	Section 3.2.1, “Structures”
		expr.*method*(*args*)	Section 4.9.2, “Method Invocation”
		expr++	Section 5.1.1.4, “Increment and Decrement Operators”
		expr--	
Prefix	right-to-left	+*expr*	Section 5.1.1.3, “Unary Plus and Minus”
		-*expr*	
		++*expr*	Section 5.1.1.4, “Increment and Decrement Operators”
		--*expr*	
		!*expr*	Section 5.1.2.5, “Logical NOT”
		~*expr*	Section 5.1.3.2, “Bitwise NOT”
Multiplicative	left-to-right	*expr* * *expr*	Section 5.1.1.2, “Multiply, Divide and Remainder”
		expr / *expr*	
		expr % *expr*	
Additive	left-to-right	*expr* + *expr*	Section 5.1.1.1, “Add and Subtract”
		expr - *expr*	
Shift	left-to-right	*expr* << *expr*	Section 5.1.3.3, “Left and Right Shift”
		expr >> *expr*	
Relational	left-to-right	*expr* < *expr*	Section 5.1.2.2, “Relational Operators”
		expr <= *expr*	
		expr > *expr*	
		expr >= *expr*	
Equality	left-to-right	*expr* == *expr*	Section 5.1.2.1, “Equality Operators”
		expr != *expr*	
Bitwise AND	left-to-right	*expr* & *expr*	Section 5.1.3.1, “Bitwise AND, OR and XOR”
Bitwise XOR	left-to-right	*expr* ^ *expr*	
Bitwise OR	left-to-right	*expr* | *expr*	
Logical AND	left-to-right	*expr* && *expr*	Section 5.1.2.3, “Logical AND”
Logical OR	left-to-right	*expr* || *expr*	Section 5.1.2.4, “Logical OR”
Conditional	right-to-left	*expr*? *expr*: *expr*	Section 5.1.2.6, “The Conditional Operator”

Type	Associativity	Expression(s)	Reference
Assignment	right-to-left	*expr* = *expr*	Section 5.1.4.1, “Direct Assignment Operator”
		expr += *expr*	Section 5.1.4.2, “Compound Assignment Operators”
		expr -= *expr*	
		expr *= *expr*	
		expr /= *expr*	
		expr %= *expr*	
		expr <<= *expr*	
		expr >>= *expr*	
		expr &= *expr*	
		expr ^= *expr*	
		expr |= *expr*	

5.3. Controlling Order of Operations

The order of operations can be explicitly controlled by putting (and) (parentheses) around expressions.

Example 5.1. Controlling Order of Operations

```
operator entry() {
  report( (2 * 3) + 5 );
  report( 2 * (3 + 5) );
}
```

Output:

```
11
16
```

5.4. Scoping Rules

The term *scope* in programming languages refers to the parts of a program in which variables, constants and functions are visible; the rules that govern how scopes work are referred to as *scoping rules*. Scopes are also responsible for managing the “lifecycle” of variables; in the case of KL, this refers to when destructors are called for structure values.

5.4.1. Types of Scopes

In KL, there are four kinds scopes: the global scope, function scopes, compound statement scopes and temporary scopes. Scopes *nest* inside each other; when a KL program refers to a variable by name, the compiler determines which variable is being referred to by searching from the current innermost scope outwards through the nested scopes to the outermost scope (which is always the global scope). Like C, KL is a statically-scoped language, meaning that the exact variable that is being referred to is resolved at compile time (and not at run time).

5.4.1.1. The Global Scope

The global scope is the top-level scope of a KL program. The symbols that are visible in the global scope are global constants as well as functions and operators. The global scope is always the outermost scope at any point in a KL program.

5.4.1.2. Function Scope

Whenever a function or operator is defined, a function scope is created that is nested inside the global scope. The arguments to the function are provided within the function scope. Within the function scope, a function definition also creates a compound statement scope corresponding to the compound statement that constitutes the body of the function.

5.4.1.3. Compound Statement Scope

Any time that a `{` followed by zero or more statements followed by `}` is used to introduce a compound statement, a new *compound statement scope* is introduced. Compound statement scopes are nested inside function scopes (when they correspond to the compound statement that constitutes the body of a function) or inside other compound statement scopes.

When control reaches the end of a compound statement (by executing the last statement or via the `return`, `break` or `continue` statements), any structure values that have corresponding destructors will have those destructors executed.

Note that declaring a loop index variable inside a loop statement is a special case of a compound statement scope. In the case that the loop body is a compound statement, the corresponding compound statement scope is nested inside the loop's compound statement scope.

5.4.1.4. Temporary Scope

Any time that a constructor is directly invoked to create a temporary value (see ???), a scope is created to contain the temporary value. The scope encloses the surrounding expression of the temporary value; this means that when the surrounding expression is finished execution, the temporary value's destructor, if it exists, is executed.

5.4.2. Nested Scopes Example

For a precise understanding of the nesting of KL scopes, study the following example carefully!

Example 5.2. Nested scopes

```
struct T {
  String s;
};

function T(String s) {
  this.s = s;
  report("created T; s = " + this.s);
}

function ~T() {
  report("destroying T; s = " + this.s);
}

function foo(T t) {
  report("start of foo; t is now " + t);
  T t("fooT");
  report("declared t in foo; t is now " + t);
  for (Index i=0; i<3; ++i) {
    report("top of loop body; t is now " + t);
    T t("loopT:" + i);
    report("declared t in loop; t is now " + t);
  }
  report("after loop; t is now " + t);
}

const Float32 t = 2.75;

operator entry() {
  report("top of entry; t is now " + t);
  T t("entryT");
  report("declared t in entry; t is now " + t);
  foo(t);
  report("came back from foo; t is now " + t);
}
```

Output:

```
top of entry; t is now 2.75
created T; s = entryT
declared t in entry; t is now {s:"entryT"}
start of foo; t is now {s:"entryT"}
created T; s = fooT
declared t in foo; t is now {s:"fooT"}
top of loop body; t is now {s:"fooT"}
created T; s = loopT:0
declared t in loop; t is now {s:"loopT:0"}
destroying T; s = loopT:0
top of loop body; t is now {s:"fooT"}
created T; s = loopT:1
declared t in loop; t is now {s:"loopT:1"}
destroying T; s = loopT:1
top of loop body; t is now {s:"fooT"}
created T; s = loopT:2
declared t in loop; t is now {s:"loopT:2"}
destroying T; s = loopT:2
after loop; t is now {s:"fooT"}
destroying T; s = fooT
came back from foo; t is now {s:"entryT"}
destroying T; s = entryT
```


Chapter 6. Statements

The bodies of functions, operators, and so on are composed of *statements*. Statements are the parts of KL programs that actually do something. Statements cause expressions to be evaluated, can conditionally execute other statements, or can iterate over another set of statements until a condition is met.

The remainder of the chapter describes the different statements that can be used inside KL function definitions. Statements are divided into two categories: simple statements and complex statements.

6.1. Simple Statements

A simple statement is a statement that does not check conditions or create a nested scope. Simple statements always end with a `;` (semicolon) character.

It is a common syntax error in KL and other C-like languages to forget to finish a simple statement with a semicolon.

6.1.1. Expression Statements

Any expression, followed by a `;` (semicolon), is a statement. The statement operates by evaluating the expression and then discarding the resulting value (if any). The most common form of an expression statement is one that invokes a function call (eg. `report`).

Example 6.1. Expression statements

```
operator entry() {
  // A very useless expression statement:
  // evaluates 2 + 2 then discards the
  // result
  2 + 2;

  // A more useful expression statement:
  // evaluates the report function, which
  // causes text to appear
  report("Hello!");
}
```

6.1.2. The `throw` Statement

The keyword `throw`, followed by any argument, causes the argument to a string and then throws an exception whose value is the string. This can be used to throw an exception, exiting out of the KL environment entirely and returning the exception to the calling environment (ie. the dynamic language using KL).

Currently there is no way of catching exceptions from within KL itself. This functionality will be added in a later version of Fabric Engine.

Example 6.2. The `throw` statement

```
operator entry() {
  report("before throw");
  throw "Test Exception";
  report("after throw");
}
```

Output:

```
before throw
Caught exception: Test Exception
```

6.1.3. The Empty Statement

A `;` (semicolon) alone is a statement that does nothing. This can be used in cases where a statement is required but there is nothing to do.

Example 6.3. The empty statement

```
function foo(io MyType myType) {
  for (; returnsFalseWhenDone(myType); )
    ; // Do nothing
}
```

6.1.4. Variable Declaration Statements

A *variable declaration statement* introduces one or more new variables into the innermost scope. These variables remain visible for the rest of the scope and are destroyed, executing destructors when appropriate, when execution exists this scope.

Variable declarations require a type. Multiple variables can be declared in the same statement by separating them with commas.

The variable names can be followed by array and dictionary specifications. See Section 3.2.2, “Arrays” and Section 3.2.3, “Dictionaries” for more information on array and dictionary specifications.

Variables can be constructed or assigned to when declared to set their initial values; see Section 4.7, “Constructors”.

Example 6.4. Variable declaration statements

```
operator entry() {
  Scalar a;
  report(a);
  Integer b[], c[2][2], d;
  report(b);
  report(c);
  report(d);
}
```

Output:

```
0
[]
[[0,0],[0,0]]
0
```

As of this writing, it is not yet possible to initialize arrays in KL in the statement where the variable is declared.

6.2. Complex Statements

A *complex statement* is a statement that groups together and/or conditionally executes other statements. Complex statements are built from simple statements, other complex statements, and expressions.

6.2.1. Compound Statements

At any point when a single statement can be inserted in KL source code, it is possible to insert multiple statements surrounded by `{` (left brace) and `}` (right brace). This has the following effects:

- The statements are executed, in order, as if they were together a single statement; and
- A new, nested scope is introduced; see Section 5.4, “Scoping Rules”.

6.2.2. Conditional Statements

6.2.2.1. The `if` Statement

The `if` keyword begins a conditional statement, which can optionally include an `else` clause. As in JavaScript and C, `if` and `else` statements can chain.

Example 6.5. The `if` statement

```
function String desc(Integer n) {
  if (n == 0)
    return "zero";
  else if (n == 1)
    return "one";
  else
    return "lots";
}
```

6.2.2.2. The `switch` Statement

The `switch`...`case` construct is a more compact form for a sequence of `if`...`else` statements, just as in JavaScript and C.

Example 6.6. The `switch` statement

```
function String descNumber(Integer i) {
  switch (i) {
    case 0:
      return "zero";
    case 1:
      return "one";
    case 2:
    case 3;
    case 4:
      return "a few";
    default:
      return "many";
  }
}
```

6.2.3. The `return` Statement

The `return` statement takes two forms:

- For a function which returns a value, the `return` statement immediately returns the given value to the calling function. There is an implicit cast to the type of the return value as specified in the function definition.
- For a function which does not return a value, the `return` statement does not take a value and simply returns immediately to the calling function.

Example 6.7. The `return` statement

```
// Function that returns a value
function String markupName(String name) {
  return "My name is '" + name + "'";
}

// Function that does not return a value
function suffixIndexDesc(io String strings[], Size index) {
  if (index >= strings.size)
    return;
  strings[index] += " (was at index " + index + ")";
}
```

6.2.4. Loop Statements

6.2.4.1. “C-Style” Loops

A *C-style loop* takes the form `for (start; check; next) body`, where `start` is a statement (possibly empty), `check` and `next` are optional expressions, and `body` is a statement. The loop operates as follows:

- A new, nested scope is created. This scope is destroyed when the loop finishes.
- The `start` statement is executed.
- The `check` expression, if present, is evaluated, and the resulting value is converted to a `Boolean`. If the result is false, the loop finishes.
- The `body` statement is executed.
- The `next` expression, if present, is evaluated, and the resulting value is discarded. Execution is transferred back to the step that evaluates the `check` expression.

Note that since `start` is a statement, it is possible to declare a new variable there. This variable will go out-of-scope when the loop finishes. It is common practice to declare loop-bound index variables in the `start` statement.

Example 6.8. “C-Style” loop

```
operator entry() {
  for (Index i=0; i<10; ++i)
    report("i is now " + i);
}
```

6.2.4.2. `while` Loops

A *while loop* takes the form `while (check) body`, where `check` is a expression and `body` is a statement. The loop operates as follows:

- The `check` expression is evaluated, and the resulting value is converted to a `Boolean`. If the result is false, the loop finishes.

- The *body* statement is executed.
- Execution is transferred back to the step that evaluates the *check* expression.

Example 6.9. `while` loop

```
operator entry() {
  Integer i = 0;
  while (i < 10) {
    report("i is now " + i);
    ++i;
  }
}
```

6.2.4.3. do...while Loops

A *do-while loop* takes the form `do` *body* `while (`*check*`);`, where *check* is a expression and *body* is a statement. The loop operates as follows:

- The *body* statement is executed.
- The *check* expression is evaluated, and the resulting value is converted to a `Boolean`. If the result is false, the loop finishes.
- Execution is transferred back to the step that executes the *body* statement.

Example 6.10. do...while loop

```
operator entry() {
  Integer i=0;
  do {
    report("i is now " + i);
    ++i;
  } while (i<10);
}
```

6.2.4.4. Dictionary Loops

A dictionary loop iterates over all the keys and values, or just the values, in a dictionary. For more information on dictionary loops, see Section 3.2.3, “Dictionaries”.

6.2.4.5. Loop Control Statements

Within the body of a loop, the `break` and `continue` statements can be used to prematurely end the current iteration of the loop as described below.

6.2.4.5.1. The `break` Statement

The `break;` statement immediately exits the innermost loop. It is an error to use the `break` statement outside of a loop.

Example 6.11. The `break` statement

```
operator entry() {
  // Only loops 5 times
  for (Integer i=0; ; ++i) {
    if (i == 5)
      break;
  }
}
```

6.2.4.5.2. The `continue` Statement

The `continue;` statement immediately jumps to the next iteration of the innermost loop. It is an error to use the `continue` statement outside of a loop.

Example 6.12. The `continue` statement

```
operator entry() {
  // Only prints 7
  for (Integer i=0; i<10; ++i) {
    if (i != 7)
      continue;
    report(i);
  }
}
```

Chapter 7. Map-Reduce

Fabric Engine provides a generic map-reduce framework that can be used to create recursively-parallel operations on large data sets.

This chapter does not attempt to explain the concepts behind and usage of the Fabric Engine map-reduce framework; it serves purely to enumerate the types and functions that are provided in KL to support the framework. For more information on concepts and usage of the map-reduce framework, refer to the "Map-Reduce Programming Guide" [http://documentation.fabricengine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] .

7.1. Map-Reduce Types

In order to support map-reduce, KL introduces two new derived types: `ValueProducer<...>` and `ArrayProducer<...>`.

7.1.1. The `ValueProducer<...>` Type

Given an existing type *`ValueType`*, the type `ValueProducer<`*`ValueType`*`>` is a map-reduce value producer that produces values of type *`ValueType`*. Values of type `ValueProducer<`*`ValueType`*`>` have the following properties:

- They can be assigned to variables of the same type; however, there must be an exact match for *`ValueType`*. In other words, `ValueProducer<`*`ValueType1`*`>` and `ValueProducer<`*`ValueType2`*`>` are the same type if and only if *`ValueType1`* and *`ValueType2`* are the same type.
- They support a method

  ```
  function ValueType ValueProducer<ValueType>.produce();
  ```

  that produces the value producer's value.
- They support a method

  ```
  function ValueProducer<ValueType>.flush();
  ```

  that recursively flushes any caches connected to the value producer.

7.1.2. The `ArrayProducer<...>` Type

Given an existing type *`ElementType`*, the type `ArrayProducer<`*`ElementType`*`>` is a map-reduce array producer that produces values of type *`ElementType`*. Values of type `ArrayProducer<`*`ElementType`*`>` have the following properties:

- They can be assigned to variables of the same type; however, there must be an exact match for *`ElementType`*. In other words, `ArrayProducer<`*`ElementType1`*`>` and `ArrayProducer<`*`ElementType2`*`>` are the same type if and only if *`ElementType1`* and *`ElementType2`* are the same type.
- They support a method

  ```
  function Size ArrayProducer<ElementType>.getCount();
  ```

  that returns the number of elements in the array producer. Calling `produce(`*`i`*`)` (below) with *`i`* not less than the result of `getCount()` will cause an exception to be thrown.

- They support a method

```
function ElementType ArrayProducer<ElementType>.produce(Index i);
```

  that produces the array producer's element at index *i*.

- They support a method

```
function ArrayProducer<ElementType>.flush();
```

  that recursively flushes any caches connected to the array producer.

7.2. Map-Reduce Functions

KL supports several functions to support the creation of new value and array producers from existing producers as well as KL functions and values.

7.2.1. Value Producer Creation Functions

7.2.1.1. createConstValue

```
ValueProducer<ValueType> createConstValue(expr)
```

Creates a value producer with value type *ValueType* that returns a constant value. *expr* must be an expression that evaluates to a value.

Example 7.1. createConstValue

```
operator entry() {
  ValueProducer<Integer> vp = createConstValue( 42 );
  report(vp);
  report(vp.produce());
}
```

Output:

```
]
ValueProducer<Integer>
42
```

7.2.1.2. createValueGenerator

```
ValueProducer<ValueType> createValueGenerator(operatorName[, ValueProducer<SharedValueType>])
```

Creates a value producer with value type *ValueType* that calls the operator named *operatorName* to generate the value.

- If ValueProducer<*SharedValueType*> is provided, *operatorName* must name an operator with prototype

```
operator operatorName(io ValueType value, SharedValueType sharedValue);
```


- Otherwise, *operatorName* must name an operator with prototype

```
operator operatorName(io ValueType value);
```

Example 7.2. createValueGenerator

```
operator gen1(io String output) {
  output = "Hello!";
}

operator gen2(io String output, String shared) {
  output = "Hello, " + shared;
}

operator entry() {
  ValueProducer<String> vp1 = createValueGenerator(gen1);
  report("vp1 = " + vp1);
  report("vp1.produce() = " + vp1.produce());

  ValueProducer<String> vp2 = createValueGenerator(gen2, vp1);
  report("vp2 = " + vp2);
  report("vp2.produce() = " + vp2.produce());
}
```

Output:

```
]
vp1 = ValueProducer<String>
vp1.produce() = Hello!
vp2 = ValueProducer<String>
vp2.produce() = Hello, Hello!
```

7.2.1.3. createValueTransform

```
ValueProducer<ValueType> createValueTransform(ValueProducer<ValueType>, operatorName[,
 ValueProducer<SharedValueType>])
```

Creates a value producer with value type *ValueType* that calls the operator named *operatorName* to transform the result of another value producer of the same type.

- If ValueProducer<*SharedValueType*> is provided, *operatorName* must name an operator with prototype

```
operator operatorName(io ValueType value, SharedValueType sharedValue);
```

- Otherwise, *operatorName* must name an operator with prototype

```
operator operatorName(io ValueType value);
```

Example 7.3. `createValueTransform`

```
operator multByTwo(
  io Integer value
  )
{
  value *= 2;
}

operator multByShared(
  io Integer value,
  Integer shared
  )
{
  value *= shared;
}

operator entry() {
  ValueProducer<Integer> vp;

  vp = createValueTransform(
    createConstValue( 42 ),
    multByTwo
    );
  report(vp);
  report(vp.produce());

  vp = createValueTransform(
    createConstValue( 2 ),
    multByShared,
    createConstValue( 3 )
    );
  report(vp);
  report(vp.produce());
}
```

Output:

```
]
ValueProducer<Integer>
84
ValueProducer<Integer>
6
```

7.2.1.4. `createValueMap`

```
ValueProducer<OutputValueType> createValueMap(ValueProducer<InputValueType>, operatorName[,
 ValueProducer<SharedValueType>])
```

Creates a value producer with value type *`OutputValueType`* that calls the operator named *`operatorName`* to map the result of another value producer of type `ValueProducer<`*`InputValueType`*`>`.

- If `ValueProducer<`*`SharedValueType`*`>` is provided, *`operatorName`* must name an operator with prototype

  ```
  operator operatorName(InputValueType inputValue, io OutputValueType
   outputValue, SharedValueType sharedValue);
  ```

- Otherwise, *`operatorName`* must name an operator with prototype

  ```
  operatorName(InputValueType inputValue, io OutputValueType outputValue);
  ```


Example 7.4. `createValueMap`

```
operator multByPi(
  Integer input,
  io Scalar output
  )
{
  output = 3.14 * input;
}

operator multByShared(
  Integer input,
  io Scalar output,
  Scalar shared
  )
{
  output = input * shared;
}

operator entry() {
  ValueProducer<Scalar> vp;

  vp = createValueMap(
    createConstValue( 42 ),
    multByPi
    );
  report(vp);
  report(vp.produce());

  vp = createValueMap(
    createConstValue( 2 ),
    multByShared,
    createConstValue( Scalar(2.71) )
    );
  report(vp);
  report(vp.produce());
}
```

Output:

```
]
ValueProducer<Scalar>
131.88
ValueProducer<Scalar>
5.42
```

7.2.1.5. `createValueCache`

```
ValueProducer<ValueType> createValueCache(ValueProducer<ValueType>)
```

Creates a value producer with value type *`ValueType`* that caches the result of another value producer of the same type. For more information on producer caches, see refer to the "Map-Reduce Programming Guide" [http://documentation.fabric-engine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] .

Example 7.5. `createValueCache`

```
operator valueGen(
  io Scalar output
  )
{
  output = 2.71;
  report("Generating output = " + output);
}

operator entry() {
  ValueProducer<Scalar> vp = createValueCache(createValueGenerator(valueGen));
  report("vp.produce() = " + vp.produce());
  report("vp.produce() = " + vp.produce());
  report("calling vp.flush()");
  vp.flush();
  report("vp.produce() = " + vp.produce());
  report("vp.produce() = " + vp.produce());
}
```

Output:

```
]
Generating output = 2.71
vp.produce() = 2.71
vp.produce() = 2.71
calling vp.flush()
Generating output = 2.71
vp.produce() = 2.71
vp.produce() = 2.71
```

7.2.2. Array Producer Creation Functions

7.2.2.1. `createConstArray`

```
ArrayProducer<ElementType> createConstArray(arrayExpr)
```

Creates a constant array producer. *`arrayExpr`* must be an expression that evaluates to an array (variable-size, fixed-size or sliced) whose element type is *`ElementType`*. The array is "frozen" in the resulting array producer; that is, if the array used in the call to `createConstArray` is later changed, the result of `createConstArray` will not change.

Example 7.6. `createConstArray`

```
operator entry() {
  Integer a[]; a.push(42); a.push(17); a.push(52); a.push(871);
  ArrayProducer<Integer> ap = createConstArray(a);
  report(ap);
  report(ap.getCount());
  report(ap.produce(2));
}
```

Output:

```
ArrayProducer<Integer>
```



```
4
52
```

7.2.2.2. `createArrayGenerator`

`createArrayGenerator` has one of the following "prototypes":

```
function ArrayProducer<ElementType> createArrayGenerator(
  ValueProducer<Size> countProducer,
  operatorName
  )

function ArrayProducer<ElementType> createArrayGenerator(
  ValueProducer<Size> countProducer,
  operatorName,
  ValueProducer<SharedValueType> sharedValueProducer
  )
```

`createArrayGenerator` creates an array producer with element type *`ElementType`* that calls the operator named *`operatorName`* to generate the elements of the array.

If `sharedValueProducer` is provided, *`operatorName`* must name an operator with prototype

```
operator operatorName(
  io ElementType elementValue,
  Size index,
  Size count,
  SharedValueType sharedValue
  );
```

Otherwise, it must name an operator with any of the following prototypes:

```
operator operatorName(
  io ElementType elementValue,
  Size index,
  Size count
  );

operator operatorName(
  io ElementType elementValue,
  Size index
  );

operator operatorName(
  io ElementType elementValue
  );
```

Example 7.7. `createArrayGenerator`

```
operator generator(
  io Float32 value,
  Size index,
  Size count,
  Float32 shared
  )
{
  report("generator: value=" + value + " index=" + index + " count=" + count + " shared=" +
 shared);
  value = shared*index;
}

operator entry() {
  ValueProducer<Size> cvg = createConstValue(Size(10));
  ArrayProducer<Float32> ag4 = createArrayGenerator(cvg, generator,
 createConstValue(Float32(3.141)));
  report(ag4);
  report(ag4.getCount());
  for (Size i=0; i<10; ++i)
    report(ag4.produce(i));
}
```

Output:

```
ArrayProducer<Float32>
10
generator: value=0 index=0 count=10 shared=3.141
0
generator: value=0 index=1 count=10 shared=3.141
3.141
generator: value=0 index=2 count=10 shared=3.141
6.282
generator: value=0 index=3 count=10 shared=3.141
9.423
generator: value=0 index=4 count=10 shared=3.141
12.564
generator: value=0 index=5 count=10 shared=3.141
15.705
generator: value=0 index=6 count=10 shared=3.141
18.846
generator: value=0 index=7 count=10 shared=3.141
21.987
generator: value=0 index=8 count=10 shared=3.141
25.128
generator: value=0 index=9 count=10 shared=3.141
28.269
```

7.2.2.3. `createArrayTransform`

`createArrayTransform` has one of the following “prototypes”:

```
function ArrayProducer<ElementType> createArrayTransform(
  ArrayProducer<ElementType> inputArrayProducer,
  operatorName
  )

function ArrayProducer<ElementType> createArrayTransform(
  ArrayProducer<ElementType> inputArrayProducer,
  operatorName,
```



```
  ValueProducer<SharedValueType> sharedValueProducer
  )
```

`createArrayTransform` creates an array producer with element type *`ElementType`* that transforms `inputArrayProducer` by calling the operator named *`operatorName`* to transform the individual elements.

If `sharedValueProducer` is provided, *`operatorName`* must name an operator with prototype

```
operator operatorName(
  io ElementType elementValue,
  Size index,
  Size count,
  SharedValueType sharedValue
  );
```

Otherwise, it must name an operator with any of the following prototypes:

```
operator operatorName(
  io ElementType elementValue,
  Size index,
  Size count
  );

operator operatorName(
  io ElementType elementValue,
  Size index
  );

operator operatorName(
  io ElementType elementValue
  );
```

Example 7.8. `createArrayTransform`

```
operator transform(
  io Float32 value,
  Size index,
  Size count,
  Float32 shared
  )
{
  report("transform: value=" + value + " index=" + index + " count=" + count + " shared=" +
 shared);
  value *= shared * (index + 1);
}

operator entry() {
  Float32 inputArray[]; inputArray.push(5.6); inputArray.push(-3.4); inputArray.push(1.4142);
  ArrayProducer<Float32> inputArrayProducer = createConstArray(inputArray);
  ArrayProducer<Float32> transformedArrayProducer = createArrayTransform(inputArrayProducer,
 transform, createConstValue(Float32(2.56)));
  report(transformedArrayProducer);
  report(transformedArrayProducer.getCount());
  Size transformedArrayProducerCount = transformedArrayProducer.getCount();
  for (Index i=0; i<transformedArrayProducerCount; ++i)
    report(transformedArrayProducer.produce(i));
}
```

Output:

```
ArrayProducer<Float32>
3
transform: value=5.6 index=0 count=3 shared=2.56
14.336
transform: value=-3.4 index=1 count=3 shared=2.56
-17.408
transform: value=1.4142 index=2 count=3 shared=2.56
10.86106
```

7.2.2.4. `createArrayMap`

`createArrayMap` has one of the following "prototypes":

```
function ArrayProducer<OutputElementType> createArrayMap(
  ArrayProducer<InputElementType> inputArrayProducer,
  operatorName
  )

function ArrayProducer<OutputElementType> createArrayMap(
  ArrayProducer<InputElementType> inputArrayProducer,
  operatorName,
  ValueProducer<SharedValueType> sharedValueProducer
  )
```

`createArrayMap` creates an array producer with element type *`OutputElementType`* that maps `inputArrayProducer` of a potentially different element type *`InputElementType`* by calling the operator named *`operatorName`* to map the individual elements.

If `sharedValueProducer` is provided, *`operatorName`* must name an operator with prototype

```
operator operatorName(
  InputElementType inputElementValue,
  io OutputElementType outputElementValue,
  Size index,
  Size count,
  SharedValueType sharedValue
  );
```

Otherwise, it must name an operator with any of the following prototypes:

```
operator operatorName(
  InputElementType inputElementValue,
  io OutputElementType outputElementValue,
  Size index,
  Size count
  );

operator operatorName(
  InputElementType inputElementValue,
  io OutputElementType outputElementValue,
  Size index
  );

operator operatorName(
  InputElementType inputElementValue,
  io OutputElementType outputElementValue
  );
```


Example 7.9. `createArrayMap`

```
operator map(
  Float32 inputValue,
  io String outputValue,
  Size index,
  Size count,
  Float32 shared
  )
{
  report("map: inputValue=" + inputValue + " index=" + index + " count=" + count + " shared=" +
 shared);
  Float32 float32Value = inputValue * shared * (index + 1);
  if (abs(float32Value) < 1.0)
    outputValue = "small";
  else if (abs(float32Value) < 10.0)
    outputValue = "medium";
  else if (abs(float32Value) < 100.0)
    outputValue = "large";
  else
    outputValue = "x-large";
}

operator entry() {
  Float32 inputArray[]; inputArray.push(5.6); inputArray.push(-0.034); inputArray.push(1.4142);
  ArrayProducer<Float32> inputArrayProducer = createConstArray(inputArray);
  ArrayProducer<String> mappedArrayProducer = createArrayMap(inputArrayProducer, map,
 createConstValue(Float32(2.56)));
  report(mappedArrayProducer);
  report(mappedArrayProducer.getCount());
  Size mappedArrayProducerCount = mappedArrayProducer.getCount();
  for (Index i=0; i<mappedArrayProducerCount; ++i)
    report(mappedArrayProducer.produce(i));
}
```

Output:

```
ArrayProducer<String>
3
map: inputValue=5.6 index=0 count=3 shared=2.56
large
map: inputValue=-0.034 index=1 count=3 shared=2.56
small
map: inputValue=1.4142 index=2 count=3 shared=2.56
large
```

7.2.2.5. `createArrayCache`

```
ArrayProducer<ElementType> createArrayCache(ArrayProducer<ElementType>)
```

Creates an array producer with array type *`ElementType`* that caches the result of another array producer of the same type. For more information on producer caches, refer to the "Map-Reduce Programming Guide" [http://documentation.fabric-engine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] .

Example 7.10. `createArrayCache`

```
operator elementGen(
  io Scalar output,
  Size index
  )
{
  output = 2.71 * index;
  report("Generating output = " + output);
}

operator entry() {
  ArrayProducer<Scalar> ap = createArrayCache(
    createArrayGenerator(
      createConstValue(Size(4)),
      elementGen
      )
    );
  report("ap.produce(2) = " + ap.produce(2));
  report("ap.produce(2) = " + ap.produce(2));
  report("calling ap.flush()");
  ap.flush();
  report("ap.produce(2) = " + ap.produce(2));
  report("ap.produce(2) = " + ap.produce(2));
}
```

Output:

```
Generating output = 5.42
ap.produce(2) = 5.42
ap.produce(2) = 5.42
calling ap.flush()
Generating output = 5.42
ap.produce(2) = 5.42
ap.produce(2) = 5.42
```

7.2.3. Reduce Creation Functions

7.2.3.1. `createReduce`

`createReduce` has one of the following “prototypes”:

```
function ValueProducer<OutputValueType> createReduce(
  ArrayProducer<InputElementType> inputArrayProducer,
  operatorName
  )

function ValueProducer<OutputValueType> createReduce(
  ArrayProducer<InputElementType> inputArrayProducer,
  operatorName,
  ValueProducer<SharedValueType> sharedValueProducer
  )
```

`createReduce` creates a value producer with value type *`OutputValueType`* that reduces `inputArrayProducer` by calling the operator named *`operatorName`* to contribute the individual elements to the reduction.

If `sharedValueProducer` is provided, *`operatorName`* must name an operator with prototype


```
operator operatorName(
  InputElementType inputElementValue,
  io OutputValueType outputValue,
  Size index,
  Size count,
  SharedValueType sharedValue
  );
```

Otherwise, it must name an operator with any of the following prototypes:

```
operator operatorName(
  InputElementType inputElementValue,
  io OutputValueType outputValue,
  Size index,
  Size count
  );

operator operatorName(
  InputElementType inputElementValue,
  io OutputValueType outputValue,
  Size index
  );

operator operatorName(
  InputElementType inputElementValue,
  io OutputValueType outputValue
  );
```

Example 7.11. `createReduce`

```
operator generator(
  io Integer outputValue,
  Size index
  )
{
  outputValue = index + 1;
}

operator reduce(
  Integer inputValue,
  io Integer outputValue
  )
{
  report("reduce: inputValue=" + inputValue);
  outputValue += inputValue;
}

operator entry() {
  ValueProducer<Integer> sum = createReduce(
    createArrayGenerator(
      createConstValue(Size(10)),
      generator
      ),
    reduce
    );
  report("sum.produce() = " + sum.produce());
}
```

Output:

```
```

```
reduce: inputValue=1
reduce: inputValue=2
reduce: inputValue=3
reduce: inputValue=4
reduce: inputValue=5
reduce: inputValue=6
reduce: inputValue=7
reduce: inputValue=8
reduce: inputValue=9
reduce: inputValue=10
sum.produce() = 55
```

